Dispensa di

Sistemi Operativi

Autore: Gabriele Frassi

Università di Pisa
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea triennale in Ingegneria Informatica

Prof. Marco Avvenuti

A.A.2021-2022

Sommario

1 Le mie premesse

Il corso mi ha dato molte soddisfazioni, gli argomenti sono interessanti e il professore ha molta passione. La dispensa è stata realizzata nell'ottica di una pubblicazione, ponendo i concetti in modo tale da essere comprensibili anche a chi non ha seguito le lezioni. Per quanto riguarda le lezioni di laboratorio dell'ing.Minici ho deciso di non includerle nella dispensa, visto le diapositive messe a disposizione dal docente: per eventuali dubbi consiglio la dispensa di Giacomo Sansone caricata su https://github.com/Guray00/IngegneriaInformatica.

La dispensa è sicuramente più "verbosa" rispetto ad altro materiale disponibile online: se avete bisogno di molti discorsi per poter comprendere alcuni concetti questa dispensa fa al caso vostro. Sentitevi liberi di stamparla e utilizzarla.

Se questi appunti sono stati utili e vuoi ringraziarmi in qualche modo: https://www.paypal.com/paypalme/GabrieleFrassi

Seguono le fonti utilizzate per la realizzazione della dispensa:

[1] M. Avvenuti, *Lezioni teoriche del corso di Sistemi Operativi,* Settembre 2021 - Dicembre 2021.

[2] P. Ancilotti, M. Boari, A. Ciampolini e G. Lipari, Sistemi operativi, seconda edizione (incluse diapositive mostrate dal docente durante le lezioni), McGraw-Hill, 2008.

[3] «Differenza tra RISC e CISC,» [Online]. Available: https://it.strephonsays.com/difference-between-risc-and-cisc.

2 Obiettivi e argomenti del corso

Il prof.Avvenuti ha svolto una lettura di quanto posto nella pagina del corso sul portale Valutami. La parola chiave del corso è sistema operativo multiprogrammato: numerose sono le questioni che dovranno essere affrontate: si pensi all'esecuzione concorrente di processi e alla programmazione concorrente (dobbiamo tenere conto di una serie di problematiche, in particolare l'accesso concorrente a risorse da parte di più processi). Il corso offre una visione olistica del calcolatore, attraverso un taglio più concettuale che pratico. Lo studente dovrà costruire un approccio critico e universale, avere una capacità di astrazione che permetta la risoluzione di problemi che vanno oltre gli argomenti del corso[1].

Programma (contenuti dell'insegnamento)

Concetti introduttivi. Principali funzioni di un sistema operativo. Cenni storici sull'evoluzione dei sistemi operativi. Richiami di architetture dei sistemi di elaborazione. Struttura dei sistemi operativi.

Gestione dei processi. Definizione di processo. Stati di un processo. Descrittore di un processo. Code di processi. Cambio di contesto. Creazione e terminazione dei processi. Interazione tra i processi. Richiami sul nucleo di un sistema a processi. Classificazione degli algoritmi di short-term scheduling e metriche per la loro valutazione (FCFS, SJF, SRTF, RR). Schedulazione di sistemi hard real-time (RM, EDF). Thread.

Sincronizzazione dei processi. Tipi di interazione tra processi. Problema della mutua esclusione. Problemi di sincronizzazione: Produttori-Consumatori, Lettori-Scrittori, 5 Filosofi. Semafori. Monitor. Primitive send e receive. Soluzione ai problemi di mutua esclusione, sincronizzazione e comunicazione tra processi. Blocco critico: condizioni, prevenzione statica e dinamica, rilevamento e recupero.

Gestione della memoria. Introduzione alla gestione della memoria. Memoria virtuale. Tecniche di gestione della memoria: segmentazione, paginazione, rimpiazzamento.

Gestione delle periferiche (I/O). Organizzazione logica del sottosistema di I/O. Gestore di un dispositivo. Device driver. Gestione e organizzazione dei dischi.

Il file system. Organizzazione del file system. La struttura logica del file system. Accesso al file system. Organizzazione e allocazione fisica.

Protezione e sicurezza. Modelli, politiche e meccanismi di protezione. Domini di protezione. Il modello matrice degli accessi. Realizzazione della matrice degli accessi. Sistema di sicurezza multilivello. Controllo degli accessi basato sui ruoli.

Esercitazioni in laboratorio su sistema operativo Linux. Comandi, shell, editor. Processi: gestione e interazione. Librerie per programmazione multi-thread. File system. Strumenti di sviluppo.

Bibliografia e materiale didattico

Libro di testo:

P. Ancilotti, M. Boari, A .Ciampolini, G.Lipari, *Sistemi Operativi*, 2ed, Mc Graw-Hill.

Modalità d'esame

L'esame è composto da una prova orale.
La prova orale può essere suddivisa in due parti: inizialmente, ai candidati viene richiesto di rispondere, generalmente in forma scritta, ad alcuni quesiti riguardanti le esercitazioni di laboratorio. L'esame prosegue con un colloquio, della durata media di 20 minuti, tra il candidato, il docente titolare e un suo collaboratore.

Il calendario degli appelli è stabilito dalla Scuola di Ingegneria.

E' necessario iscriversi sul sito: esami.unipi.it

[1] Il professore è storicamente pignolo e ci tiene a una discussione con terminologia appropriata e non ambigua. Durante la prova orale lo studente sarà valutato secondo le sue capacità di comprensione, presentazione e risoluzione di problematiche relative ai sistemi operativi. Il docente apprezza moltissimo uno che durante l'esame si accorge di aver sbagliato, ritorna sui suoi passi e individua la soluzione giusta (capacità di ragionamento, non spiattella le cose a memoria ma si rende conto dell'errore dopo un primo ragionamento e arriva alla soluzione giusta dopo ulteriori ragionamenti).

3 Unimap (registro, solo lezioni del prof.Avvenuti)

1. Mar 28/09/2021 08:30-10:30 (*2:0 h*) lezione: Introduzione al corso: obiettivi formativi, contenuti, prerequisiti, modalità d'esame. Principali funzioni di un sistema operativo: facilità di programmazione e portabilità dei programmi; gestione delle risorse; protezione. (MARCO AVVENUTI)
2. Mer 29/09/2021 16:00-19:00 (*3:0 h*) lezione: Cenni storici sull'evoluzione dei sistemi operativi. I primi sistemi di elaborazione. I primi sistemi batch. Sistemi batch multiprogrammati. Sistemi time-sharing. Sistemi in tempo reale. Personal computer. Sistemi distribuiti. (MARCO AVVENUTI)
3. Gio 30/09/2021 10:30-13:30 (*3:0 h*) lezione: Richiami di architetture dei sistemi di elaborazione: architettura di un calcolatore, rappresentazione dell'informazione, il processore, il bus, esempi di istruzioni in linguaggio macchina, ciclo della macchina (fasi di esecuzione). La gerarchia di memoria, memorie cache. Visione funzionale delle interfacce. (MARCO AVVENUTI)
4. Mar 05/10/2021 08:30-10:30 (*2:0 h*) lezione: Richiami di architetture dei sistemi di elaborazione: Chiamata di sottoprogramma. Controllo di programma e interruzione di programma. Gestione delle interruzioni e loro importanza nella costruzione di un sistema operativo. Direct Memory Access. Meccanismi di protezione, istruzioni int, cli, sti. Prinicipali modelli strutturali dei sistemi operativi. (MARCO AVVENUTI)
5. Mer 06/10/2021 16:00-19:00 (*3:0 h*) lezione: Tassonomia di Flynn: macchine parallele e sistemi distribuiti. Macchine SIMD, MIMD(Shared memory e Distributed memory). Metriche per la valutazione delle prestazioni. Topologie per reti di interconnessione. Gestione dei processi. Definizione di processo. Stati di un processo in un sistema multiprogrammato. Descrittore di un processo (PCB). Code di processi. (MARCO AVVENUTI)
6. Gio 07/10/2021 10:30-13:30 (*3:0 h*) lezione: Cambio di contesto. Creazione e terminazione dei processi. Processi concorrenti e interazione tra processi. Funzioni del nucleo di un sistema operativo. Scheduling: classificazione, obiettivi, metriche per la valutazione degli algoritmi. Algoritmo First-Come-First-Served (FCFS). Shortest-Job-First (SJF). (MARCO AVVENUTI)
7. Mar 12/10/2021 08:30-10:30 (*2:0 h*) lezione: Algoritmi di scheduling: Shortest-Remaining-Time-First (SRTF), Round-Robin (RR). Stima del CPU-burst. Simulazione e discussione comparativa. Scheduling su base prioritaria. Scheduling a code multiple. (MARCO AVVENUTI)
8. Mer 13/10/2021 16:00-19:00 (*3:0 h*) lezione: Scheduling a code multiple: multilevel feedback queue. Schedulazione di sistemi in tempo reale. Definizione di un sistema hard real-time, problema della schedulabilità, soluzioni nel caso di processi periodici. Algoritmo Rate Monotonic (RM). Fattore di utilizzazione. Condizioni necessarie e sufficienti per la schedulazione di sistemi periodici. Algoritmo Earliest Deadline First (EDF). Esempi di simulazione. (MARCO AVVENUTI)
9. Gio 14/10/2021 10:30-11:30 (*1:0 h*) lezione: Processi leggeri (thread). Sincronizzazione dei processi: Tipi di interazione tra i processi. Modelli ad ambiente globale e locale. (MARCO AVVENUTI)
10. Mar 19/10/2021 08:30-10:30 (*2:0 h*) lezione: Problema della mutua esclusione. Soluzioni per il problema della mutua esclusione. Semafori. Implementazione dei semafori. Atomicità delle primitive wait(), signal(). Semafori in ambiente multiprocessore. Soluzione al problema della mutua esclusione con semaforo mutex. (MARCO AVVENUTI)
11. Mer 20/10/2021 16:00-19:00 (*3:0 h*) lezione: Problema della comunicazione. Soluzione al problema della comunicazione con buffer condiviso, produttori-consumatori. Discussione di soluzioni alla mutua esclusione, produttori-consumatori (bounded buffer), lettori-scrittori. Ambiente locale: primitive send, receive. Modelli di programmazione a scambio di messaggi e aspetti implementativi. (MARCO AVVENUTI)
12. Gio 21/10/2021 10:30-11:30 (*1:0 h*) lezione: Problemi di sincronizzazione: Lettori-Scrittori, 5 Filosofi. Soluzione ai 5 filosofi basata su semafori. Esempi di deadlock. Monitor: definizione, variabili condition. (MARCO AVVENUTI)
13. Mar 26/10/2021 08:30-10:30 (*2:0 h*) lezione: Monitor: soluzione del problema dei 5 Filosofi. Realizzazione di monitor per mezzo di semafori: signal&wait, signal&continue. Monitor come allocatore di risorse. Blocco critico (deadlock). Grafo di assegnazione delle risorse. (MARCO AVVENUTI)
14. Mer 27/10/2021 16:00-19:00 (*3:0 h*) lezione: Condizioni per il blocco critico. Metodi per il trattamento del blocco critico: prevenzione statica, prevenzione dinamica, deadlock avoidance. Resource-Allocation Graph. Algoritmo del banchiere. (MARCO AVVENUTI)
15. Gio 28/10/2021 10:30-11:30 (*1:0 h*) lezione: Resource-Allocation Graph e Wait-For Graph. Algoritmi di deadlock detection. Tecniche di recovery. (MARCO AVVENUTI)
16. Gio 28/10/2021 11:30-13:30 (*2:0 h*) laboratorio: Ricerca di file (find, locate). Ricerca di testo nei file (grep). Archiviazione di file (tar). (MARCO AVVENUTI)
17. Mar 02/11/2021 08:30-10:30 (*2:0 h*) lezione: Algoritmi di deadlock detection. Tecniche di recovery. Gestione della memoria: memoria virtuale. (MARCO AVVENUTI)
18. Mer 03/11/2021 16:00-19:00 (*3:0 h*) lezione: Gestione della memoria: rilocazione statica e dinamica, spazio virtuale unico o segmentato, allocazione fisica contigua e non-contigua,

caricamento tutto insieme o a domanda. Classificazione delle tecniche di allocazione della memoria. (MARCO AVVENUTI)

19. Gio 04/11/2021 10:30-11:30 (*1:0 h*) lezione: Tecniche di gestione della memoria: partizioni fisse, variabili, multiple. Allocazione best-fit e first-fit. Problema della frammentazione. (MARCO AVVENUTI)
20. Mar 09/11/2021 08:30-10:30 (*2:0 h*) lezione: Memoria segmentata: traduzione degli indirizzi virtuali, protezione. Aspetti implementativi, descrittore di segmento, condivisione. Segmentazione a domanda. Segment-fault. (MARCO AVVENUTI)
21. Mer 10/11/2021 16:00-19:00 (*3:0 h*) lezione: Gestione della memoria: paginazione, paginazione a domanda. Gestione del page-fault. Segmentazione con paginazione (MARCO AVVENUTI)
22. Gio 11/11/2021 10:30-11:30 (*1:0 h*) lezione: Algoritmi di rimpiazzamento. (MARCO AVVENUTI)
23. Mar 16/11/2021 08:30-10:30 (*2:0 h*) lezione: Segmentazione con paginazione.Gestione degli spazi virtuali. I processi Unix. Stati dei processi. Immagine di un processo Unix: process strutture, user structure, text table, relazioni fra strutture dati. Gestione della memoria virtuale in Unix. (MARCO AVVENUTI)
24. Mer 17/11/2021 16:00-19:00 (*3:0 h*) lezione: Gestione delle periferiche (I/O): concetti generali, organizzazione logica del sottosistema di I/O, funzioni del livello device-independent. funzioni del livello device-dependent. (MARCO AVVENUTI)
25. Gio 18/11/2021 10:30-11:30 (*1:0 h*) lezione: Gestione di un dispositivo: processi esterni, descrittore di un dispositivo, controllo di programma, interruzione. Struttura di un device driver. (MARCO AVVENUTI)
26. Mar 23/11/2021 08:30-10:30 (*2:0 h*) lezione: Flusso di controllo durante un trasferimento. Gestione del temporizzatore. (MARCO AVVENUTI)
27. Mer 24/11/2021 16:00-19:00 (*3:0 h*) lezione: Organizzazione fisica e logica dei dischi. Criteri di scheduling delle richieste di accesso al disco. Criteri di scheduling delle richieste di accesso al disco: Algoritmi FCFS, SSTF, SCAN. (MARCO AVVENUTI)
28. Gio 25/11/2021 10:30-11:30 (*1:0 h*) lezione: Organizzazione del file system. La struttura logica del file system. (MARCO AVVENUTI)
29. Mar 30/11/2021 08:30-10:30 (*2:0 h*) lezione: Accesso al file system. Strutture dati e operazioni per l'accesso ai file. (MARCO AVVENUTI)
30. Mer 01/12/2021 16:00-19:00 (*3:0 h*) lezione: Organizzazione fisica del file system. Tecniche di allocazione dei file: contigua, a lista concatenata, a indice. File Allocation Table. Valutazioni sul costo di accesso al blocco e sulla scalabilità. Unix: organizzazione del file system, strutture dati del kernel per l'accesso ai file. (MARCO AVVENUTI)
31. Gio 02/12/2021 10:30-11:30 (*1:0 h*) lezione: System calls Unix per l'accesso ai file: open, close, read, write. Protezione e sicurezza. Modelli, politiche e meccanismi di protezione. (MARCO AVVENUTI)
32. Mar 07/12/2021 08:30-10:30 (*2:0 h*) lezione: Il modello matrice degli accessi. Rappresentazione dello stato di protezione. Rispetto dei vincoli di accesso. Graham& Denning: Modifica dello stato di protezione. Realizzazione della matrice degli accessi. Access Control List. Riferimenti a Unix. (MARCO AVVENUTI)
33. Gio 09/12/2021 10:30-11:30 (*1:0 h*) lezione: Realizzazione della matrice degli accessi: Capability List. Riferimenti a Unix (MARCO AVVENUTI)
34. Mar 14/12/2021 08:30-10:30 (*2:0 h*) lezione: Sistemi di sicurezza multilivello (Bell-La Padula). Esempi di controllo flusso informativo, Trojan Horse, sistemi BIBA. Esercizi di riepilogo. (MARCO AVVENUTI)
35. Mer 15/12/2021 16:00-17:00 (*1:0 h*) lezione: Domande ed esercizi di riepilogo. (MARCO AVVENUTI)
36. Gio 16/12/2021 17:00-19:00 (*2:0 h*) non tenuta: Ore cedute a Reti Informatiche (MARCO AVVENUTI)

4 Concetti introduttivi

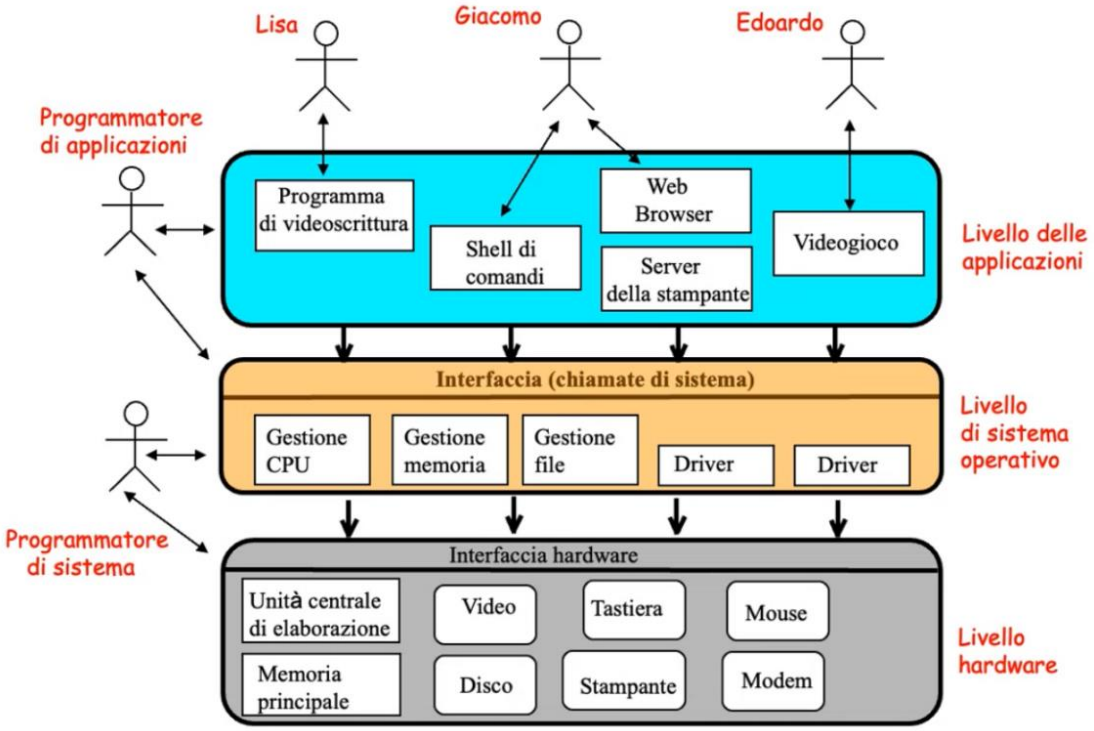
4.1 Definizione di sistema operativo

Un sistema operativo è un componente software di un sistema di elaborazione il cui compito principale è quello di controllare l'esecuzione dei programmi applicativi e di agire come intermediario tra questi e la macchina fisica (hardware), con lo scopo di:

- facilitarne l'uso;
- garantire che tale uso sia effettuato in maniera efficace ed efficiente.

4.2 Rappresentazione del sistema operativo attraverso uno schema

Per avere le idee chiare rappresentiamo il sistema operativo attraverso uno schema a livelli. Analizziamo, in particolare, le relazioni tra i vari livelli. Si tenga conto che il sistema operativo a cui facciamo riferimento è un sistema *general purpose*, cioè pensato per un uso generale.



4.2.1 Livello Hardware

A livello basso abbiamo l'hardware, nel nostro caso una versione semplificata del calcolatore. Per essere tale deve presentare almeno i seguenti elementi…

- **Processore** (*Unità centrale di elaborazione*, o CPU).
  Circuito estremamente complesso che esegue un insieme di istruzioni definito in un set (nella sua potenza è un mero esecutore di istruzioni). Nella realizzazione di un processore:
  1. definisco un set di istruzioni;
  2. implemento il circuito in modo tale che le istruzioni vengano eseguite nel modo più efficiente possibile.

  Gli approcci architetturali nella scelta delle istruzioni sono:
  - **RISC** (*Reduced Instruction Set Computer*), architettura semplice, istruzioni semplici e ridotte nell'ottica di rendere la macchina efficiente;
  - **CISC** (*Complex Instruction Set Computer*), già vista con Lettieri, singole istruzioni complesse e in grado di fare più cose.

  La cosa è approfondita più avanti. Ricordarsi che le istruzioni vengono eseguite dal processore seguendo il cosiddetto ***ciclo della macchina***:
  1. fase di fetch (caricamento dell'istruzione dalla memoria ai registri della CPU);
  2. decodifica dell'istruzione, preparazione dei componenti necessari per l'esecuzione dell'istruzione:
  3. esecuzione dell'istruzione.

  Alcune istruzioni leggono, altre scrivono, altre ancora interagiscono con le periferiche. Il numero di fasi preciso dipende dall'architettura del processore, e sono ripetute ciclicamente dal processore in modo

tale che le istruzioni appartenenti ai programmi vengano eseguite uniformemente. Il ciclo, la velocità con cui avviene la ripetizione del ciclo della macchina, dipende dal clock, l'oscillatore che nei processori moderni è nell'ordine dei GHz.

- **La memoria principale** (volatile)
  Memoria volatile, ma veloce: per avere velocità nel tempo di accesso, sia in lettura che in scrittura, devo rinunciare alla persistenza di un supporto fisico. La memoria principale serve per mantenere i dati e i programmi in esecuzione: la cosa riflette il modello di Von Neumann, che prevede un esecutore di istruzioni, una memoria dove viene caricato il programma insieme ai relativi dati, interfacce per memorizzare dati in uscita e/o in ingresso. Il modello entra in crisi negli anni 60 con l'avvento della multiprogrammazione: questo perché tutti i programmi devono essere in memoria per essere eseguiti, e la memoria (all'epoca) è molto piccola.

- **Bus**
  Fili di interconnessione, necessari per far comunicare i moduli di un calcolatore (incluse CPU e memoria) e permettere lo scambio di informazioni. In particolare, la CPU potrà, quando ne ha bisogno, caricare un'istruzione da eseguire in memoria.

- **Periferiche esterne**.
  Il calcolatore richiede la presenza di alcune periferiche di I/O, altrimenti il sistema è autistico (cit.), nel senso che non comunica con l'esterno. Una delle periferiche più importanti è l'interfaccia di rete, che nasce verso la fine degli anni 60 e che vedremo a Reti informatiche. Vedremo verso la fine del corso come sia fondamentale la presenza di una memoria secondaria permanente, ai fini della multiprogrammazione.

Le componenti hardware sono dette **RISORSE FISICHE**

4.2.2 Interfaccia hardware

L'accesso alle risorse fisiche è possibile solo attraverso un'interfaccia hardware, che nel nostro caso è rappresentata dalle istruzioni Assembler del processore. La programmazione è a livello molto basso, e soprattutto *processor-specific*: la cosa non ci va bene, manca la portabilità ma soprattutto diventa più facile compiere errori (parliamo di istruzioni molto specifiche).

Quando programmiamo a basso livello, ricordiamo, si ragiona in termini di somme (ripensiamo a quanto visto a Reti logiche). Per le motivazioni dette dobbiamo passare a un livello di astrazione superiore: il sistema operativo.

4.2.3 Livello di sistema operativo

Il livello del sistema operativo consiste in un insieme di componenti software che hanno il compito di gestire le risorse fisiche della macchina offrendo un'interfaccia standard, più semplice da usare, al programmatore. Le funzioni presenti in questo livello vengono invocate dai programmi applicativi per intervenire sulle componenti hardware del sistema in modo controllato ed efficiente (come abbiamo visto a Calcolatori Elettronici).

4.2.4 Livello delle applicazioni

In realtà il programmatore lavora in un livello superiore: il livello delle applicazioni. Il livello delle applicazioni contiene le applicazioni usate direttamente dagli utenti del sistema. Per quanto già detto si capisce che chi si trova in questo livello non interagisce mai direttamente con l'hardware, ma sempre attraverso l'interfaccia del sistema operativo. La cosa è decisamente più conveniente per noi:

Programmazione a basso livello -> Si devono conoscere più dettagli, maggiore rischio di errore

Il sistema operativo non è altro che un insieme di programmi **che nascondono la macchina fisica**. È necessario che le applicazioni siano indipendenti il più possibile dal livello hardware, in generale da tutti quei dettagli specifici che variano frequentemente nel tempo.

- Un programma applicativo opera sulle risorse fisiche per il tramite delle funzioni del sistema operativo.
- Il sistema operativo fornisce ai programmi applicativi un'interfaccia costituita da un insieme di funzioni che mascherano la struttura della macchina fisica, mostrando a tutti gli effetti una macchina virtuale.

Nella macchina virtuale sono definite nuove funzioni/istruzioni diverse da quelle hardware. Queste funzioni:

- hanno valenza generale (sono valide in qualunque architettura, sono usate dal programmatore rendendo quindi i programmi portabili e meno soggetti ad errore);
- sono implementabili attraverso le funzioni dell'interfaccia hardware specifica.

4.2.5 [Extra da strephonsays] Differenze tra RISC e CISC

Il professore ha recuperato RISC e CISC, ma non ha approfondito fin troppo, forse dando per scontato un approfondimento da parte di Lettieri nel corso di Calcolatori elettronici. Approfondiamo la cosa!

<table>
<tr><th>RISC</th><th>CISC</th></tr>
<tr><td>RISC sta per Computer Set di istruzioni ridotto. Si riducono i tempi di esecuzione semplificando il set di istruzioni. Il set di istruzioni è piccolo e altamente ottimizzato. I processori basati su RISC sono comunemente usati per dispositivi portatili come telefoni cellulari e tablet in quanto sono più efficienti.</td><td>CISC sta per Computer Set di istruzioni complesso. L'obiettivo principale è la riduzione del numero di istruzioni in un programma. Viene principalmente utilizzato per laptop e computer desktop.
<u>Poiché le istruzioni sono complesse, richiede un numero multiplo di cicli di clock per eseguire una singola istruzione. Inoltre, la decodifica delle istruzioni è più complessa</u>.</td></tr>
<tr><td>- Set piccolo e altamente ottimizzato
- Orientato alla macchina
- Numero maggiore di registri
- Istruzioni semplici, richiedono un ciclo di clock per l'esecuzione (o comunque un numero molto basso)
- Le istruzioni hanno formati semplici e fissi con poche modalità di indirizzamento.
- Programmi di lunghezza maggiore.
- RAM richiesta maggiore, dobbiamo memorizzare un numero maggiore di istruzioni.</td><td>- Set ampio, specializzato e complesso
- Orientato al programmatore
- Numero inferiore di registri
- Istruzioni complesse, richiesto un numero maggiore di cicli di clock per eseguire una sola istruzione. Decodifica delle istruz. complessa.
- Le istruzioni hanno formati variabili con diverse modalità di indirizzamento complesse.
- Programmi più corte (istruz. fanno più cose)
- RAM richiesta minore, dobbiamo memorizzare un numero inferiore di istruzioni.</td></tr>
<tr><td colspan="2">Conclusione.
La differenza tra RISC e CISC è che il RISC contiene un set di istruzioni piccolo e altamente ottimizzato mentre il CISC contiene una serie di istruzioni specializzate e complesse. In altre parole, RISC ha un set di istruzioni più piccolo e semplice mentre CISC ha un set di istruzioni ampio e complesso.</td></tr>
</table>

4.3 Funzioni di un sistema operativo

4.3.1 Premessa alle funzioni: distinzione tra meccanismo e politica

Attenzione:

- Le politiche sono criteri di scelta applicati in certi ambiti (per esempio assegnazione delle risorse ai processi).
- Il meccanismo è un qualcosa di cablato in circuiti hardware o realizzato in software che permette l'attuazione delle politiche.

Cosa intendiamo con kernel? Il kernel, ciò che abbiamo visto a Calcolatori elettronici, consiste nell'implementazione di tutti quei servizi necessari per implementare politiche. Il kernel permette l'astrazione della CPU virtuale.

4.3.2 Facilitare lo sviluppo e la portabilità dei programmi applicativi

Il sistema operativo permette una programmazione ad alto livello fornendo delle primitive: l'elevata astrazione semplifica la vita al programmatore, che può programmare pensando a una macchina astratta indipendente dal livello hardware (questo è un bene, non devo programmare pensando alle caratteristiche delle singole componenti hardware, che in commercio sono tante e ciascuna con le proprie caratteristiche).

- L'interfaccia fornita al programmatore è l'***Application Programming Interface*** (API), cioè un set di funzioni (o *primitive* nel caso di un sistema operativo, o *metodo* nel caso di API relative a Java) che il programmatore può invocare nei suoi programmi. Di esse il programmatore conosce l'intestazione, quindi nome, parametri di ingresso ed eventuale parametro di uscita.
- Conoscere l'API significa comprendere non solo l'elenco, ma anche la semantica (non nell'aspetto implementativo, da cui siamo sollevati). Il programmatore deve essere consapevole di cosa succede quando va a chiamare una funzione, quale sia il suo scopo preciso.
- Le interfacce stesse sono portabili. I programmi non devono essere riscritti se cambio ciò che sta sotto.

4.3.3 Realizzare politiche di gestione delle risorse del sistema operativo

Abbiamo visto che la macchina hardware è già di per sé la collezione di risorse fisiche, ma è anche, attraverso alcune delle sue periferiche, una collezione di risorse logiche (files, variabili, strutture dati in memoria…). La presenza di sistemi multiprogrammati richiede una gestione adeguata di queste risorse (pensare al fatto che i processi competono per usufruire delle risorse fisiche).

- **Gestire le risorse**: allocare risorse di vario tipo (logiche e fisiche) nei vari programmi che sta eseguendo il sistema. Per esempio. un programma ha bisogno della CPU e della memoria principale (architettura di Von Neumann), andando avanti potrebbe aver bisogno delle periferiche.
- Deve essere garantita l'efficienza, cioè usare al massimo il tempo macchina. Se io ho un certo numero di CPU devo dare programmi da eseguire a tutte le CPU, in modo tale da avere il 100% di utilizzo. Nel caso di un programma interattivo sono necessari tempi di risposta accettabili all'occhio umano.
- Per quanto riguarda il gestire alcune periferiche dobbiamo realizzare un qualcosa di trasparente: si pensi a quando colleghiamo un hard disk, dobbiamo fare questo senza lasciare pensieri al programmatore, che continua a comportarsi come se nulla fosse.

4.3.4 Fornire meccanismi di protezione, garantire la sicurezza del sistema e la tolleranza ai guasti

Questo punto è estremamente importante in un sistema *multiprogrammato*: ogni processo deve essere protetto da possibili interferenze degli altri utenti.

- In presenza di più processi che usano le risorse di sistema dobbiamo garantire che il processo acceda solo nelle aree dove può effettivamente accedere (si pensi alla memoria). Dobbiamo anche stabilire chi controlla, e come controlla, *run time*.

Dobbiamo garantire privacy (riservatezza, i dati in memoria che non riguardano un processo non devono essere letti da altri processi) e protezione (i dati in memoria che riguardano un processo non devono essere manipolati da altri processi).

4.4 Cenni storici e tipologie di sistemi operativi

4.4.1 Premessa: articolo di Wired sull'Elea 9003

Quanto segue è un articolo di wired.it sulla storia di Mario Tchou e del suo Elea 9003, grande successo della società guidata da Adriano Olivetti. L'Elea 9003 è il primo calcolatore interamente a transistor: i vecchi calcolatori, a valvole, erano termodinamicamente parlando un disastro. L'articolo riepiloga a grandi linee le cose dette a lezione (il prof. ha fatto vedere un video dove sono sostanzialmente dette le stesse cose dell'articolo).

Storia dell'ingegner Mario Tchou dell'Olivetti, che riuscì a battere sul tempo l'Ibm

L'ingegnere di origini cinesi progettò l'Elea 9003, il primo computer italiano totalmente a transistor, arrivato con alcuni mesi di anticipo rispetto a quello dell'Ibm

Nato a Roma il 26 giugno 1924, Mario Tchou era figlio di un diplomatico cinese che lavorava all'ambasciata della Cina imperiale presso il Vaticano. Tchou ebbe **un'istruzione italiana** prima di **specializzarsi negli Stati Uniti**. Diplomatosi al liceo classico Torquato Tasso di Roma, iniziò gli studi universitari alla Sapienza per poi continuarli oltre oceano, dove si laureò in ingegneria elettronica a Washington nel 1947. Dopo essersi spostato a New York, Tchou conseguì un master al Polytechnic Institute of Brooklyn con una tesi sulla diffrazione ultrasonica. Cominciò inoltre a insegnare, prima al Manhattan College e successivamente alla Columbia University.

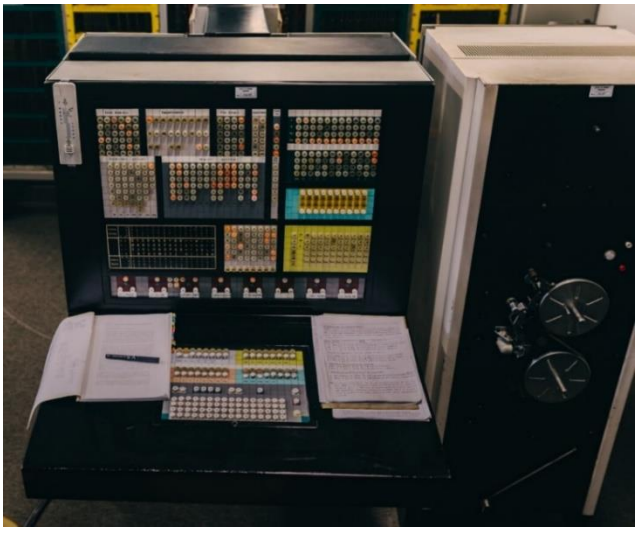
L'incontro con Olivetti

Fu proprio a New York che **nel 1954 Tchou incontrò Adriano Olivetti**. Tchou era stato segnalato all'imprenditore di Ivrea da **Enrico Fermi**, che già da qualche anno stava cercando di convincere l'Olivetti a investire sull'**elettronica**. Tchou poteva quindi essere la persona giusta per ricoprire il ruolo di direttore per il nuovo Laboratorio di ricerche elettroniche della Olivetti. L'ingegnere si presentò per sostenere il colloquio presso la sede americana della Olivetti, ma le domande che gli fece l'industriale non riguardarono la tecnologia. Tchou raccontò che durante il colloquio, Olivetti sembrava più interessato a **conoscere aspetti sociali e relazionali** piuttosto che quelli tecnici. Ciò lo colpì positivamente. Così **Tchou decise di accettare l'offerta di lavoro e di far rientro in Italia**, anche per ragioni familiari.

Il laboratorio fu aperto a **Barbaricina**, un sobborgo di Pisa, poiché l'Olivetti in quegli anni aveva instaurato una collaborazione con l'università toscana per la costruzione di un nuovo calcolatore scientifico denominato **Calcolatrice elettronica pisana** (Cep). Il laboratorio però aveva un suo progetto specifico, ovvero la progettazione della **prima calcolatrice commerciale**, l'**Elea (Elaboratore elettronico aritmetico)**. L'acronimo si ispirava all'antica città della Magna Grecia sede di scuole di filosofia, scienza e matematica.

Tchou si occupò personalmente della selezione del personale, privilegiando candidati sotto i 30 anni e in possesso di requisiti come entusiasmo, spirito innovativo, immaginazione e capacità di lavorare in gruppo. Tra i primi assunti, figuravano pionieri dell'informatica come Franco Filippazzi e altri tecnici e scienziati che furono poi ricordati come i **"ragazzi di Barbaricina"**.

Il primo prototipo dell'Elea



Nel 1957 il gruppo realizzò il primo prototipo del nuovo elaboratore, l'Elea 9001 (o "Macchina Zero"), che era a valvole e quindi di grandi dimensioni. L'anno successivo seguì l'Elea 9002 (o "Macchina 1V"), più veloce della versione precedente e considerata il prototipo di una macchina commerciale. Tchou, tuttavia, ne sospese il lancio sul mercato poiché intuì che grazie all'emergente tecnologia dei transistor sarebbe stato possibile costruire una macchina senza valvole, più veloce e meno costosa.

Il laboratorio da Barbaricina fu poi spostato a Borgolombardo, in provincia di Milano, dove dopo poco tempo il team guidato da Tchou riuscì a portare a compimento il progetto. A novembre del 1959 Adriano Olivetti presentò al presidente della Repubblica Giovanni Gronchi l'**Elea 9003, il primo computer italiano totalmente a transistor**. Chiamato anche "Macchina 1T", era un **prodotto d'avanguardia** per l'epoca, arrivato alcuni mesi in anticipo rispetto al primo computer a transistor dell'Ibm, allora leader nell'elettronica.

Ad occuparsi del design era stato **Ettore Sottsass**, che aveva messo al centro l'uomo e non la macchina, realizzando una console dalla forma ergonomica e coi comandi facilmente alla portata dell'operatore. Grazie all'Elea 9003, nel 1959 Sottsass si aggiudicò il Compasso d'oro. Nello stesso anno l'Elea 9003 venne presentata alla Fiera campionaria di Milano e l'anno successivo venne consegnata al primo cliente, la Marzotto di Guadagno. In totale, furono **venduti circa 40 esemplari** a grandi aziende, banche e enti pubblici.

Lo sviluppo di Elea non si fermò lì. Nel 1960 fu realizzata l'**Elea 6001**, un calcolatore di minor costo e dimensioni orientato ad applicazioni di carattere scientifico e rivolto quindi per un'utenza media come istituti universitari, enti pubblici e media industria. Ebbe un forte successo, vendendo circa 100 esemplari.

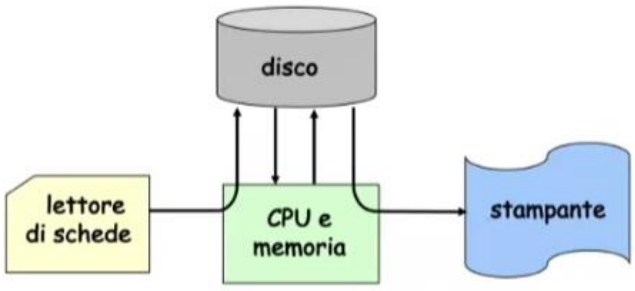
4.4.2 Sistemi batch monoprogrammati

Nei primi sistemi batch viene introdotto, novità, il sistema operativo (un embrione). Lo scopo principale del sistema operativo è ridurre l'intervento manuale dell'utente.

- Si introduce una memoria di massa, la sua presenza permette di leggere le schede e memorizzarle. La memoria di massa permette anche di gestire l'output, inizialmente salvato in memoria e successivamente restituito attraverso le periferiche.
- L'utente-programmatore forniva il proprio programma sorgente a un operatore, l'unico abilitato a operare direttamente sulla macchina. Le memorie centrali diventano più grandi, quindi nasce il primo embrione di sistema operativo: esso consiste nel Job Control Language (JCL), un linguaggio che permette di scrivere degli script di comandi. Lo scopo del JCL è indicare alla macchina, tra tante cose, quale programma avviare, l'input del programma, cosa fare dopo averlo terminato, quali risorse allocare prima dell'esecuzione…[2].
- Il programmatore produce il suo pacco di schede: tra queste si hanno le schede di controllo contenenti i comandi del JCL. Non si forniscono schede relative al job di un singolo utente, ma pacchi di schede relativi a job di più utenti. L'insieme forma il **batch**.
- Viene definito un programma di sistema detto **monitor**, il quale risiedeva in una porzione di memoria.
    - Lo scopo del programma era leggere le schede e, per ogni scheda di controllo, interpretarne il contenuto (quindi è interprete del JCL).
    - A seguito del caricamento dei programmi in memoria il monitor trasferisce il controllo della CPU al programma.
    - Al termine del programma il monitor riottiene il controllo della CPU, che valuta la successiva scheda di controllo (e lo esegue cedendo nuovamente il controllo della CPU).
- L'unico intervento da parte dell'operatore si ha al termine dell'esecuzione del batch di job.
- **Quindi, perché embrione?** Abbiamo:
    - linguaggio JCL, che richiede un interprete (visto che si ha un linguaggio di scripting) detto monitor (pacchetto di schede residente in memoria);
    - BIOS (Basic Input Output System), routine per caricare le schede sui nastri, trasferirle col DMA (va inizializzato, quale indirizzo e quanti byte).

4.4.3 Sistemi di spooling (*Simultaneous peripheral operation on-line*), più flussi col DMA

I primi sistemi continuano ad avere efficienza bassa, principalmente per la differenza di velocità tra CPU (veloce) e dispositivi periferici (lenti). I sistemi di spooling presentano più flussi in contemporanea, grazie all'introduzione del *Direct Memory Access* (DMA):



- trasferimento da dispositivo di input alla memoria (!!!! Grazie al DMA !!!!);
- trasferimento di dati e programma dal disco alla memoria per eseguirli;
- trasferimento dalla memoria al disco dei risultati dell'esecuzione;
- trasferimento dal disco a un qualunque dispositivo di output (!!!! Grazie al DMA !!!!).

L'uso del DMA è rilevante perché permette di caricare il batch successivo durante l'esecuzione del precedente (non devo più aspettare la fine dell'esecuzione di un batch per caricare il batch successivo). L'efficienza aumenta in modo evidente, e siamo pronti per fare il salto ai sistemi multiprogrammati. Il fatto che si carichi un batch tutto insieme permette anche di elaborare le prime forme di scheduling (cioè l'ordine con cui eseguire i job che costituiscono il batch).

[2] Originally, mainframe systems were oriented toward batch processing. Many batch jobs require setup, with specific requirements for main storage, and dedicated devices such as magnetic tapes, private disk volumes, and printers set up with special forms. JCL was developed as a means of ensuring that all required resources are available before a job is scheduled to run.

4.4.4 Sistemi multiprogrammati

Il momento di svolta si ha con l'introduzione delle interruzioni, che rende possibile la multiprogrammazione.

Il quadrato in foto rappresenta la memoria centrale che contiene il sistema operativo e i vari programmi applicativi. Nel pieno rispetto dell'architettura di Von Neumann andiamo a mettere più programmi nella stessa memoria centrale: ovviamente la memoria deve avere la capacità necessaria.

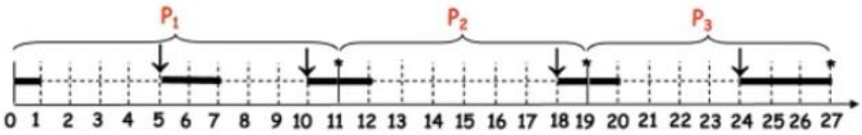
Supponiamo di avere tre programmi caricati:

- nel *Program counter* il sistema ha posto l'indirizzo della prima istruzione del primo programma, dunque iniziamo eseguendo il primo dei tre programmi;
- il primo programma, a un certo punto, esegue un'istruzione di I/O (per tutta la durata dell'istruzione la CPU non ha nulla da fare relativamente a questo programma);
- il sistema operativo, a questo punto, altera il *Program counter* e pone l'indirizzo della prima istruzione del secondo programma (si sospende l'esecuzione del processo e si passa a lavorare sul secondo programma applicativo).

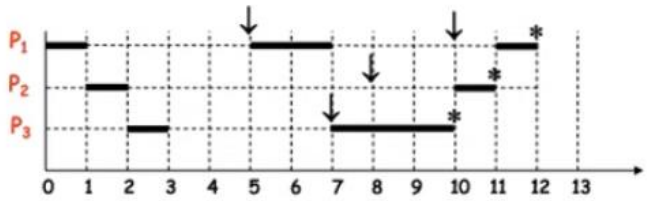
l'idea sembra banale, ma in realtà è estremamente rivoluzionaria. In tutto questo dovremo considerare un'ulteriore componente: quella che si dedica allo *scheduling* (che gestisce l'esecuzione concorrente dei programmi applicativi).

4.4.4.1 Esecuzione sequenziale e multitasking a confronto

Vediamo il vantaggio principale della multiprogrammazione:



esecuzione sequenziale



esecuzione in multi-tasking

- **Esecuzione sequenziale**
  Nella prima figura i programmi P1, P2 e P3 vengono eseguiti in modo sequenziale: quando la CPU viene assegnata a un programma questo la tiene fino a quando non esegue l'ultima istruzione. Si hanno dei trasferimenti che durano certe unità di tempo, ma non si cede il controllo a nessuno (non si cede il controllo neanche in presenza di operazioni di I/O). Complessivamente il programma viene eseguito in 27 unità di tempo: unità in cui la CPU attiva / unità totale. Otteniamo un'efficienza al 37%

- **Esecuzione multitasking**
  Nell'esecuzione multitasking si cede il controllo ad altri programmi in presenza di operazioni di IN/OUT, la CPU non lavora solo quando non ci sono programmi effettivamente eseguibili (cioè quando tutti e i tre programmi hanno lanciato operazioni di I/O). Complessivamente il programma viene eseguito in 12 unità di tempo: se facciamo i calcoli otteniamo un'efficienza all'83%.

Domanda: è possibile migliorare ancora l'efficienza? Se il sistema rimane così no, dovrei avere più programmi da eseguire. Ma posso mettere altri programmi? La memoria deve avere la capacità necessaria per ospitare ulteriori programmi (ricordiamo che le RAM sono COSTOSE, fare caso al prezzo di due PC dello stesso modello ma dimensione della RAM diversa). Maggiore è il livello di multiprogrammazione (cioè maggiore è il numero di processi), maggiore è il livello di efficienza (che può arrivare tranquillamente al 100%).

4.4.4.2 Programmi I/O-bound e CPU-bound

Per proseguire i nostri ragionamenti conviene distinguere due tipologie di programmi (ora li chiamiamo programmi, poi saranno processi):

- **Programmi I/O-bound**, passa più tempo a trasferire dati che a eseguire;
- **Programmi CPU-bound**, l'opposto rispetto ai programmi precedenti.

Chiaramente il programma perfetto (CPU-bound) è quello che esegue sempre, ma è un qualcosa di ideale e non ottenibile. Nell'esempio precedente i programmi sono I/O-bound.

Cosa ci interessa fare nella pratica. Nella realtà dobbiamo ottenere un misto tra programmi I/O-bound e CPU-bound: questa cosa permette di avere un'efficienza vicina al 100% (ho un buon numero di programmi CPU-bound che tengono impegnata la CPU) e di sfruttare al massimo le periferiche a disposizione del calcolatore.

4.4.4.3 **Sistemi con diritto di revoca (preemption), introduzione delle interruzioni**

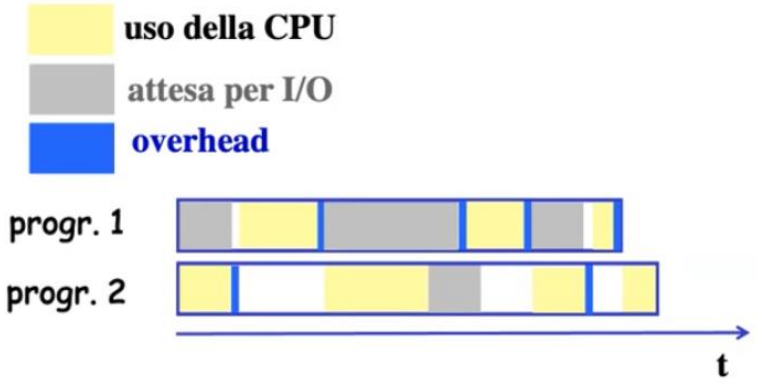
Il sistema che abbiamo appena visto è senza diritto di revoca: la CPU non può revocare il controllo al programma dopo averglielo assegnato, deve aspettare PER FORZA il termine dell'esecuzione del programma (oppure attende che il programma si sospenda da solo con un'istruzione di IN/OUT).

- **Abbiamo bisogno di un ulteriore meccanismo hardware: le interruzioni**!
  Una periferica può lanciare un segnale di interruzione alla CPU, chiedendo alla stessa di fare altro (ricordare come abbiamo introdotto, non abbiamo il processore che controlla di continuo il registro di una periferica ma la periferica che avverte la CPU che il valore del registro è cambiato).
- Le interruzioni, soprattutto quelle esterne, vengono gestite attraverso un controllore delle interruzioni. Introduciamo anche le interruzioni di sistema, lanciabili da un processo utilizzando l'istruzione INT (con cui gestiamo le interruzioni di sistema).

- ***preemption (prelazione, o pre-rilascio)***
  Attraverso il lancio di un'interruzione possiamo avviare una routine che fa la *preemption*: il processo viene privato del controllo della CPU in situazioni diverse da quelle già introdotte. La flessibilità aumenta!

- **Perchè la *preemption* è così importante?**
  Supponiamo di avere un mix di programmi CPU-bound, per qualche ragione nel creare il mix eseguiamo prima i programmi CPU-bound. Se questi programmi durano tre giorni questi mantengono il controllo della CPU per tre giorni di fila, lasciando i programmi I/O-bound alla fine. **Non è mai un bene che un programma CPU-bound mantenga il controllo per così tanto tempo: vogliamo proporzione tra tempo di risposta e complessità come percepita dall'utente**.

4.4.4.4 La questione dell'overhead



Per poter ottenere tutto quello che abbiamo detto (in particolare il salvataggio del processo con l'aggiornamento delle strutture dati necessarie) è necessario considerare l'*overhead*, **cioè il tempo che la macchina spende per operazioni dovute alla multiprogrammazione** (operazioni che non sono di per sé utili per il funzionamento dei singoli programmi, si parla di routine di sistema non associate a un particolare processo).

L'overhead deve essere accettabile, altrimenti arrivo al paradosso di avere una macchina perfetta che passa tutto il suo tempo ad allocare e deallocare risorse (con i programmi che stanno fermi perché non hanno la CPU a loro disposizione).

4.4.5 Sistemi time-sharing

Nei sistemi *time-sharing* l'idea è partizionare il tempo della CPU in intervalli di tempo (*quanti*). Il sistema, che ha diritto di revoca, assegna la CPU a un programma e la revoca al termine dell'intervallo di tempo. A quel punto la CPU viene assegnata ad un altro programma.

Osservazione. Attenzione a non confondere i sistemi time-sharing con sistemi in tempo reale. In entrambe le tipologie di sistema si pone una deadline, ma nel caso dei sistemi time-sharing superare l'intervallo di tempo *quanti* non è errore: semplicemente il processo sarà sospeso e ripreso più avanti.

4.4.6 Sistemi operativi distribuiti

I sistemi distribuiti sono pensati per essere eseguiti su macchine fisicamente distribuite. Vedremo meglio questa cosa nelle prossime lezioni, con la tassonomia di Flynn. I dischi, ad esempio, vengono posti in modo tale da dare all'utente la parvenza di un unico file system. Si pensi ai datacenter, ai server dove sono ospitati i vari siti web.

4.4.7 Sistemi in tempo reale

I sistemi in tempo reale sono sistemi *special-purpose*, pensati per eseguire particolari applicazioni.

Le periferiche sono pensate per acquisire dati attraverso sensori, da un ambiente operativo esterno. Sulla base dei dati ottenuti si decide come influire sull'ambiente esterno, attraverso degli attuatori. Si dice "in tempo reale" poiché legati a un tempo:

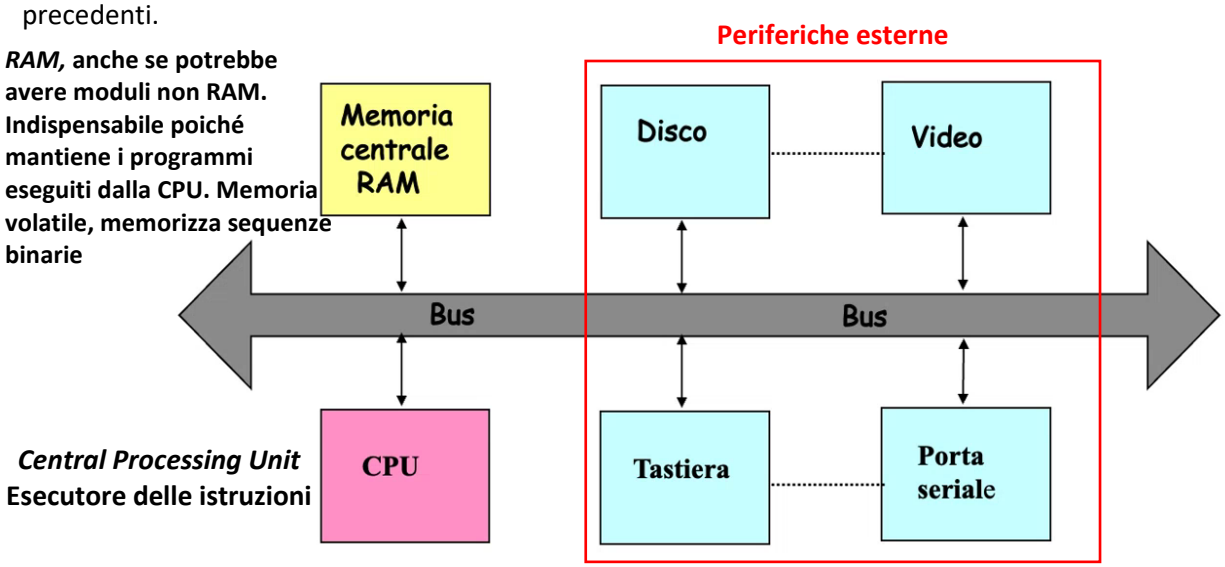
- abbiamo un ambiente operativo che si evolve nel tempo, secondo una legge temporale;
- sono richiesti dei vincoli temporali stringenti, cioè si impone una deadline (si produce un output entro un tempo accettabile).

È necessario uno scheduling che garantisca l'esecuzione di tutti i task entro la loro deadline: cambia il requisito, il superamento della deadline è errore e non si può fare time-sharing. Lo scheduling è su base prioritaria: assegno una priorità e sulla base di questa do la CPU ai processi. Si distinguono due tipologie di sistemi:

- ***hard real time***, sistemi dove dato un insieme di task periodici devo garantire che la deadline non venga violata (nessun overflow);
- ***soft real time***, sistemi dove la deadline può essere violata, ma si ha una degradazione della qualità del servizio (Esempio: sistemi multimediali).

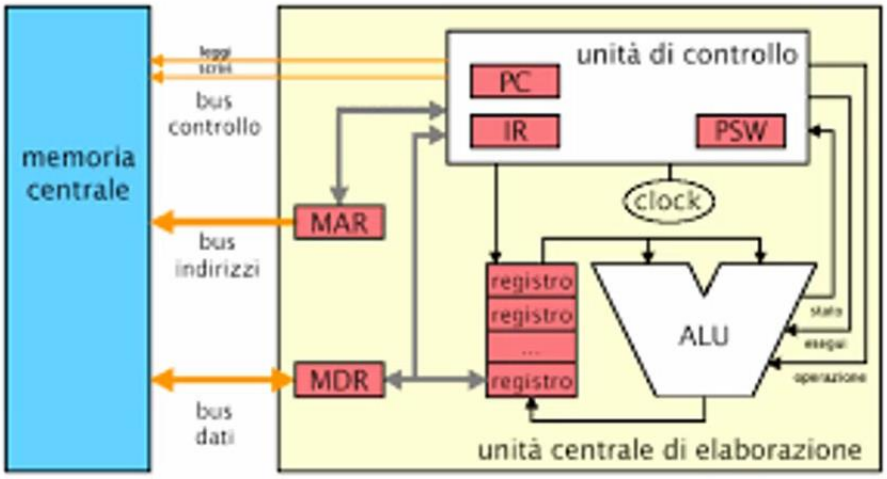
4.5 Richiami all'architettura di sistemi di elaborazione

La lezione di oggi è rivolta a tutti, in particolare a coloro che seguono dalla magistrale. Dobbiamo fissare bene i concetti principali su cui si basa l'architettura di un calcolatore. Non richiameremo tutti i concetti visti nei corsi precedenti.



L'architettura in figura si basa sul modello di Von Neumann, che influenza tutte le architetture moderne. I moduli presenti comunicano grazie al bus, che sono linee di comunicazione (la CPU comunica con i moduli e viceversa).

4.5.1 Struttura della CPU



Lo schema rappresentato in figura è uno schema a blocchi, cioè vediamo tutte le componenti fondamentali che caratterizzano la CPU. Distinguiamo, in particolare, unità di controllo e ALU. Teniamo conto dei concetti appresi a Reti logiche, in particolare riguardo le metodologie di realizzazione di una rete complessa e le nozioni su struttura del calcolatore (il calcolatore elementare che abbiamo descritto con Verilog).

4.5.1.1 Unità di controllo

L'unità di controllo è caratterizzata dalla presenza di tre registri: essi consistono in unità di memorizzazione, molto veloci rispetto alle singole locazioni della RAM (che invece immaginiamo come un vettore). Sono realizzate fisicamente all'interno del processore, hanno dimensione molto piccola, e permettono di fare operazioni con la stessa velocità del processore (tutto collegato allo stesso clock, se ho 1GHz potrò effettuare operazioni sul registro con velocità inferiore al nanosecondo). Essi sono:

- **PC** (*Program counter*, quello che abbiamo chiamato Instruction Pointer), che contiene l'indirizzo di memoria centrale della prossima istruzione da eseguire;
- **IR** (*Instruction register*), contiene l'istruzione che deve essere eseguita (dopo essere stata caricata);
- **PSW** (*Program Status Word*), registro di stato che serve per mantenere informazioni (si pensi ai flag manipolati dalle operazioni aritmetiche oppure il livello di privilegio)[3].

[3] Sì, è il registro dei flag.

I tre registri sono registri di stato e di controllo, normalmente chi programma in Assembler non ha accesso a questi (sono riservati al processore).

L'unità di controllo gestisce le fasi che caratterizzano il ciclo della macchina. Possiamo dire, a grandi linee (ricordiamo che il numero di fasi può differire da processore a processore), quali sono le fasi:

- **prelievo dell'istruzione** dalla memoria centrale (fase di fetch);
    - Il processore deve poter fare operazioni di lettura sulla memoria centrale per ottenere l'istruzione (e porla nel registro IR). La velocità è fondamentale nel compiere queste operazioni: emergono le problematiche viste a Calcolatori elettronici e conseguentemente la cache (con tutte le strategie connesse - cacheline e principi di località spaziale e temporale).
- **decodifica dell'istruzione**, attraverso i bit dell'istruzione dobbiamo capire di quale istruzione si tratta, ed eventualmente quali sono i suoi operandi (operazione tutta interna, non si coinvolge il bus o altre componenti del processore);
- **esecuzione dell'istruzione**, l'unità di controllo usa la ALU (Unità aritmetico-logica) per eseguire l'istruzione.

La regola di default è che il contenuto di PC venga aumentato: in alcuni casi viene solo incrementato, in altri (per esempio quando abbiamo costanti e/o indirizzi) aumentiamo di un certo valore. Le istruzioni di salto condizionato alterano questo comportamento di default: non andiamo ad aumentare, ma a sostituire il valore con un altro.

4.5.1.2 Bus

Abbiamo già detto in cosa consiste il bus. Attenzione alle frecce che vanno verso la memoria centrale: **sono bidirezionali**. Normalmente i fili non sono distinti in blocchi, quanto posto nell'immagine è una semplificazione. Distinguiamo:

- **Bus di controllo**
    - Tante linee quante sono necessarie per indicare il tipo di operazione sul bus.
    - Le operazioni di base sono lettura e scrittura.
        - Lettura prevede un trasferimento da modulo esterno verso CPU.
        - Scrittura prevede trasferimento da CPU verso modulo esterno.
    - Abbiamo anche un bit che indica lo spazio di indirizzamento (insieme di indirizzi che fanno riferimento a uno spazio virtuale, un vettore di locazioni) su cui stiamo lavorando. Tipicamente ne abbiamo due:
        - spazio di memoria (memoria centrale);
        - spazio di I/O, contiene gli indirizzi di tutti i registri che rappresentano le periferiche.
    - Per accedere a una periferica usiamo un protocollo simile a quello per accedere alla memoria: si rappresenta ciascuna periferica come una memoria RAM, e permettiamo l'interazione attraverso un certo numero di registri. Solitamente sono tre registri: registro di stato, registro di comando, registro di dati.
- **Fili di indirizzo**
    - Il numero di fili implica la dimensione dello spazio di indirizzamento.
    - Si ricordi le misure binarie viste nei corsi precedenti (le potenze di 2… vanno sapute a memoria, almeno fino a 10, tenendo conto di KB, MB, GB, TB).
- **Fili di dati**, poniamo i dati che vogliamo trasferire con le nostre transazioni (potrei avere un solo byte, così come ne potrei avere di più).

Si ricordi che il bus è bidirezionale: anche i moduli (master) possono avviare transazioni verso la CPU o verso altri moduli (slave). Può succedere che fili di indirizzi e fili di dati siano multiplexati, cioè uso gli stessi fili sia come fili di indirizzo che come fili di dati (si ripensi a cosa veniva fatto nel DMA, visto a Calcolatori elettronici).
Per i principi visti a Reti logiche abbiamo due ulteriori registri nella CPU:

- **MAR**, registro di appoggio per i fili di indirizzo;
- **MDR**, registro di appoggio per il trasferimento dei dati.

Protocollo nell'uso del bus. Dobbiamo definire un protocollo per le transazioni eseguite sul bus:

- abbiamo un dispositivo *master* che inizia la comunicazione, e tanti dispositivi *slave* in attesa;
- conoscendo il bus fisico a disposizione, e la sua velocità di trasferimento, posso determinare una durata fissa della transazione (quindi posso stabilire una deadline oltre la quale abortire la transazione in mancanza di risposta dagli slave);
- il master diventa trasmettitore, attiva i circuiti che si interfacciano col bus propagando le informazioni necessarie alla transazione;
- uno degli slave collegati al bus reagisce, solo a quel punto l'operazione viene effettivamente avviata e si ha un dialogo tra *master* e *slave*.

Nell'operazione di lettura lo *slave* ha un ruolo attivo poiché deve fornire dei dati a chi ha lanciato la transazione (al master). Nell'operazione di scrittura lo *slave* legge il bus dati e a memorizza gli elementi nelle locazioni dedicate. Alla fine del tempo tutti i segnali ritornano sul livello di riposo.

4.5.1.3 Unità aritmetico-logica (ALU)

L'esecuzione dell'istruzione è fatta dalla ALU (*Arithmetic Logic Unit*). L'ALU è letteralmente un sommatore: con le somme si può fare tutto, ricordare quanto visto a Reti logiche nel sEP8 (la rete combinatoria `alu_flag`).

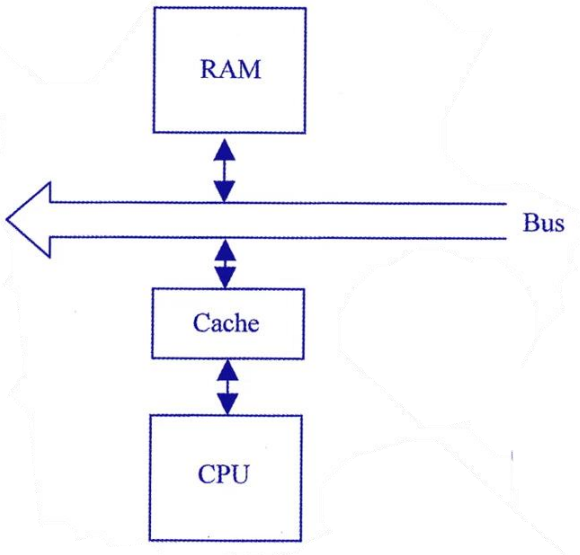
- Abbiamo una serie di linee in ingresso che permettono di configurare la ALU nell'operazione da svolgere. Il contenuto delle linee in ingresso è determinato al termine della fase di decodifica.
- Per svolgere le varie operazioni ci diamo supporto coi registri operativi, che contrariamente ai registri di controllo sono accessibili direttamente al programmatore. Sono collegati sia in ingresso che in uscita nella ALU (forniscono gli operandi in ingresso, ma permettono anche di memorizzare i risultati delle operazioni).
- Nelle architetture RISC, dove il numero di istruzioni è molto ridotto, i registri operativi assumono una certa importanza: ne abbiamo un numero maggiore e si limita il più possibile l'accesso alla memoria.

In un sistema operativo multiprogrammato dobbiamo ricordarci che lo stato complessivo della CPU è caratterizzato da tutti questi registri: se io voglio fare multiprogrammazione ogni volta che faccio cambio di contesto devo memorizzare in una struttura dati il valore dei registri operativi, oltre a tutte le altre informazioni che vanno a determinare il contesto del processo. Ricordarsi che queste operazioni contribuiscono all'overhead.

4.5.2 La memoria centrale e la cache, gerarchia delle memorie

La memoria centrale è implementata attraverso vari circuiti:

- **Memoria RAM**
  La RAM (Memoria ad accesso diretto, vettore…), memoria volatile che è un modulo separato, che sta al di la del bus.



- **Read Only Memory**
  Una parte della memoria principale è costituita dalla ROM (*Read only Memory*), una memoria di sola lettura, sempre ad accesso diretto. Alcuni modelli di ROM non sono immodificabili: si pensi alle EPROM, dove la lettura è alla pari della RAM, mentre la lettura risolta molto lenta e dispendiosa. La cosa non è un problema perché la modifica della ROM è un fenomeno molto raro.

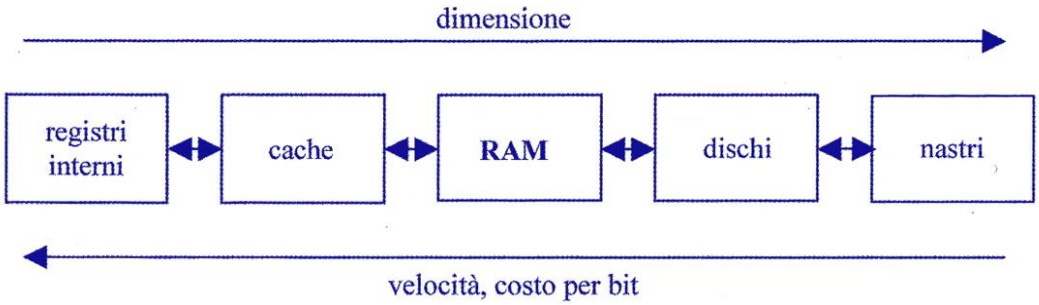
  Perché abbiamo bisogno di una ROM?
  La RAM è volatile, se tutta la memoria principale fosse volatile allora non avrei le istruzioni da far eseguire alla CPU non appena la collego all'alimentazione. La ROM, al contrario, è persistente. Il PC avrà come indirizzo quello relativo alla prima istruzione del programma bootstrap (nella ROM). Il programma eseguito quando avviamo il calcolatore è detto *programma bootstrap*.

- **Cache**
  In un mondo ideale la CPU esegue miliardi di istruzioni al secondo. Tuttavia, abbiamo visto che ogni istruzione da eseguire deve essere lettera nella memoria RAM. Introduciamo la *cache*, tenendo conto del **principio di località dei riferimenti** già visto a Calcolatori. Nella cache non poniamo singole righe ma blocchi di memoria (che identifichiamo attraverso parte dei bit più significativi).

La cache è anch'essa una RAM, molto piccola ma decisamente più veloce. Posso avere diversi livelli di cache, in alcuni casi la cache è direttamente integrata nella CPU (questo significa avere velocità al passo della CPU, che è una grande cosa).
La cache viene posta tra bus e CPU, quindi tutte le richieste provenienti dalla CPU vengono intercettate e controllate: se quanto richiesto è già presente in cache allora non abbiamo bisogno di muovere la richiesta sul bus e il numero di clock utilizzati risulta abbattuto (in questo caso avremo una hit, se la cosa non è presente si ha miss ed è necessario andare in RAM).

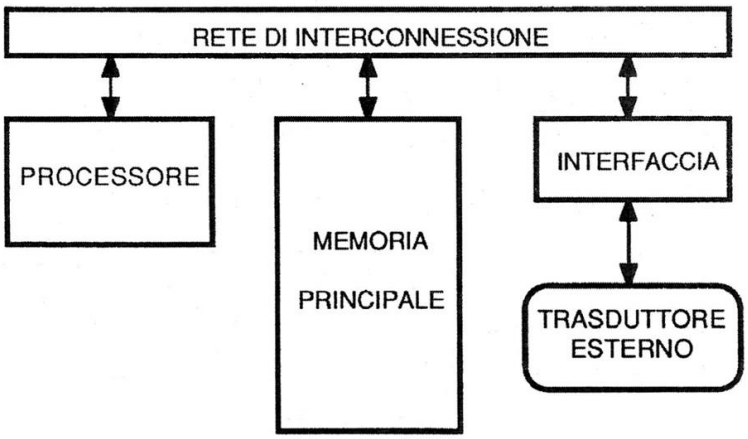
- **Esempio sul principio di località**: stiamo scorrendo una matrice, io non prenderò solo il primo elemento, ma anche quelli immediatamente successivi.

Con tutte le cose dette prima possiamo costruire una gerarchia delle memorie, dove ci si muove dalla memoria con capacità minore a quella con capacità maggiore, quindi dalla memoria con velocità più elevata a quella con velocità più bassa. Ricordiamo che non esistono memorie RAM veloci e con grande capacità. La cache è indispensabile per evitare sistemi troppo lenti, altrettanto irrinunciabile è la memoria a dischi permanente.



La velocità è inversamente proporzionale alla dimensione della memoria

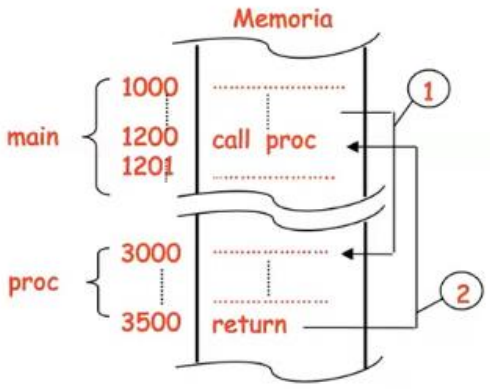
4.5.3 Periferiche

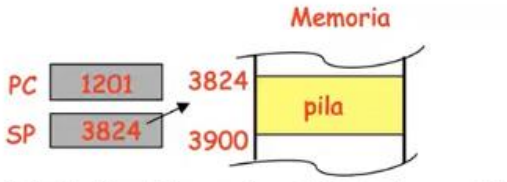


Le periferiche devono essere guardate in un modo univoco e generale (vogliamo vedere l'eterogeneità delle periferiche attraverso un unico protocollo).

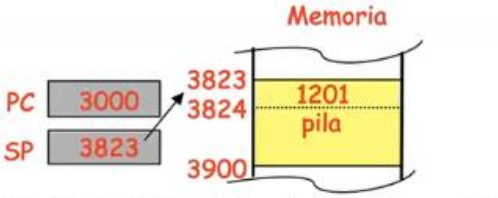
- Per ogni periferica abbiamo un'**interfaccia**, detta anche *controllore*, e un **trasduttore esterno** (che dipende dalla natura della periferica).
- La periferica è dedicata e parallela rispetto al processore, quindi può lavorare in contemporanea ad esso. Entrambi sono collegati al bus, dunque possono comunicare tra loro (anche attraverso meccanismi DMA, se previsti).
- Il controllore implementa i registri funzionali, quelli che rappresentano la visione funzionale della periferica: controllo, stato e dati.
- Il controllore esegue un firmware, cioè delle istruzioni poste in una memoria sua (non a caso si parla di *processo esterno*). Si osservi che il controllore può fare solo quello, non si ha esecuzione libera di istruzioni come nel processore.
- Vedremo che si hanno nel sistema operativo i *device driver*, ossia software eseguito per dialogare con un'interfaccia e quindi con la relativa periferica.

4.5.4 Istruzioni di CALL/RETURN

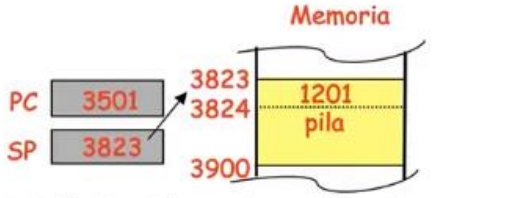




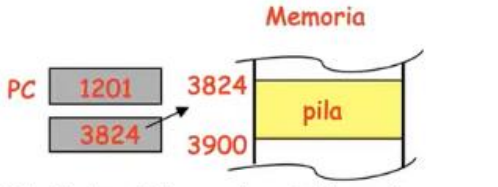
1a) all'inizio dell'esecuzione dell'istruzione **`call`**



1b) alla fine dell'esecuzione dell'istruzione **`call`**



2a) all'inizio dell'esecuzione dell'istruzione **`return`**



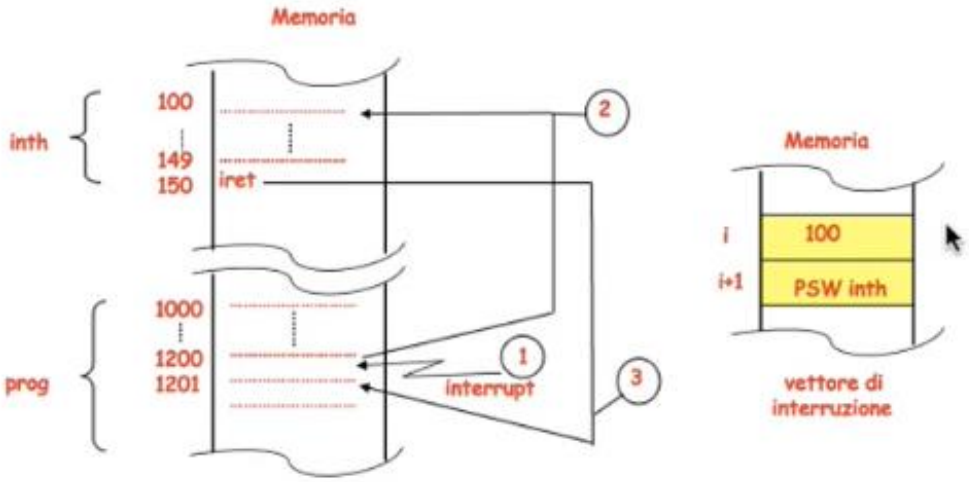
2b) alla fine dell'esecuzione dell'istruzione **`return`**

Il sottoprogramma è un blocco di istruzioni che può essere invocato, e quindi eseguito (solitamente con un passaggio di parametri in ingresso e in uscita), dal parametro in uscita. Una volta che il blocco di esecuzioni è eseguito si rientra al programma chiamante (metafora: parentesi che si apre e si chiude).

- L'esecuzione del sottoprogramma avviene grazie a un salto non condizionato: il blocco di istruzioni del sottoprogramma si trova in un'altra area di memoria rispetto al programma principale. *proc* è un nome simbolico, che verrà tradotto in un indirizzo numerico: il primo indirizzo del sottoprogramma.
- **Istruzione CALL**. L'implementazione dell'istruzione CALL prevede la memorizzazione in pila dell'indirizzo di ritorno (equivalente della PUSH), il decremento dello stackpointer e l'aggiornamento dell'indirizzo posto in PC. In questo modo al prossimo fetch la CPU caricherà la prima istruzione del sottoprogramma.
- **Istruzione RETURN**. L'implementazione dell'istruzione RETURN prevede il recupero dalla pila dell'indirizzo di ritorno (equivalente della POP), l'incremento dello stackpointer e l'aggiornamento di PC col valore estratto dalla pila. Ci ritroviamo nella situazione antecedente l'esecuzione dell'istruzione CALL.
- Attenzione ai vari step:
    - Stiamo eseguendo l'istruzione CALL all'indirizzo 1200:
        - All'inizio dell'esecuzione PC è 1201 (l'istruzione successiva), mentre SP punta a 3824.
        - Dopo l'esecuzione PC ha come valore 3000 (l'indirizzo della prima istruzione del sottoprogramma proc), mentre SP punta a 3823 (decrementato perché abbiamo posto in pila l'indirizzo di ritorno).
    - Stiamo eseguendo l'istruzione RETURN all'indirizzo 3500:
        - All'inizio dell'esecuzione PC è 3501 (indirizzo successivo), mentre SP punta a 3823 (se siamo alla RETURN e prima abbiamo messo ulteriori cose in pila allora tutto a posto, abbiamo già eseguito le POP necessarie per riportare SP a 3823)
        - Dopo l'esecuzione PC ha come valore 1201 (l'indirizzo di ritorno che avevamo memorizzato in pila), mentre SP punta a 3824 (abbiamo incrementato, visto che l'indirizzo di ritorno è stato rimosso dalla pila.

4.5.5 Meccanismo di interruzione

Il meccanismo di interruzione somiglia molto alla chiamata di un sottoprogramma. Durante le prime lezioni abbiamo detto che si deve dare la possibilità di rispondere a stimoli esterni: l'idea è appunto interrompere il flusso di esecuzione e lanciare una routine di sistema.



Sono così necessarie le interruzioni?

- In un sistema monoprogrammato il meccanismo non è così vitale: viene eseguito un solo programma, se questo lancia un'istruzione di I/O il processore rimarrà fermo.
- In un sistema multiprogrammato il meccanismo assume una grande importanza: se un programma lancia un'operazione di I/O questo cede il controllo a un altro programma. Ho bisogno di un meccanismo che mi permetta di risvegliare il programma sospeso dopo il termine dell'operazione di I/O.

Controllo del registro di status VS interruzioni

- La prima cosa che il programmatore pensa è fare un controllo periodico di un registro di stato, per verificare se il trasferimento si è concluso. La cosa è poco efficiente: si parla di ***busy wait***, la CPU è usata inutilmente.
- Il meccanismo delle interruzioni è decisamente più efficiente rispetto alla lettura continua di un registro. Quando avvio l'operazione di I/O abilito la relativa periferica a lanciare un segnale di interruzione (*call back*, visto che segnala la fine dell'operazione). Il programma che si è posto in attesa, dopo aver chiamato l'istruzione, viene temporaneamente escluso dalla coda dei processi (non deve essere eseguito finchè l'istruzione non risulterà conclusa).

In quale momento conviene interrompere la macchina?

- Abbiamo detto che ci sono tre fasi: *prelievo, decodifica, esecuzione*.
- Non interromperemo la CPU tra una fase e un'altra, ma solo alla fine del ciclo (dopo l'esecuzione).
- Se una o più interruzioni sono in attesa queste vengono eseguite: la CPU smette di eseguire il programma relativo al flusso e inizia ad eseguire gli ***interrupt handler***.
- In memoria abbiamo un **vettore di interruzione**, che contiene informazioni relative alle interruzioni esterne. Per ogni entrata abbiamo:
    - il contenuto del program counter (indirizzo della prima istruzione dell'interrupt handler), e
    - il program status (registro PSW).

  Ogni interruzione è identificata da un indice i-esimo, che utilizziamo per leggere le relative informazioni nel vettore di interruzione.

Cambio di privilegio

- Risulta necessario modificare i privilegi con cui il processore sta operando. Normalmente, nell'esecuzione di programmi applicativi, il processore lavora in una modalità a basso privilegio. Nell'esecuzione di una routine dobbiamo innalzare il livello di privilegio:
    - salvo in pila il contenuto attuale di PSW;
    - sostituisco il valore attuale di PSW con quello nel vettore di interruzione.

  Dopo questo *overhead* la macchina riprende il ciclo normale, e in modo inconsapevole esegue l'handler.
- Come in un qualunque sottoprogramma dobbiamo ritornare al programma che è stato interrotto (differenza rispetto al sottoprogramma, non è lui che chiama ma qualcuno esterno che lo interrompe). Nella pila abbiamo tutte le informazioni necessarie, basta fare le operazioni inverse.

Tipologie di interruzioni

Fino ad ora abbiamo parlato solo di interruzioni esterne, che permettono una sincronizzazione tra periferiche esterne e processore. In realtà non sono le uniche interruzioni!

- **Interruzioni hardware**
  Interruzioni lanciate in caso di malfunzionamenti hardware (per esempio surriscaldamento).
- **Eccezioni**
  Le eccezioni sono interruzioni lanciate a causa dell'esecuzione di istruzioni (per esempio la divisione per zero). Eccezione a livello logico, interruzione a livello fisico.
- **Interruzioni interne/software**
  Le interruzioni software sono generate in modo esplicito da un programma in esecuzione. Questo meccanismo è INDISPENSABILE per fare multiprogrammazione, pensiamo a quando vogliamo fare preemption e revocare l'assegnazione del programma alla CPU. Tutte le volte che evochiamo una funzione del sistema operativo si genera un'interruzione software, che permette di innalzare il privilegio.

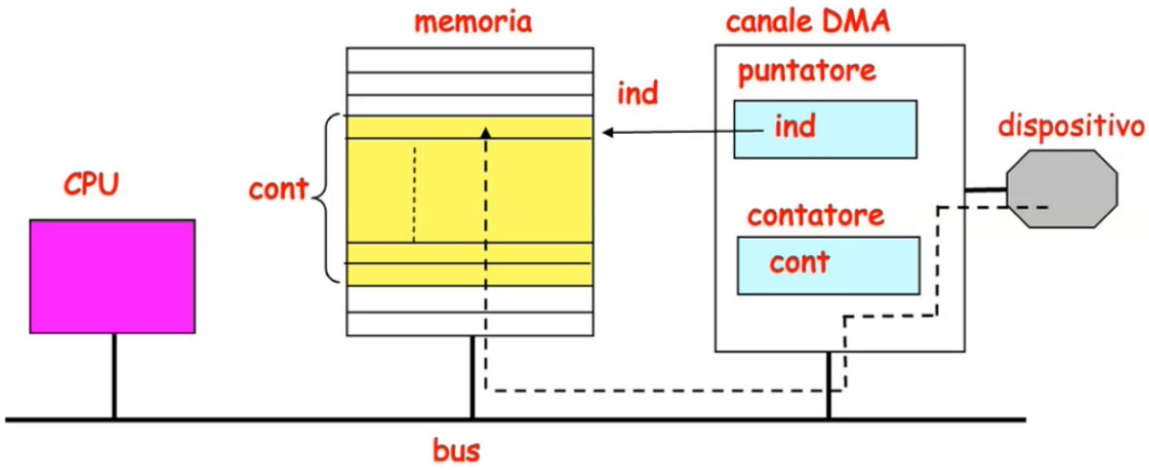
Considerando le interruzioni esterne abbiamo ben quattro tipologie di interruzioni.

Modalità di funzionamento espressa dal registro PSW

Abbiamo due modalità:

- Modalità utente.
  - Usato per la normale esecuzione dei programmi.
  - Non possiamo fare tutto: non possiamo accedere alle risorse di sistema, per esempio ai dispositivi di I/O.
- Modalità kernel
  - Usato per lo svolgimento dei servizi richiesti al sistema operativo tramite le system call.
  - Non esistono limiti alle operazioni effettuabili.
  - Si mascherano le interruzioni, rendendo così atomiche le system call.

4.5.6 Direct Memory Access (DMA)



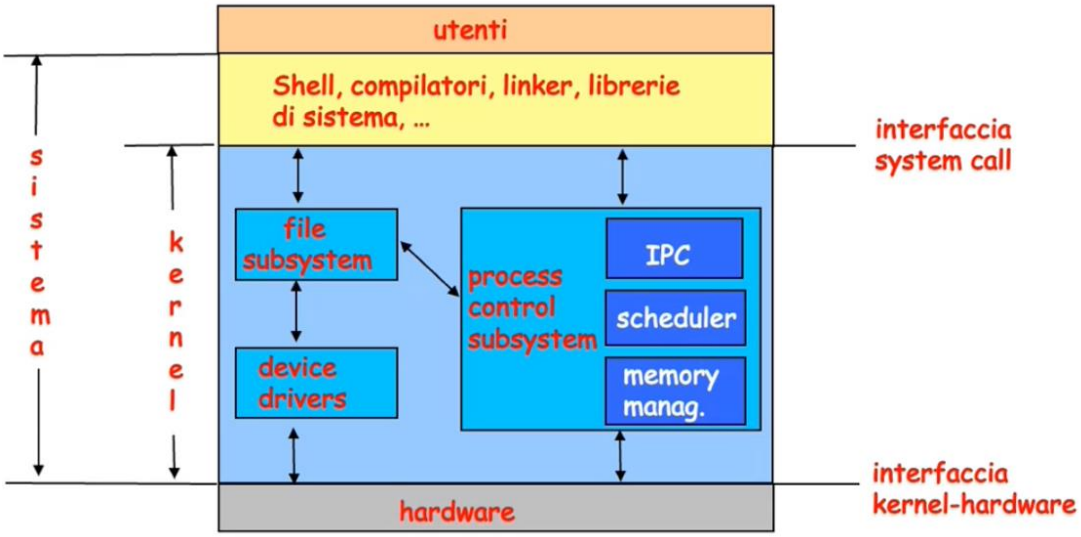
Il DMA è un dispositivo hardware che permette di gestire trasferimenti tra periferiche e memoria in modo indipendente rispetto alla CPU. Quando io richiedo di fare il trasferimento di un certo numero di byte devo avere un *interrupt handler* che mi permette di fare ciò. Questa cosa, tuttavia, tiene impegnata la CPU. L'idea è di delegare la cosa a un circuito hardware: il canale DMA. Esso è visto dal processore come una nuova periferica (quindi un'interfaccia) avente due registri:

- **registro puntatore**, inizializzo con l'indirizzo di memoria dove si trova il buffer coinvolto nel trasferimento (sia in operazioni di ingresso che di uscita);
- **registro contatore**, numero di byte oggetto del trasferimento.

Quando abbiamo inizializzato il canale DMA questo, in parallelo al processore principale, sfrutterà i cicli in cui il bus è libero (e quindi non usato dal processore) per effettuare questo trasferimento. Ogni operazione avverrà sfruttando i dati presenti nei registri: si aumenta il registro puntatore e si decrementa il registro contatore. A conclusione del trasferimento viene lanciato dal DMA un'interruzione esterna: la CPU prende atto, nei modi decisi dal software, del trasferimento appena avvenuto. Un altro vantaggio è che il trasferimento ha un tempo costante, cosa molto utile nei sistemi in tempo reale.

4.5.7 Struttura del sistema operativo

Inizialmente il sistema operativo era un qualcosa di monolitico: tantissime istruzioni Assembler senza grossi pensieri sulla modularità, le funzioni venivano chiamate con le CALL. L'assenza di modularità e struttura non è una bella cosa: esse permettono di ridurre gli errori e di poter sostituire intere aree del sistema operativo (risoluzione dei bug, miglioramento delle prestazioni…) senza dover modificare tutto il resto.

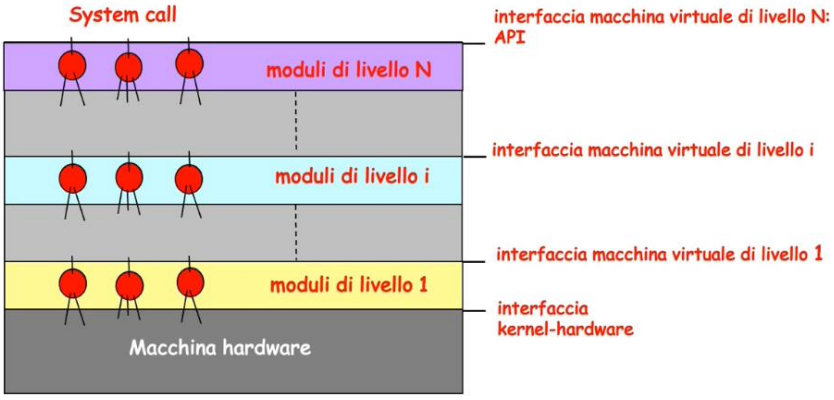


Il sistema operativo che vedremo è UNIX, che si poggia su quattro livelli:

- **Hardware**, con la sua interfaccia (istruzioni a livello macchina);
- **Kernel**, che contiene tutti i moduli principali richiamati prima
  - il Process control subsystem (che implementa al suo interno lo scheduler, il gestore della memoria, e il controllore delle interruzioni IPC)
  - il file subsystem;
  - i device drivers.

  Notare che il PCS verso il basso richiama funzioni dell'interfaccia hardware (linguaggio macchina, essendo meccanismi si cambiano raramente), mentre il file system dialoga sia col PCS che coi drivers. In UNIX, ricordiamo, ogni risorsa è vista come un file: questo aiuta a scrivere un'interfaccia di programmazione pulita e con poche funzioni.
- **Shell, compilatori, linker, librerie di sistema** (API!!!)… Questo strato e il precedente costituiscono il sistema.
- **Utenti**

4.5.7.1 [IMPORTANTE] Generalizzazione di una struttura a livelli gerarchici



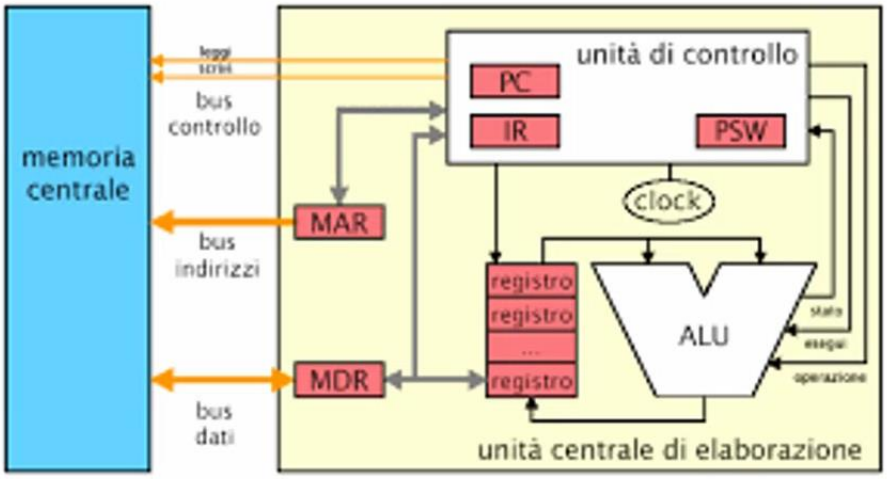
Ogni sistema ingegneristicamente complesso non è monolitico, ma organizzato a strati.

- Ogni strato implementa delle funzionalità, solitamente utilizzate dallo strato superiore. Lo strato a sua volta utilizza funzionalità implementate dallo strato inferiore.
- Le funzionalità che ci fornisce lo strato inferiore sono presentate attraverso un'interfaccia, che possiamo immaginare come l'insieme delle intestazioni delle funzioni che lo strato può utilizzare. **Nell'interfaccia abbiamo solo l'intestazione**, mentre l'implementazione si trova nel livello inferiore (e non è interesse dello strato superiore conoscere la precisa implementazione della funzionalità invocata).

L'esempio siamo noi programmatori, a livello applicativo, che utilizziamo le primitive fornite dal sistema operativo attraverso l'interfaccia tra livello applicativo e livello del sistema operativo: conosciamo l'intestazione (presente nell'interfaccia), abbiamo un'idea delle conseguenze, ma non conosciamo l'implementazione precisa.

4.6 Alcuni problemi del libro

Primo problema. *Nei sistemi multiprogrammati, il numero massimo di processi sotto il controllo del Sistema Operativo a un dato istante prende il nome di grado di multiprogrammazione. Dal punto di vista teorico tanto più è grande il grado di multiprogrammazione tanto maggiore è la probabilità che la CPU trovi sempre un processo da mettere in esecuzione e che quindi cresca il suo utilizzo percentuale. In pratica questo non si verifica perché al crescere del numero dei processi si presentano altri problemi. Discutere di quali problemi si tratta.*



I sistemi multiprogrammati consentono che un processo P possa essere interrotto prima del completamento della sua esecuzione ed il controllo della CPU trasferito ad un altro processo Q. I casi in cui questa cosa può avvenire sono:

- La sospensione del processo, nel caso di invocazione di un'istruzione di I/O
- La revoca della CPU al processo, nel caso di sistemi dotati di *preemption*.

L'interruzione dell'esecuzione di P comporta il salvataggio di tutti i risultati fino ad ora prodotti e di tutte le informazioni che sono necessarie affinché il processo possa successivamente riprendere la sua esecuzione dal punto in cui era stato interrotto. Queste informazioni comprendono

- i valori dei registri non operativi (PC e PSW, il primo è l'indirizzo da cui il processo dovrà riprendere l'esecuzione quando vedrà la CPU nuovamente in suo possesso, il secondo contiene i flag)
- il valore del registro Stack Pointer (SP) che punta alla cima dello stack del processo
- i valori contenuti nei registri operativi (registri indice, accumulatore, registri per tenere traccia della memoria utilizzata dal processo, etc..).

Questa funzione di salvataggio viene eseguita dall'hardware per quanto riguarda i registri PC e PSW e dal sistema operativo per tutti gli altri registri. Una situazione analoga si presenta per la messa in esecuzione del nuovo processo Q. Occorre infatti che il sistema operativo provveda alle operazioni di caricamento sul registro SP e sui registri generali del processore di tutte le informazioni relative a Q e l'hardware a ripristinarne i valori di PC e PSW.

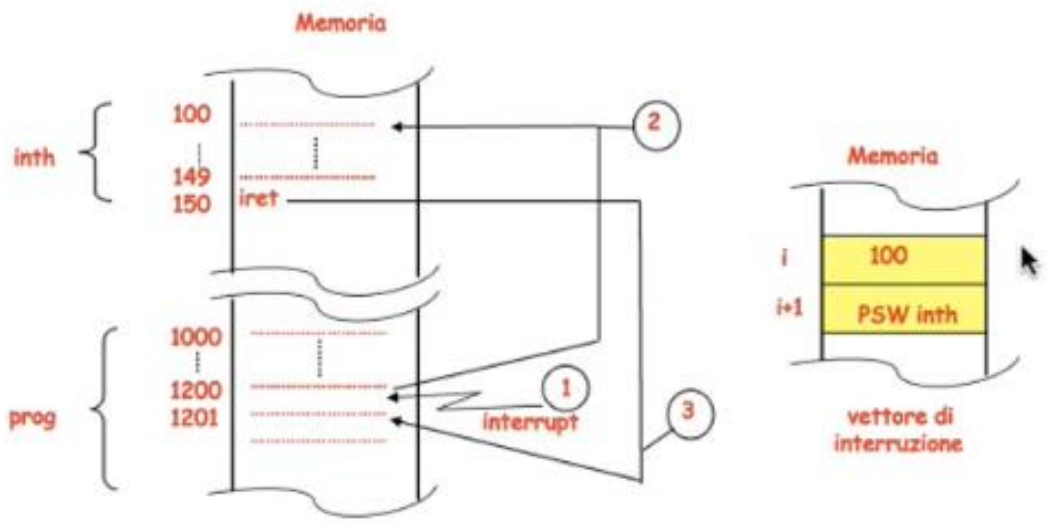
L'insieme delle operazioni necessarie per il passaggio dal processo P al processo Q, che prende il nome di **cambio di contesto**, impegna la CPU riducendo quindi il tempo che essa può dedicare alla esecuzione dei processi. Si indica con il termine ***overhead*** il tempo dedicato dalla CPU per le operazioni di cambio di contesto.

La crescita del numero dei processi presenti contemporaneamente in memoria centrale può portare ad una crescita dell'overhead. È quanto succede, ad esempio, in un sistema timesharing al crescere del numero di utenti collegati.

Secondo e terzo problema (a cui diamo una risposta unica).

- *Si descriva la sequenza di operazioni provocate dall'arrivo di un segnale di interruzione esterno in entrambe le ipotesi che il servizio dell'interruzione sia interrompibile o non interrompibile (contenuto in risposta alla domanda **evidenziato in rosso**).*
- *L'interruzione di un processo in un ambiente multiprogrammato comporta il salvataggio di tutte le informazioni che sono necessarie per riprendere successivamente la sua esecuzione. Descrivere il processo di salvataggio indicando quali informazioni devono essere salvate, chi esegue l'operazione di salvataggio e dove queste informazioni devono essere memorizzate.*

Sia P il processo attualmente in esecuzione. All'arrivo di un segnale di interruzione, sia esso esterno (proveniente cioè da un dispositivo di I/O) o interno (causato cioè da una system call) la sequenza di operazioni hardware e software è la seguente:



a) salvataggio del valore dei registri non operativi PC e PSW in cima allo stack associato a P il cui indirizzo è contenuto in SP.:
b) inserimento in PC e PSW dei valori relativi alla routine di risposta all'interruzione prelevati dal **vettore di interruzione**[4].
c) **eventuale disabilitazione del sistema di interruzioni mediante esecuzione di un'apposita istruzione** (se si guarda i capitoli successivi l'istruzione dovrebbe alterare un bit del registro di stato della relativa periferica, in modo da mascherare le interruzioni);
d) salvataggio dei valori dei registri operativi e di SP del processore utilizzati da P in un'area dati privata di P (l'array *contesto* del descrittore di processo);
e) esecuzione del codice della routine di risposta all'interruzione;
f) ripristino dei valori dei registri generali e di SP relativi a P nei registri del processore (**fondamentale farlo prima del prossimo punto**, altrimenti si corre il rischio di non consistenza a causa del lancio di ulteriori interruzioni);
g) **eventuale riabilitazione del sistema di interruzioni**;
h) Inserimento dei valori di PC e PS di P nei rispettivi registri del processore attraverso l'esecuzione dell'istruzione `iret` (nessun rischio di essere interrotti, la cosa viene fatta a livello hardware da un'istruzione Assembler che è atomica).

L'algoritmo di scheduling viene chiamato in causa ogni volta che bisogna assegnare la CPU a un qualche processo (si ricordi che è un *non sense* affermare che un processo richieda la CPU, visto che non potendo eseguire istruzioni non può formalmente richiederla). Non è detto che l'algoritmo risvegli lo stesso processo precedentemente sospeso dall'interruzione: **dipende dalle priorità stabilite**. A seguito della scelta del processo si recupera dal relativo descrittore il contenuto di tutti i registri (si capisce da questo che l'algoritmo di scheduling viene chiamato tra il punto e ed il punto f).

[4] Si ricordi che il **vettore di interruzione** è una tabella posta in memoria, contenente le informazioni relative alle tipologie di interruzioni: per ogni elemento indicato il valore del PSW (che conterrà il livello di privilegio da adottare nell'esecuzione dell'handler) e l'indirizzo della prima istruzione dell'handler da eseguire.

4.7 Concetti per la comprensione di sistemi operativi distribuiti

4.7.1 Classificazione delle architetture (tassonomia di Flynn)

La tassonomia di Flynn classifica i sistemi di elaborazione da due punti di vista:

- **Flusso di istruzioni** (*instruction stream*, che abbiamo presente fin da Reti logiche);
- **Flusso di dati** (*data stream*), sequenza di strutture dati (di qualunque natura) che sono oggetto di elaborazione da un flusso di istruzioni.

In presenza di un unico flusso di istruzioni si parla di *single instruction stream*, altrimenti si ha un *multiple instruction stream*. In presenza di un unico flusso di dati si parla di *single data stream*, altrimenti si ha un *multiple data stream*.

Otteniamo la seguente tabella, quindi quattro macchine possibili…

	Single Instruction stream (SI)	**Multiple Instruction stream (MI)**
Single Data stream (SD)	Macchine SISD	Macchine MISD
Multiple Data stream (MD)	Macchine SIMD	Macchine MIMD

Dobbiamo vedere, per ogni tipologia di macchina, due cose: l'architettura concettuale, e se effettivamente esistono.

4.7.1.1 Macchine SISD (Modello di Von Neumann, macchina classica)

Consideriamo il processore. Esso ha al suo interno almeno due sottounità: l'unità di controllo (gestione delle istruzioni prima dell'esecuzione) e l'ALU (dedicata all'esecuzione delle istruzioni). Si distingue a tal proposito *instruction processor* (decodifica) da *data processor* (esecuzione, si elaborano due operandi ottenendone un terzo).

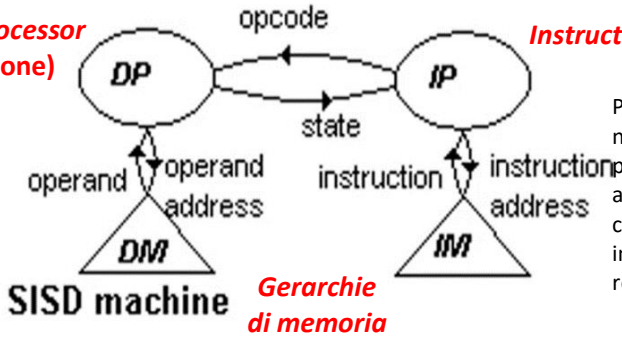
Flynn applica gli stessi ragionamenti anche alle memorie. Col triangolo si rappresenta delle gerarchie di memorie. Come prima si distingue una *instruction memory* da una *data memory*: la prima contiene le istruzioni, la seconda gli operandi.

(3) A seguito della decodifica IP deve fornire a DP l'opcode dell'istruzione da eseguire: lo fa attraverso l'omonimo bus. Dopo l'esecuzione DP comunica a IP i flag: lo fa attraverso il bus state.

***Data processor* (esecuzione)**

(2)

Nell'esecuzione la ALU ha bisogno di operandi. Li otteniamo attraverso due bus distinti: uno da DP a DM con cui si indica l'indirizzo dell'operando alla memoria, l'altro da DM a DP con cui la memoria restituisce l'operando richiesto.



***Instruction processor* (decodifica)** **(1)**

Prima della decodifica dell'istruzione è necessario che avere all'interno del processore l'istruzione da eseguire. Tra IP e IM abbiamo due bus distinti: uno da IP a IM con cui si richiede l'istruzione posta a un certo indirizzo, un altro da IM a IP con cui la memoria restituisce l'istruzione richiesta.

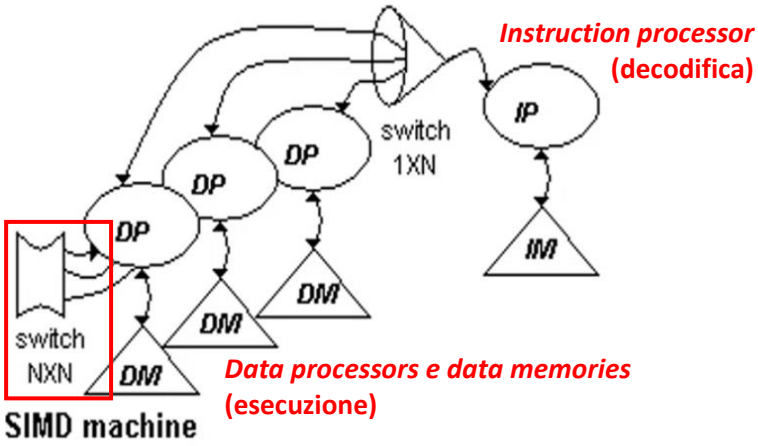
Si tenga conto che molto spesso i bus vengono "multiplexati", la separazione dei due bus è una questione soprattutto concettuale. Nel disegno non si evidenziano i trasferimenti di operandi sorgente da IP a DP: in realtà gli operandi sono dati e quindi stanno in DM.

La capacità della macchina è unica: esegue un unico flusso di istruzioni su unico flusso di dati. Questo si capisce perché abbiamo un unico IP e un unico DP.

4.7.1.2 *Macchine SIMD (parallelismo spaziale)*

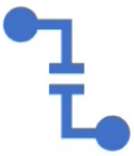
Abbiamo un solo IP, quindi una sola IM. Abbiamo una serie di DP, ciascuno con la sua DM.

Attenzione allo switch NXN: è difficile immaginare un programma dove si lavora su sottoinsiemi di dati distinti senza farli dialogare. Fra sottoinsiemi diversi potrei avere dati comuni, che si trovano su una delle DM. Le gerarchie di memorie, attenzione, sono private: essendo private devo realizzare una qualche forma di comunicazione. La soluzione è uno switch NXN che permette a coppie di DP di scambiarsi messaggi.



In sostanza abbiamo un insieme di macchine sequenziali messe insieme. Il "megafono" in figura rappresenta una rete di comunicazione tra IP e tutte le DP (comunicazione broadcast, *switch 1XN*), un bus che permette di trasferire l'OPCODE e lo stato (bidirezionalità) tra IP/DP e DP/IP. Se IP trasmette l'OPCODE tutte le DP lo ricevono: questo implica che tutte le DP eseguono in contemporanea le stesse istruzioni, visto che si ha una sola IP. Si ha una distribuzione dei dati tra le varie gerarchie di memoria DM: ciascuna lavora su un sottoinsieme dei dati. Il parallelismo è dovuto non alle istruzioni, ma ai dati.

- **Esempio**: Supercomputer (pensati per complessi calcoli matematici). Si osservi che non tutti gli algoritmi/programmi traggono vantaggio dalla parallelizzazione dei dati (si pensi a Teams, difficilmente un'applicazione del genere trarrà vantaggio dalla parallelizzazione).
- **Dove sono attaccate le periferiche**? Alle DM, le periferiche scambiano tra memoria e mondo esterno, quindi sono decisive nel fornire gli operandi necessari all'esecuzione delle istruzioni.
- Esistono compilatori che offrono un eseguibile adatto per macchina SIMD. Si veda il seguente esempio:



Pensiamo al prodotto scalare o al prodotto tra matrici. Non eseguiamo i prodotti in sequenza, ma lo facciamo in parallelo per poi sommarli.

Diversità di funzionamento tra un processore scalare ed un processore vettoriale

Esempio

c = a + b;

Processore scalare: gli operandi sono scalari

Processore vettoriale: gli operandi sono vettori

Compilatore vettoriale

Esempio

int i = 0;

for (; i < 10; i++)

•c[i] = a[i] + b[i];

Riconosce tutti quei cicli sequenziali trasformabili in un'unica operazione vettoriale

4.7.1.3 *Macchine MISD (parallelismo temporale, pipeline)*

- Macchine strane (cit.): abbiamo più flussi di istruzioni che lavorano sullo stesso flusso di dati (un po' come avere tanti programmi che lavorano sulla stessa matrice).
- Non esistono macchine di questo tipo, a meno che non si crei una pipeline. Per eseguire un'istruzione sono necessarie diverse fasi: in una pipeline ho singoli processori, ciascuno dedicato a una singola fase (in contrasto con l'esecuzione sequenziale tipica di un processore monolitico).
- La cosa non è così strana: l'abbiamo nel processore. La cosa è paragonabile a una catena di montaggio: l'unità di controllo si dedica al fetch, mentre la ALU si dedica all'esecuzione.
- Il *throughput* di istruzioni è molto più alto: nell'unità di controllo avviene il fetch di un'istruzione mentre nella ALU viene eseguita un'istruzione precedente. Se consideriamo questa cosa come una serie di flussi distinti allora si può parlare di macchina MISD (la questione può essere lasciata ai filosofi).

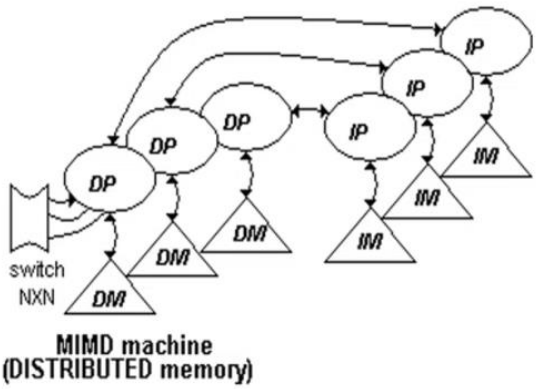
4.7.1.4 Macchine MIMD (macchine SISD in parallelo)

Le macchine MIMD presentano molteplicità in flussi di istruzione e flussi di dati, dunque abbiamo tanti IP e tanti DP. Esistono due tipologie di queste macchine:

- **A memoria distribuita**

  Ogni DP ha la sua DM dedicata, e ogni IP ha la sua IM dedicata. Ogni coppia IP-DP costituisce una macchina sequenziale SISD (quella classica, la prima vista nella classificazione).

  Distribuita poiché ogni data processor può accedere a un solo spazio di indirizzamento. Se vogliamo avere applicazioni che prevedono scambi di informazioni tra flussi di esecuzioni distinti inevitabilmente serve una nuova rete, guarda caso **abbiamo lo switch NXN**. Il modello di comunicazione è uno scambio di messaggi



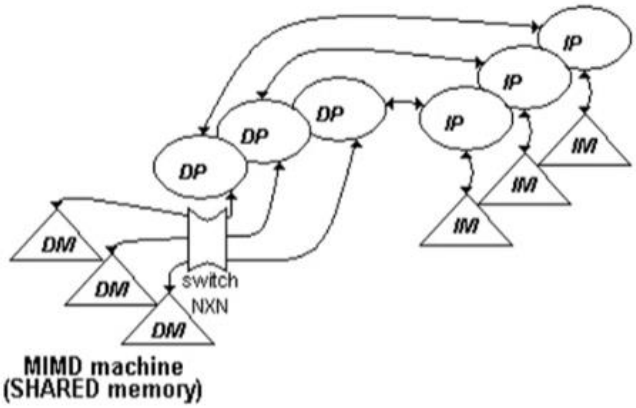
MIMD machine
(DISTRIBUTED memory)

  - **Esempi**:
    - i *multicomputer*
      - Abbiamo tanti computer distinti, che hanno una rete per comunicare e quindi possono essere usati per eseguire applicazioni che prevedono intensa comunicazione tra i flussi di programma.
      - La rete di interconnessione ha una banda molto larga, è compatta, estremamente costosa e regolare nella forma.
      - Se voglio avere scalabilità, cioè fornire prestazioni e qualità del servizio che non cambia all'aumentare delle richieste del servizio allora il multicomputer è la soluzione più ideale. La scalabilità è elevata: posso collegare nuovi moduli facilmente (in caso di reti più elaborate la cosa potrebbe essere un po' più complicata).
      - Macchina che lavora bene con algoritmi ad elevata località, **cioè se l'algoritmo raramente richiede lo scambio di informazioni tra flussi**.
    - *Cluster of Workstations*, con le seguenti funzionalità…
      - Alta disponibilità (high availability): in caso di guasti la computazione può migrare da un nodo all'altro (il sistema redistribuisce il carico di lavoro).
      - Load-balancing: le macchine devono essere sempre attive, i task sono allocati dal sistema nei nodi col minor carico.

      Queste caratteristiche sono spesso attribuite a uno strato software detto *middleware*, tutto ciò che sta tra il sistema operativo e l'applicazione.

- **A memoria condivisa (multiprocessore, l'architettura ideale)**

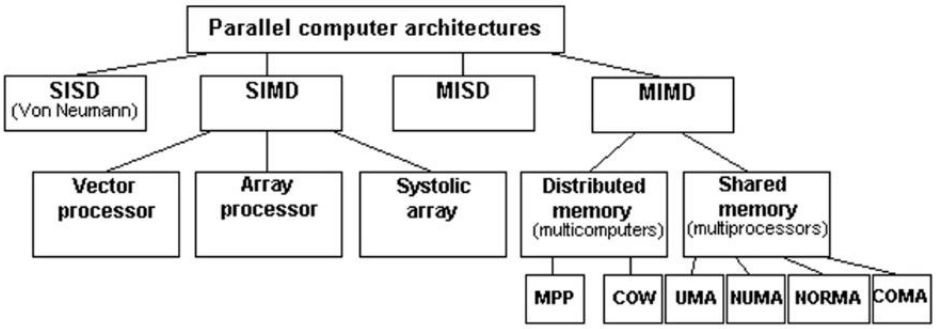
  Anche qua abbiamo tanti IP col relativo IM e la relativa DP. La novità sta nella memoria condivisa, ogni gerarchia di memoria DM è raggiungibile dalle DP attraverso uno switch NXN (tra tutte le DM e tutte le DP, posso avere in parallelo tante comunicazioni tra DP e DM).



MIMD machine
(SHARED memory)

  Si parla di multiprocessore. La cosa è diversa da multicomputer: i nodi sono minori rispetto al multicomputer, questo per avere uno switch efficiente. La scalabilità è limitata.

4.7.1.5 Recap

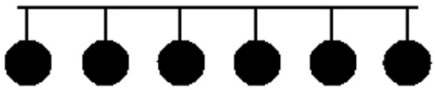


4.7.2 Topologie di interconnessione

Alcuni concetti:

- Il **grado** è il numero di connessioni/link che il nodo deve avere per essere collegato alla topologia.
- Il **diametro** è la distanza massima necessaria per passare da un nodo a un altro all'interno della topologia.
- Il **numero di link** è, banalmente, il numero di collegamenti presenti all'interno della topologia.

4.7.2.1 Bus



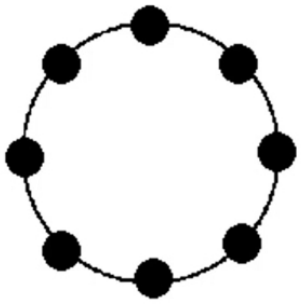
- La rete di interconnessione più semplice. Insieme di linee, con i pallini che rappresentano i vari nodi.
- Basta che un nodo si interfacci con un'unica interfaccia al bus per ottenere un sistema di comunicazione.
- Lo posso espandere con una certa semplicità (sfrutto la sua lunghezza, tenendo conto che non è possibile allungarlo troppo – si consideri che maggiore lunghezza significa tempo di risposta maggiore).
- **Grado**: 1 per ogni nodo.
- **Diametro**: 1
- **Numero di link**: 1, c'è solo il bus.
- **Svantaggio**: il numero di link basso comporta una forte competizione nell'accesso al mezzo. Uno alla volta, si ha mutua esclusione. Se ho più nodi che desiderano accedere al bus questi devono essere messi fila da un arbitro seguendo certe politiche.

4.7.2.2 Array lineare

- Se io parto dal bus e aumento il numero di link, cioè immagino di avere dei collegamenti punto-punto tra i nodi, allora vado a costruire un *array lineare*.
- **Grado**: I nodi sono in fila, il grado del primo e dell'ultimo nodo è 1, quello dei nodi intermedi 2.
- **Diametro**: n-1. Se i due nodi estremi vogliono scambiarsi un messaggio questo dovrà passare tra tutti i nodi intermedi. Il tempo necessario è maggiore, in più i nodi intermedi devono fare routing (dunque segue un carico computazionale maggiore).
- **Numero di link**: n-1.
- **Vantaggi**: il numero di link maggiore comporta la riduzione della competizione[5] al minimo, ho dei link dedicati.
- **Tolleranza ai guasti**: molto bassa. Se si rompe un nodo la rete si divide in due parti che non sono in grado di comunicare tra di loro.

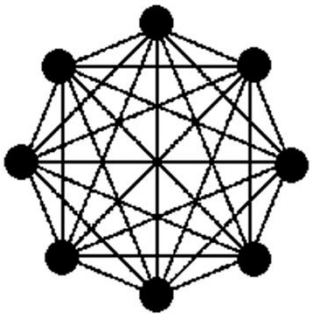
[5]La competizione si ha quando due nodi vogliono trasmettere in contemporanea: non è questo il caso, le comunicazioni in contemporanea possibili sono pari al numero di coppie di nodi.

4.7.2.3 Ring



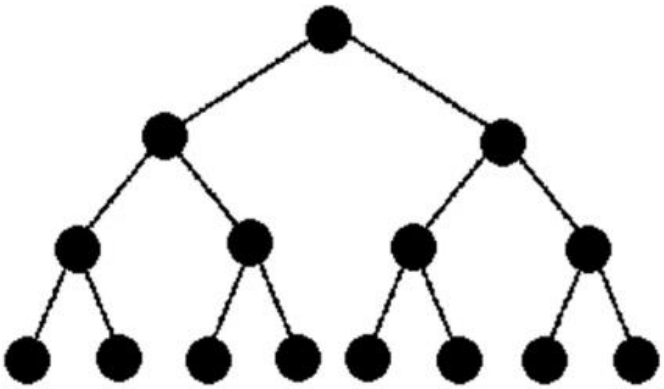
- Il ring è letteralmente un array lineare chiuso collegando i due estremi: l'operazione costa poco elettricamente parlando e da grandi vantaggi (in particolare rispetto alla velocità).
- **Grado**: 2 per tutti i nodi.
- **Diametro**: $\left\lceil \frac{N}{2} \right\rceil$
- **Numero di link**: Il numero di link è pari a N, tanti quanti i nodi presenti sul ring (abbiamo aggiunto il collegamento tra i due estremi dell'array lineare).
- **Tolleranza dei guasti**: può tollerare un guasto. Se si rompe un nodo la rete diventa un array lineare.

4.7.2.4 Connessione completa (tutti a tutti)



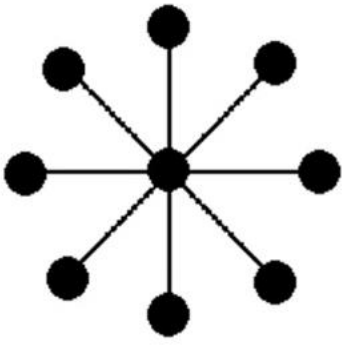
- **Grado**: n-1 per tutti i nodi (l'idea è che mi connetto a tutti).
- **Diametro**: 1, collegamento diretto tra tutti i nodi.
- **Numero totale di link**: $\frac{n(n-1)}{2}$
- **Svantaggi**: scarsa scalabilità, a un certo punto i fili iniziano a sovrapporsi. Se ho un certo numero di livelli questi prima o poi si esauriranno.

4.7.2.5 B-tree



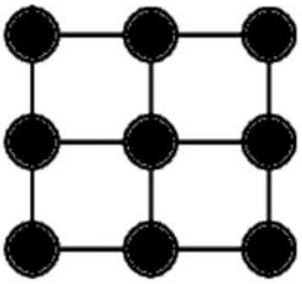
- Albero binario
- **Grado**: al più tre, la radice ha grado 2 e le foglie grado 1.
- **Diametro**: 2*(h-1)
- **Numero totale di link**: N-1
- **Tolleranza ai guasti**: non proprio vantaggioso. Se ho un guasto in vicinanza di una foglia non ho grossi problemi, ma se si rompe un nodo in vicinanza della radice potrei perdere una parte consistente dell'albero. Altra debolezza è la maggiore congestione in prossimità della radice, più mi allontano dalla radice e minore è la congestione (fino a diventare dedicato, con le foglie). La cosa non è bella: per risolvere adotto una topologia *fat-tree*, dove la banda è direttamente proporzionale alla vicinanza alla radice.

4.7.2.6 Star



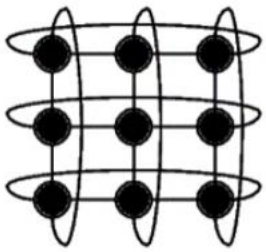
- **Grado**: sistema disomogeneo, elemento centrale di grado N-1 e tutti gli altri elementi di grado 1.
- **Numero di link**: n-1
- **Diametro**: 2, al più due spostamenti se voglio andare da un nodo a un altro diverso da quello centrale.
- **Tolleranza dei guasti**: dipende fortemente dalla robustezza del nodo centrale. Se si guasta il nodo centrale si guasta tutto.

4.7.2.7 Mesh bidimensionale



- Connessione bidimensionale.
- **Grado**: al più 4, i vertici hanno grado due, i nodi interni quattro, i rimanenti lungo la frontiera tre.
- **Diametro**: 2*(r-1).
- **Vantaggi**: scalabilità alta, mi basta collegare l'elemento senza dover manipolare la struttura già esistente.
- **Tolleranza ai guasti**: molto buona, se si rompe un link la rete non si spacca in due, posso individuare percorsi alternativi.

4.7.2.8 Torus bidimensionale



- Sfruttando il grado 4 della mesh bidimensionale ottengo il toro.
- **Grado**: 4 per tutti i nodi
- **Diametro**: $2 \cdot \left\lfloor \frac{r}{2} \right\rfloor$
- **Numero totale di link**: 2*n
- **Vantaggi**: scalabile anche se non in modo semplice come la mesh bidimensionale.
- **Tolleranza ai guasti**: molto alta.

4.7.3 Calcolo dello *speed-up*

Un modo semplice per analizzare l'efficienza di una rete è il calcolo dello speed-up, ossia il guadagno di velocità che ho nel passare da una macchina sequenziale a una parallela.

Prendo un programma, implementato in modo sequenziale, e lo faccio girare su una macchina sequenziale SISD. Prendo lo stesso programma e lo faccio girare su una macchina parallela di nodi N. Lo speed-up consiste nel seguente rapporto:

$$S = \frac{T1}{TN}$$

Lo speed-up ideale è lineare: $S = N$. Purtroppo ciò non avviene: si pensi all'*overhead*, se prendo la SIMD ho i due switch che provocano rallentamento (in particolare lo switch NXN). Le macchine SIMD vanno molto vicino a uno speed-up uguale ad N:

- gli algoritmi sono pensati per un'esecuzione in parallelo;
- le applicazioni per macchine SIMD richiedono poco I/O (al più prendono dati all'inizio).

Al contrario, nelle macchine MIMD è veramente complicato avere programmi in grado di permettere a tutti i processori di lavorare costantemente in parallelo (maggiore I/O e maggiore necessità di comunicazione).

4.7.4 Calcolo dell'efficienza

Se divido lo speed-up per il numero di processori N ottengo l'efficienza

$$E = \frac{S}{N}$$

In un mondo ideale abbiamo $E = 1$, in realtà avremo sempre $E < 1$

4.7.5 Legge di Amdahl

La legge afferma che il parallelismo perfetto non è raggiungibile: ci si può avvicinare ad N ma si ha un limite invalicabile. Ogni programma, per sua natura, non sarà mai totalmente parallelizzabile: per una parte di istruzioni mi basta un solo processore, non traggo vantaggi dall'averne più di uno.

Definisco $Tseq$ come il tempo relativo alle istruzioni che non si parallelizzano e vado a fare il seguente rapporto:

$$S = \frac{T1}{Tseq + \left[\frac{T1 - Tseq}{N}\right]}$$

Se io faccio il limite con $N \to \infty$ ottengo

$$S = \frac{T1}{Tseq}$$

Abbiamo trovato il limite.

5 Gestione dei processi

5.1 Concetto di processo

5.1.1 Definizione di processo, differenza tra programma e processo

Il programma consiste in una rappresentazione del procedimento logico che deve essere eseguito per risolvere un determinato problema. Si ricorre a un particolare linguaggio di programmazione che rende possibile l'esecuzione da parte del calcolatore. Il calcolatore che esegue il programma produce una sequenza di eventi, influenzata dai dati posti in ingresso: input diverso comporta esecuzione diversa (si pensi agli if, la validità delle condizioni può dipendere dall'input).

Il processo rappresenta la sequenza di eventi generati dall'elaboratore durante l'esecuzione del programma. Questi eventi possono essere di carattere macroscopico (si pensi alle periferiche, quando scriviamo sulla tastiera), ma anche microscopico (pensiamo ai cambi di strutture dati che noi non vediamo).

Il processo coincide con la cosiddetta *unità di esecuzione* all'interno di un sistema operativo multiprogrammato: un programma e le risorse necessarie per eseguire (dati, processore e memoria in primis).

Rendiamo chiara la differenza tra programma e processo col seguente esempio.

```
{
a=x; b=y;
while (a !=b)
        if (a > b) then a=a-b;
        else b=b-a;
}
```

Programma (statico)

esecuzione del programma con valori iniziali: x=18, y=24

	Stato iniziale					Stato finale
x	18	18	18	18	18	18
y	24	24	24	24	24	24
a	-	18	18	18	12	6
b	-	-	24	6	6	6

Storia del processo
(Evoluzione dello stato)

Rappresentiamo il processo attraverso l'evoluzione del suo stato: il fatto che io possa rappresentare lo stato in un certo istante mi permette di sospendere il processo e riprenderlo successivamente.

Istanze di un programma. Dato un programma, posso avere più processi distinti che si generano a partire dall'esecuzione dello stesso programma. La differenza principale sta nell'input (a Lettieri viene l'orticaria se sente dire così, ma nel libro di Ancilotti si parla letteralmente di un insieme di processi come istanze del programma)!

5.1.2 Rappresentazione del processo nel calcolatore

Il processo di per sé è un concetto astratto, ma noi siamo ingegneri: a noi interessano cose concrete. Come si passa dal concetto astratto a un qualcosa di elaborabile in un calcolatore?

- Codice (tipicamente sta in un file, ma potrebbe stare anche in una memoria flash…)
- Dati (modificabili, quindi immagino operazioni di lettura e scrittura)
- Registri della CPU (che rappresentano lo stato, si pensi in particolare al *Program Counter*)
- Lo stack (pila, anche il contenuto della pila è parte del processo)

A un processo possono essere assegnate risorse fisiche e/o logiche:

- CPU,
- Memoria,
- file aperti (non tutti i file, solo quelli che interessano al programma), e
- dispositivi di I/O.

5.1.3 Stati di un processo

Quali stati può assumere un processo?

5.1.3.1 Sistema operativo monoprogrammato

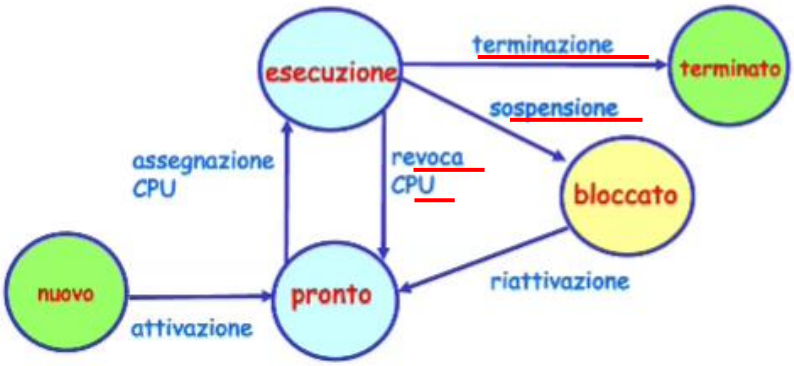
Non esistono una molteplicità di processi, ne abbiamo uno solo.

- **Attivo**: il processo è stato creato (le strutture dati sono state allocate, è stata allocata la memoria appena introdotta – questo significa che il sistema operativo ha calcolato la dimensione dello spazio e ha individuato l'esistenza di una partizione abbastanza grande per contenere questo spazio virtuale) e l'esecuzione è iniziata.
- **Bloccato**: il processo è sospeso, ha lanciato un'operazione di I/O (attende il compimento del trasferimento) e non può fare altro finchè l'operazione non risulta completata.

5.1.3.2 Sistema operativo multiprogrammato

In un sistema multiprogrammato le cose diventano più complesse, con l'introduzione di una molteplicità di processi:

- **Nuovo**: processo appena creato (bisogna ancora allocare le strutture dati necessarie, sistemare la memoria, inizializzare tutto ciò che è necessario…)
- **Pronto**: i processi creati e posti in memoria. Non stanno in esecuzione perché non hanno assegnata la CPU. La transizione da *pronto* a *esecuzione* avviene quando la CPU viene assegnata al processo. L'insieme dei processi pronti è rappresentato da una coda.
- **Bloccato**: uguale a prima. Ci si pone in questo stato solo da esecuzione, visto che un processo si blocca solo dopo aver lanciato un'operazione di I/O. Altra cosa è che non si passa da *bloccato* a esecuzione, ma solo da bloccato a pronto. Anche questo insieme può essere rappresentato da code.
- **Esecuzione**: un solo processo alla volta se il sistema è monoprocessore, n processi se l'architettura ha n processori.
- **Terminato**: processo terminato.



Si osservi che con gli eventi (archi) che si possono manifestare nello stato *esecuzione* la CPU si libera. Dobbiamo metterci in testa che in presenza di queste transizioni lo *scheduler* va sempre in esecuzione (bisogna scegliere a quale processo in coda *pronti* assegnare la CPU).

5.1.4 Descrittore di processo

Il descrittore di processo è una struttura dati associata ad ogni processo, che vari campi:

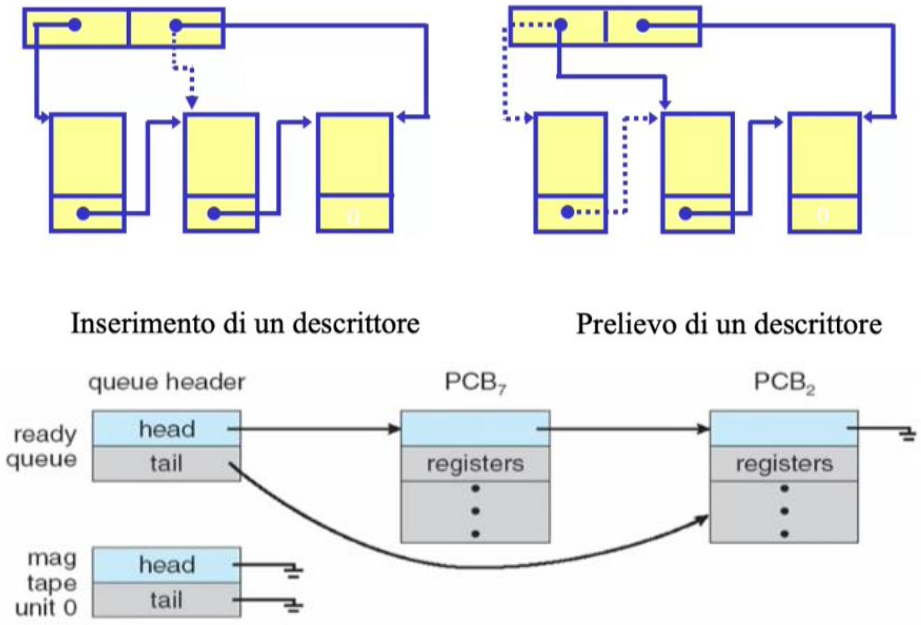
- **Nome del processo**
  Un identificatore univoco del processo, solitamente un indice (si pensi al pid – *process id* – nel caso di UNIX)
- **Stato del processo**
  Lo stato del processo (uno tra quelli introdotti poco fa)
- **Modalità di servizio dei processi**
  Informazioni utilizzate dall'algoritmo di scheduling. Variano in base agli algoritmi di scheduling adottati. Esempio: livello di priorità associato al processo, rappresentabile in vari modi (un semplice indice, una deadline nel caso di sistemi in tempo reale, oppure il *quanto* di tempo dedicabile nel caso di sistemi time-sharing).

- **Informazioni sulla gestione di memoria**
  Informazioni su come raggiungere l'area di memoria principale relativa al processo.
- **Contesto del processo**
  Già visto a Calcolatori elettronici, letteralmente un array contenente il valore di tutti i registri (una fotografia del processo, memorizzata ogni volta che si ha un cambio di contesto e il processo perde controllo della CPU).
- **Utilizzo delle risorse**
  Informazioni riguardo le risorse fisiche e logiche assegnate al processo (periferiche, file aperti, tempo di uso della CPU…)
- **Identificazione del processo successivo**
  Già visto a Calcolatori elettronici, puntatore con cui il descrittore può essere collocato all'interno di code.

Esiste in memoria una tabella dei processi che contiene tutti i descrittori di processo. È privilegiata: vi accede solo il sistema operativo. Tanti dettagli risulteranno più chiari nei capitoli successivi (introdurremo anche il descrittore presente in UNIX).

5.1.4.1 Algoritmi per la manipolazione delle code

Il sistema operativo gestisce svariate code: non solo la coda *pronti*, ma anche quelle relative alla sospensione dei processi (quando viene sospeso un processo per un'operazione su una periferica I/O, ad esempio, questo verrà posto in una coda appositamente dedicata per quella periferica – e rimangono finchè non si manifesterà un particolare evento). Le code vengono manipolate ricorrendo ai classici algoritmi visti a Fondamenti di programmazione:



L'algoritmo di scheduling, usando queste code, sceglie a quale processo assegnare la CPU.

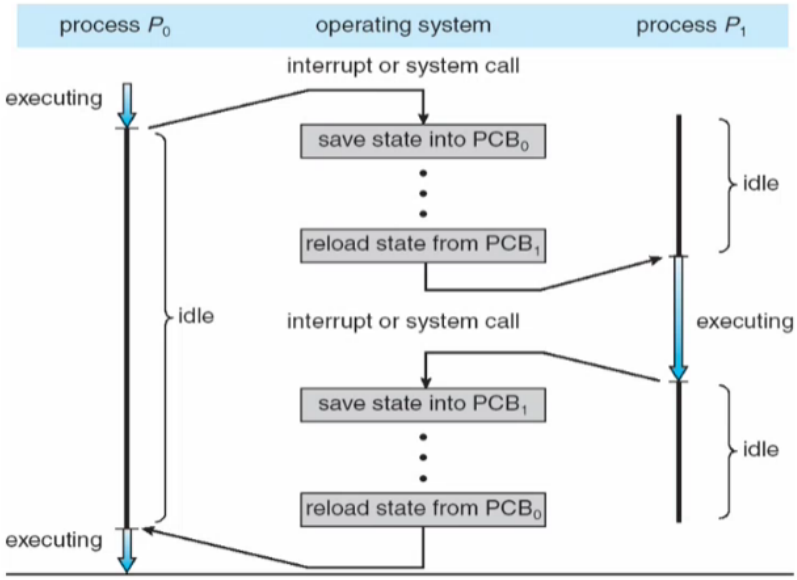
5.1.5 Cambio di contesto

Il cambio di contesto è un meccanismo che appartiene al nucleo di un sistema operativo multiprogrammato. Ricordiamo che il contesto è l'insieme dei contenuti dei registri della CPU.

1. **Salvataggio stato**.
   Ogni volta che la CPU viene assegnata ad un altro processo dobbiamo salvare i valori uscenti per poterli utilizzare in futuro (si fanno una serie di copie da processore a memoria per salvare lo stato).
2. **Inserimento coda**.
   Quando abbiamo salvato lo stato del processo il relativo descrittore deve essere messo nella coda giusta.
3. ***short term scheduling [Non è parte dello schheduling, diciamo solo per completezza]***.
   Dobbiamo selezionare un altro processo dalla coda dei processi pronti. Questo passo, precisamente la politica adottata, non fa parte del cambio di contesto (lo diciamo solo per completezza): il cambio di contesto è indipendente dalla politica di scheduling. Si chiama un'altra procedura che implementa lo scheduling, e si pone nel registro processo in esecuzione l'identificativo del processore. Le politiche di scheduling devono poter essere modificabili (il fatto che si abbia una procedura dedicata non è casuale).

4. **Ripristino stato**.
   Ripristino dello stato del processo scelto (che ricordiamo è salvato nell'array *contesto* del descrittore del processo scelto dall'algoritmo di scheduling).

Vediamo dalla figura cosa succede nel cambio di contesto



- Nella parte sinistra della figura si ha un processo P0 inizialmente in esecuzione
- A un certo punto il sistema operativo impone il cambio di contesto (non ci interessa la motivazione).
  - Viene lanciata un'interruzione interna (una system call).
  - Si memorizza lo stato nel descrittore PCB0.
  - Si ripristina lo stato copiandolo dal descrittore PCB1.

  Siamo passati al processo P1. Ricordarsi che nel contesto è incluso anche il *Program counter*, quindi abbiamo anche l'indirizzo del processo P1 laddove era stato sospeso.
- P1 viene eseguito per un certo tempo. A un certo punto vogliamo ritornare sul processo P0: facciamo le stesse cose di prima (salvataggio in PCB1 e ripristino da PCB0).

Attenzione all'overhead: servono cicli macchina per fare tutto questo!

Il cambio di contesto richiede onerosi trasferimenti. Sarà necessario anche riassegnare parti della memoria principale (vedremo tecniche di allocazione della memoria che minimizzano questo *overhead*). Dobbiamo tenere conto anche del contenuto della cache, che deve essere invalidato.

5.1.6 Processi padri e processi figli, creazione e terminazione

Nei sistemi operativi, in generale, esiste sempre un processo "padre" che richiede la creazione di un processo "figlio". Si ha una gerarchia utile nel trasferimento di dati (passaggio di strutture dati).
Ogni processo è figlio di un altro processo, e un processo può essere a sua volta padre di altri processi.

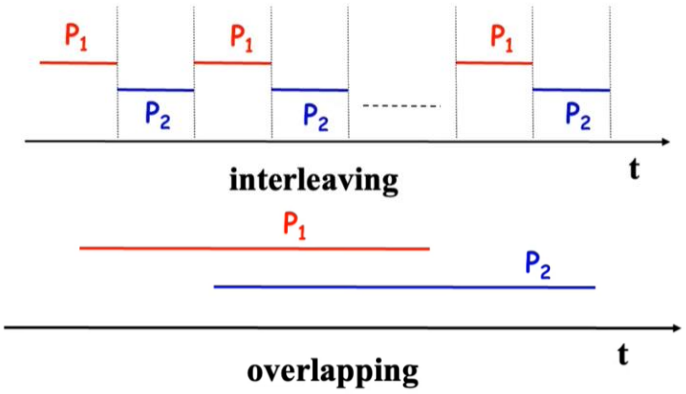
Terminazione di un processo figlio e/o di un processo padre. Quando un processo termina è necessario che padre e figli siano a conoscenza. Il padre di un processo che termina, in UNIX, se ne può accorgere. Abbastanza scontato che con la terminazione di un processo termineranno anche i processi figli.

5.1.7 Definizione di processo concorrente e differenza con i processi paralleli

In un sistema multiprogrammato abbiamo un'esecuzione di tipo concorrente.

- Due o più processi si dicono concorrenti se le loro esecuzioni si sovrappongono nel tempo.
- Attenzione a non fraintendere la cosa: concorrente è diverso da parallelismo, l'unico modo per avere parallelismo è avere più di un processore. Possiamo dire che due processi sono concorrenti se la prima operazione di uno comincia prima che termini l'ultima dell'altro.

In presenza di un solo processore abbiamo l'*interleaving* (letteralmente *interfogliamento*, i processi condividono la stessa unità di elaborazione), mentre con più processori abbiamo l'*overlapping* (letteralmente *sovrapposizione*, i due processi hanno ciascuno un'unità di elaborazione).



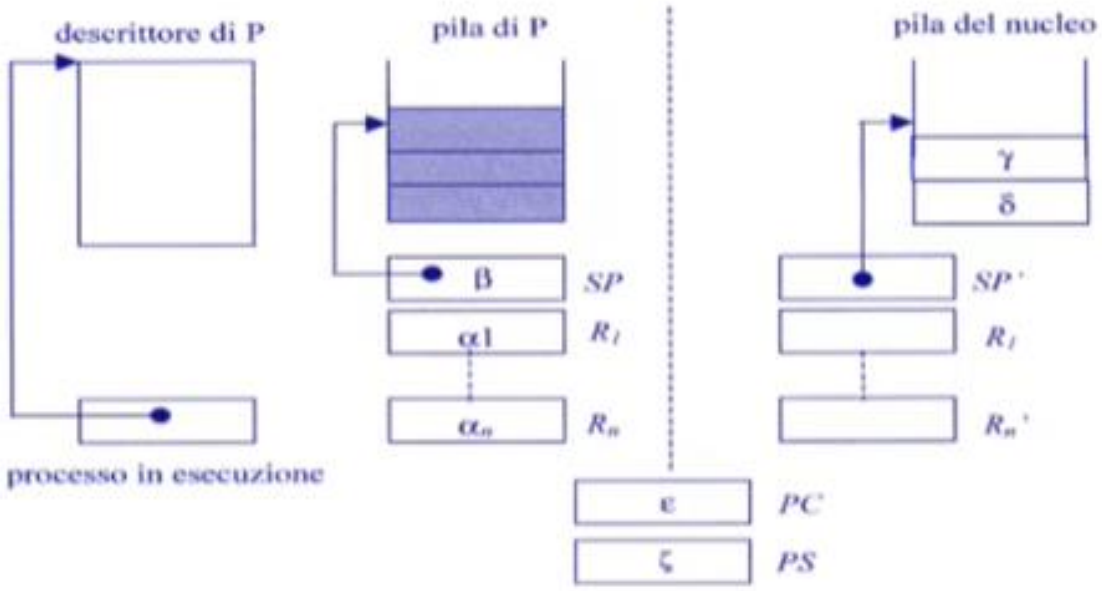
5.1.8 Processi indipendenti e processi interagenti, proprietà della riducibilità

- **Processi indipendenti**
  Processi concorrenti sono indipendenti se l'esecuzione di uno non è influenzata dall'esecuzione dell'altro, e viceversa. Un insieme di processi indipendenti gode della proprietà della riducibilità.
- **Processi interagenti**
  Due processi sono interagenti se l'esecuzione di uno è influenzata dall'esecuzione dell'altro, e viceversa. L'effetto dell'interazione dipende dalla velocità dei processi. Un insieme di processi interagenti non gode della proprietà della riproducibilità (ogni esecuzione è diversa, ciò non è sbagliato).

Proprietà della riducibilità. Un insieme di processi gode della proprietà della riducibilità se l'evoluzione temporale dei processi è sempre la stessa, cioè se il risultato dall'esecuzione dei processi è sempre lo stesso. La riproducibilità è una proprietà possibile solo con un insieme di processi indipendenti e in presenza di un algoritmo di scheduling deterministico (cioè prevedibile).

5.1.9 Riduzione dell'overhead con replicazione di pila e registri (riduzione overhead)

Alcune architetture adottano la seguente struttura per ridurre l'overhead.



- La CPU prevede la replica di alcuni registri: a destra abbiamo la "pila del nucleo", struttura dati dedicati esclusivamente dedicata al nucleo, uno stackpointer e una serie di registri operativi dedicati esclusivamente al nucleo.
- A sinistra abbiamo uno stackpointer e registri operativi validi per i processi applicativi. Chiaramente abbiamo una pila in memoria diversa da quella del nucleo.
- Nella CPU deve esistere un registro che contiene il puntatore al descrittore del processo in esecuzione.
- Si hanno in comune due registri: PC e PS (abbastanza ovvio, il processore è unico).
- **Perché la situazione migliora?** Perché non ho da fare il cambio di contesto: semplicemente cambio i registri utilizzati, dunque l'overhead si riduce perché non ho da fare trasferimenti da memoria a registri e viceversa.
- **Attenzione:** le cose dette riguardano le chiamate di system call, non il cambio di contesto da un processo a un altro (In quel caso è inevitabile fare dei trasferimenti).

5.2 Introduzione allo scheduling

5.2.1 Definizione di scheduling e tipologie di scheduling

Lo scheduling (pianificazione) sono le attività mediante le quali il sistema operativo effettua delle scelte:

- quali processi caricare in memoria centrale (*long term scheduling*);
- quali processi spostare dalla memoria centrale a quella secondaria (*medium term scheduling*);
- a quale processo assegnare la CPU (*short term scheduling*).

Abbiamo tre tipi di scheduling: a breve, a medio e a lungo termine. L'unico scheduling che ci interessa in questo capitolo è il primo, gli altri saranno visti nei capitoli successivi. Definiamo tutte e tre le tipologie per costruire un quadro completo fin da subito.

5.2.1.1 Scheduling a breve termine

Lo scheduling a breve termine sceglie tra i processi nella coda *pronti* quello a cui assegnare la CPU. Interviene quando il processo in esecuzione perde il controllo della CPU. Si distinguono due scheduling:

- ***Non preemptive scheduling***: senza diritto di revoca, lo scheduler a breve termine non può revocare la CPU a un processo dopo avergliela precedentemente assegnata. Si aspetta che succeda qualcosa (termina il processo, lancia un'operazione di I/O, si manifestano certe tipologie di errori).
- ***Preemptive scheduling***: con diritto di revoca.

5.2.1.2 Scheduling a medio termine (swapping)

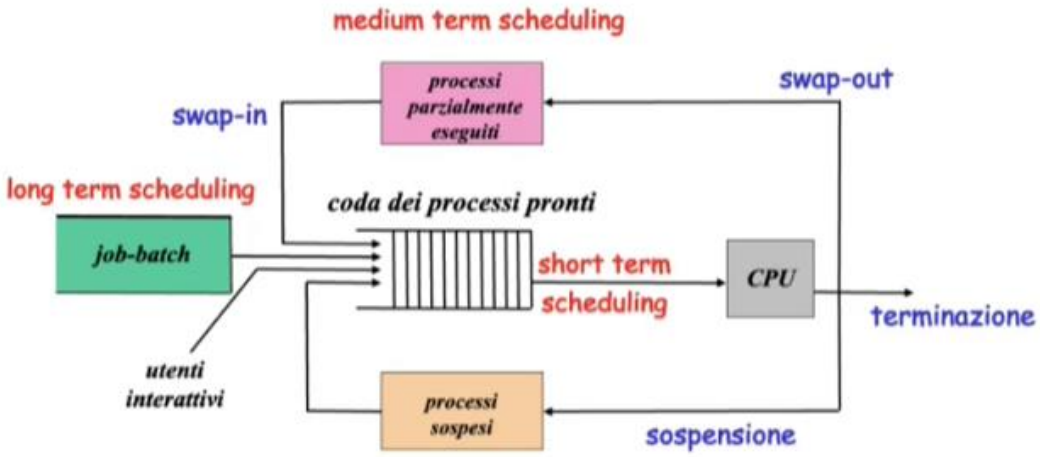
Lo scheduling a medio termine sceglie nella memoria centrale quali processi trasferire temporaneamente nella memoria secondaria.

- Si fa riferimento al meccanismo di swapping (che, ricordiamo, consiste in un'area del disco persistente – area di swap – utilizzata per copiare in ingresso e in uscita il contenuto della memoria principale).
- Posso trasferire temporaneamente processi in memoria secondaria.
- *Medio termine* perché dura un po' di più dello scheduling precedente, ma si manifesta con maggiore rarità.

5.2.1.3 Scheduling a lungo termine

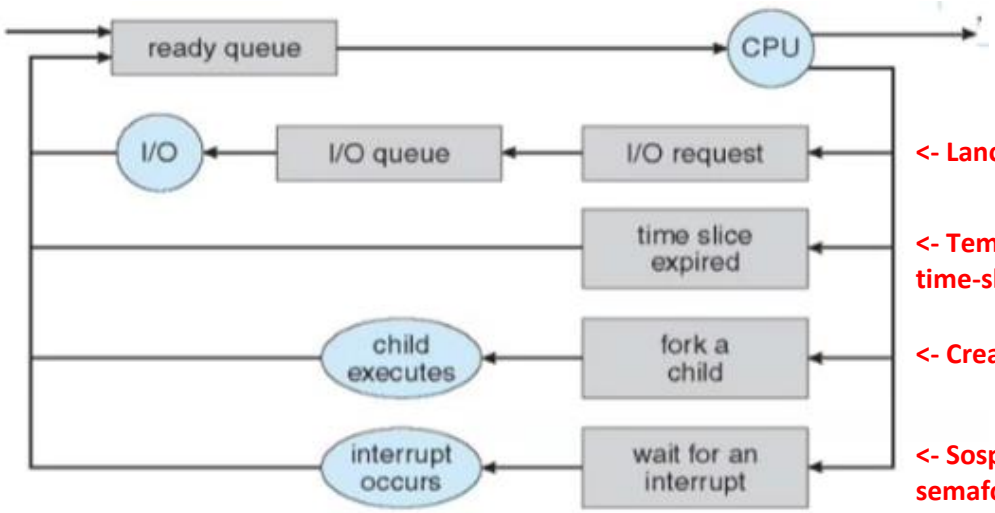
Lo scheduling a lungo termine sceglie nella memoria secondaria quali *task*/*processi* caricare in memoria centrale. L'algoritmo di scheduling pensato deve garantire un equilibrio tra processi CPU-bound e processi I/O bound, in modo tale da portare l'efficienza della CPU vicina al 100%.

5.2.2 Recap



- Con il ***long term scheduling*** otteniamo un batch di job posto in memoria centrale. Per ogni job si inizializza un processo e relativo descrittore, ciascuno posto in coda *pronti*.
- A un certo punto interviene lo ***short term scheduling***, che decide a quale processo in coda pronti assegnare la CPU.
- **Strade possibili**:
  - il processo termina e non ritorna nella coda pronti;
  - il processo viene sospeso e finisce in qualche coda di processi bloccati, quando si riattiva ritorna nella coda pronti (la cosa è fuori dallo scheduling in quanto dipendente da eventi esterni);
  - la memoria non basta più e il sistema operativo fa *swap-out*, cioè sceglie un processo vittima (medium term scheduling) e lo priva della memoria precedentemente assegnata (lo spazio relativo al processo finisce nell'area di swap, è al sicuro ma non può essere usato finché non ritorna in memoria principale con lo *swap-in* – anche questi processi sono posti in una coda).

5.2.3 Recap con punto di vista diverso



<- Lancio di una istruzione I/O

<- Tempo scaduto in un sistema time-sharing

<- Creazione di un processo figlio

<- Sospensione su primitiva semaforica wait

- Lo schema ribadisce le cose di prima ponendole da un punto di vista diverso.
- Il processo va in esecuzione solo dopo essere passato dalla coda dei processi pronti (*ready queue*).
- Il processo rilascia la CPU per una serie di motivazioni:
    - Operazione di I/O (si sospende e finisce in coda I/O queue, arriva l'evento esterno che segnala il completamento dell'operazione e il processo torna in coda pronti);
    - in un sistema di time sharing il tempo allocato per il processo scade (*time slice expired*, cosa possibile solo con un preemptive scheduling);
    - viene creato un processo figlio con la primitiva *fork*, il processo padre si sospende e permette l'esecuzione del progetto figlio (possibile, ma non succede sempre);
    - necessaria sincronizzazione, il processo esegue la primitiva semaforica wait che potrebbe portarlo alla perdita della CPU (per esempio nell'accesso a una risorsa).

5.3 Algoritmi per short-term scheduling

5.3.1 Criteri per la valutazione degli algoritmi

L'algoritmo di scheduling si basa su dei criteri. Tra i più importanti abbiamo:

Criterio	Descrizione
Uso della CPU	Occupazione della CPU da parte di un processo.
Turnaround time *(o tempo medio di completamento)*	Tempo che passa dall'ingresso del processo in coda pronti al suo completamento (non necessariamente terminazione, ma restituzione di un risultato). Il tempo medio dipende dall'algoritmo di scheduling adottato, ma anche dalla complessità dell'algoritmo implementato (il task).
Throughput rate *(o produttività)*	Numero di processi completati nell'unità di tempo. Prendo l'intervallo di osservazione, conto i processi che terminano dall'istante 0 dell'intervallo in poi e divido per il tempo.
Tempo di risposta	Tempo che va dall'istante di richiesta a quando si fornisce il risultato.
Tempo di attesa	Somma degli intervalli di tempo in cui il processo attende in coda pronti.
Rispetto dei vincoli temporali	Garantire il rispetto dei vincoli posti in sistemi in tempo reale.

5.3.2 Introduzione agli algoritmi

Gli algoritmi per short-term scheduling individuano il processo a cui assegnare il controllo della CPU. Quelli che ci interessano sono i seguenti:

- **Algoritmi senza *preemption***:
    - ***First-come-first-served (FCFS)***
      Si sceglie il processo arrivato per primo (letteralmente una coda FIFO) servo per primo il processo arrivato per primo (letteralmente una coda FIFO)

- ***Shortest-Job First (SJF)***:
  Si sceglie il processo che richiede meno tempo (tempo misurato in cicli di CPU). Non si guarda all'ordine di arrivo.

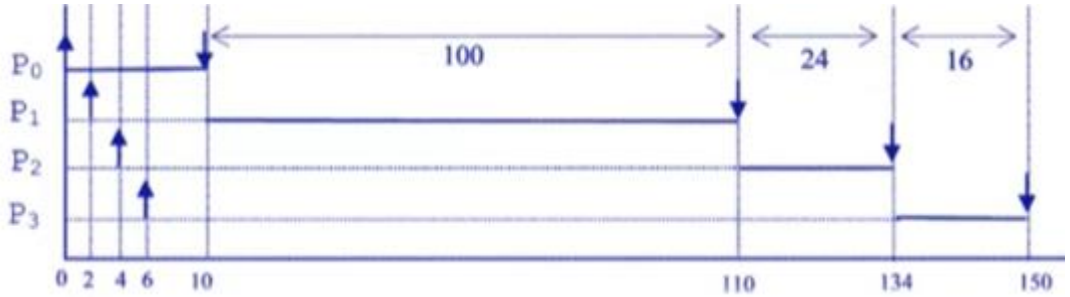
- **Algoritmi con *preemption***:
  - ***Shortest-Remaining-Time-First (SRTF)***:
    Si sceglie il processo col tempo di esecuzione rimanente minore.
  - ***Round-Robin (RR)***:
    Algoritmo di scheduling classico per i sistemi time-sharing. Insieme di FCFS trasformato in algoritmo preemptive, la preemption dipende dal tempo passato: si pone nei descrittori un tempo uguale per tutti i processi, col lancio dell'interruzione da parte del timer si lancia l'algoritmo Round-Robin e si fa cambio di contesto se necessario.

5.3.3 Algoritmi senza preemption

5.3.3.1 Algoritmo First-Come-First-Served (FCFS)

Supponiamo di avere quattro processi: P_0, P_1, P_2, P_3. Per ogni processo devo conoscere l'istante di arrivo in coda pronti. Devo anche conoscere la durata del CPU burst (il tempo necessario per il completamento del processo, si parla di unità di tempo in senso neutro).

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	10
P_1	2	100
P_2	4	24
P_3	6	16



Ai fini dell'algoritmo non è necessario conoscere a priori il CPU burst: lo teniamo solo a fini simulativi, ci serve solo per capire quanto dura il processo. Il CPU burst permette anche di classificare i vari processi:

- P_0 è un processo I/O bound, ha un burst molto basso (quindi si suppone che il processo si sospenda subito)
- P_1 è un processo CPU bound, ha un burst decisamente maggiore, dunque svolge un numero decisamente maggiore di istruzioni non I/O.

Vediamo gli step simulativi:

- Inizia il processo P0, mentre questo viene eseguito arrivano in coda pronti i processi P1, P2 e P3.
- P0 continua finché non finisce nell'istante di tempo 10. A questo punto dobbiamo passare a un altro processo. Si vuole prendere il primo arrivato tra i rimanenti, quindi si comincia P1.
- P1 fa tutto quello che deve fare e finisce dopo 100 unità di tempo. Si guarda chi è arrivato dopo P1 e lo mettiamo in esecuzione: P2!
- Dopo 24 unità di tempo finisce P2. Iniziamo ad eseguire P3, l'unico rimasto e l'ultimo arrivato in coda pronti.
- Dopo 150 unità di tempo si conclude la simulazione e la coda pronti è stata svuotata.

Facciamo le nostre conclusioni basandoci su alcuni dei criteri elencati:

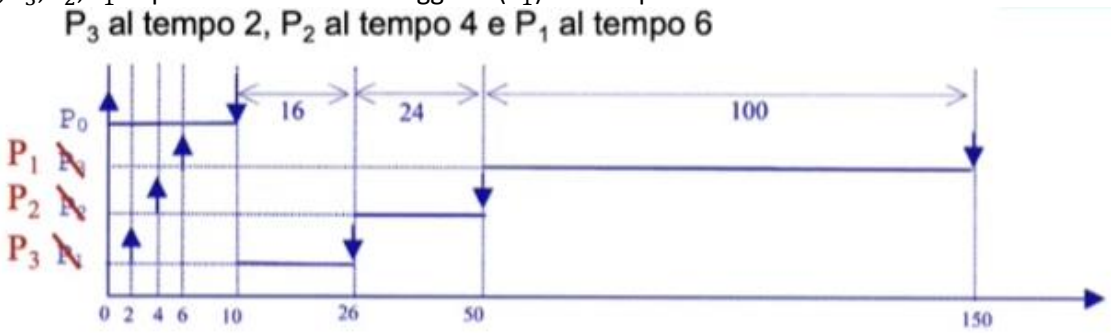
- **Calcolo del turnaround medio**
  Otteniamo il turnaround medio facendo la media dei turnaround dei singoli processi. Ricordiamoci che il turnaround consiste nel tempo che va dall'arrivo in coda pronti al completamento

  $$\text{Turnaround medio} = \frac{P_0 + P_1 + P_2 + P_3}{4} = \frac{(10-0) + (110-2) + (134-4) + (150-6)}{4} = 98$$

  Ottengo un turnaround medio di 98. Non ho fatto un buon servizio ai processi veloci (che hanno un burst di 10, 16 e 24).

- **Calcolo del tempo medio di attesa**
  Otteniamo il tempo medio di attesa facendo la media tra i tempi di attesa dei singoli processi. Ricordiamoci che il tempo di attesa consiste nel tempo che un processo attende in coda pronti.
  $$\text{Tempo medio di attesa} = \frac{P_0 + P_1 + P_2 + P_3}{4} = \frac{0 + (10-2) + (110-4) + (134-6)}{4} = 60.5$$
  Ottengo un tempo medio di attesa di 60.5. Anche qua non è andata bene per i processi brevi. Anche qua un ordine diverso degli elementi avrebbe fatto la differenza (anche qua ponendo P1 in fondo). L'assenza di *preemption* mi impedisce di dare garanzia sui processi brevi.

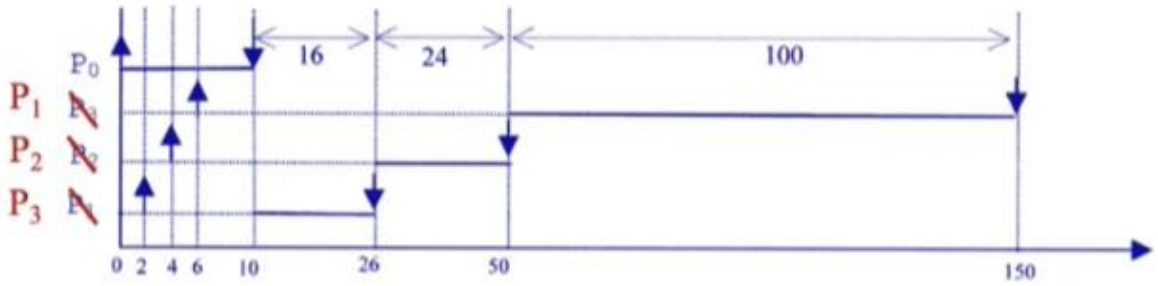
Proviamo a fare una nuova simulazione alterando l'ordine di arrivo dei processi: supponiamo di avere, nell'ordine, P_0, P_3, P_2, P_1. Il processo di durata maggiore (P_1) è stato posto in fondo.



- **Il turnaround medio** adesso è di 56
  $$\text{Turnaround medio} = \frac{P_0 + P_3 + P_2 + P_1}{4} = \frac{10 + (26-2) + (50-4) + (150-6)}{4} = 56$$
  non è migliorato molto, ma sicuramente meglio di prima.
- **Il tempo medio di attesa** è molto più basso: 18.5
  $$\text{Tempo medio di attesa} = \frac{P_0 + P_3 + P_2 + P_1}{4} = \frac{0 + (10-2) + (26-4) + (50-6)}{4} = 18.5$$
  Non siamo soddisfatti perché ci è andata bene in modo **<u>del tutto casuale</u>**.

5.3.3.2 Algoritmo Shortest-Job-First (SJF)

L'algoritmo Shortest-Job-First (SJF) pone come criterio il CPU burst dei vari processi. Contrariamente a prima il CPU burst non ha solo finalità simulative, ma è proprio il criterio che determina la scelta di un particolare processo. L'ordine di arrivo non è minimamente considerato (si ottiene lo stesso risultato nella simulazione con qualunque ordine di arrivo).



- Il turnaround medio e il tempo medio sono gli stessi visti nella simulazione precedente (quella dove abbiamo posto P1 in fondo).
  $$\text{Turnaround medio} = \frac{P_0 + P_3 + P_2 + P_1}{4} = \frac{10 + (26-2) + (50-4) + (150-6)}{4} = 56$$
  $$\text{Tempo medio di attesa} = \frac{P_0 + P_3 + P_2 + P_1}{4} = \frac{0 + (10-2) + (26-4) + (50-6)}{4} = 18.5$$
- L'algoritmo è più complesso: è necessario che nel descrittore del processo sia presente il CPU burst (che deve essere conosciuto a priori quando viene eseguito l'algoritmo).
- L'overhead è maggiore: dobbiamo scorrere tutti i CPU burst per decidere a chi assegnare la CPU. La cosa non è un problema: l'algoritmo non viene seguito troppo frequentemente (non c'è la preemption).
- I processi potrebbero essere vittima di *starvation*: processi lunghi non vanno mai in esecuzione perché entrano sempre processi più brevi (nel caso in cui la coda venga manipolata con rate elevato, maggiore è il *rate* più è probabile che i nuovi processi siano di breve durata).

5.3.4 Stima del CPU burst

Non sempre il valore del CPU burst è noto a priori. Quello che facciamo è una stima ricorrendo alla media esponenziale dei valori misurati nei precedenti intervalli di esecuzione. Dato t_n la durata del CPU-burst n-esimo e la sua stima s_n otteniamo la stima $n+1$-esima

$$s_{n+1} = a \cdot t_n + (1-a)s_n$$

a è un coefficiente compreso tra 0 e 1. Nei casi stremi abbiamo:

- stime uguali a quella iniziale se $a = 0$;
- stime prive di considerazione della storia con $a = 1$.

Tipicamente si pone $a = \frac{1}{2}$. Si pone un bilanciamento tra previsione futura e ciò che è effettivamente successo.

In un certo senso faccio in modo che il valore più vecchio pesi sempre meno nella stima:

$$s_{n+1} = a \cdot t_n + (1-a)a \cdot t_{n-1} + \cdots + (1-a)^i a \cdot t_{n-1} + \cdots + (1-a)^{n+1} s_0$$

5.3.5 Algoritmi con preemption

Gli algoritmi a priorità statica e senza preemption visti precedentemente presentano dei limiti:

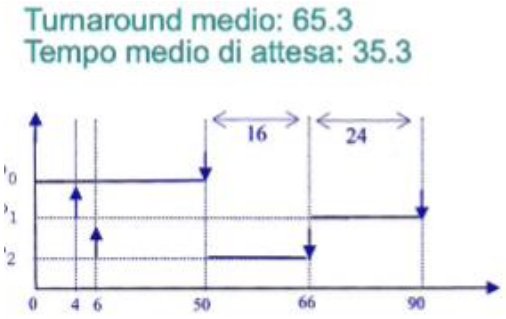
- First-Come-First-Served ha turnaround medio e tempo di attesa molto elevati (non è l'algoritmo ideale in presenza di processi con CPU burst basso)
- Shortest-Job-First presenta dei miglioramenti nei due dati, soprattutto nel tempo medio di attesa, tuttavia emerge il problema della starvation.

Per cercare di superare questi problemi introduciamo algoritmi con preemption.

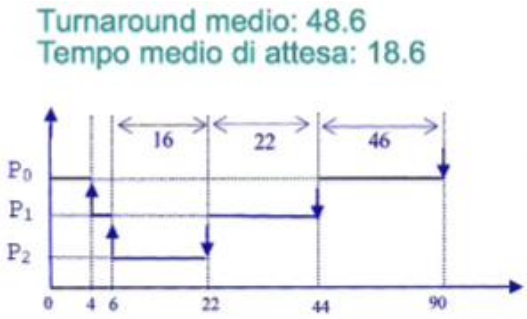
5.3.5.1 Algoritmo Shortest-Remaining-Time-First (SRTF)

L'algoritmo adotta una priorità simile allo *Shortest-Job-First*: si considera la durata del CPU burst come prima, ma l'accesso di un processo alla CPU viene revocato qualora entri in coda *pronti* un processo con CPU burst minore. Abbiamo tre processi (P0, P1, P2) di cui si conosce l'istante di arrivo in coda pronti e la durata del CPU burst. La priorità è dinamica: viene ricalcolata ogni volta (limitatamente al solo processo in esecuzione) in modo da considerare il tempo rimanente.

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	50
P_1	4	24
P_2	6	16



SJF (Fig. 2.15)



SRTF (Fig. 2.16)

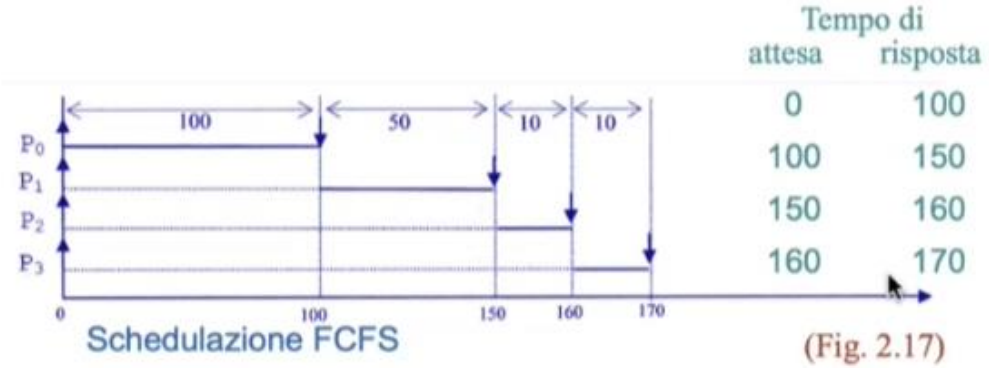
- **Quando va in esecuzione l'algoritmo?** Quando un nuovo processo entra nella coda pronti. L'algoritmo può essere triggerato da più eventi rispetto agli altri algoritmi già visti. Le interruzioni interne permettono di togliere il controllo della CPU al processo, se necessario.
- **Simulazione e confronto tra SJF ed SRTF**:
  - All'istante 0 abbiamo un solo processo in coda pronti. Va in esecuzione P0, con CPU burst 50.
  - L'algoritmo SJF, non preempted, lascia liberi i processi finchè questi non vengono completati.
  - L'algoritmo SRTF, invece, entra in scena non appena vengono posti i nuovi processi in coda. Durante l'esecuzione di P0 si passa a P1 non appena questo viene introdotto in coda pronti: questo perché il CPU burst è minore (24 vs 50).
  - Durante l'esecuzione di P1 si passa a P2 per le stesse motivazioni di prima: il CPU burst è minore (15 vs 24). Al termine dell'esecuzione di P2 si torna ad eseguire P1, che a confronto con P0 (gli unici processi rimasti) è quello con CPU burst minore (24 vs 50).

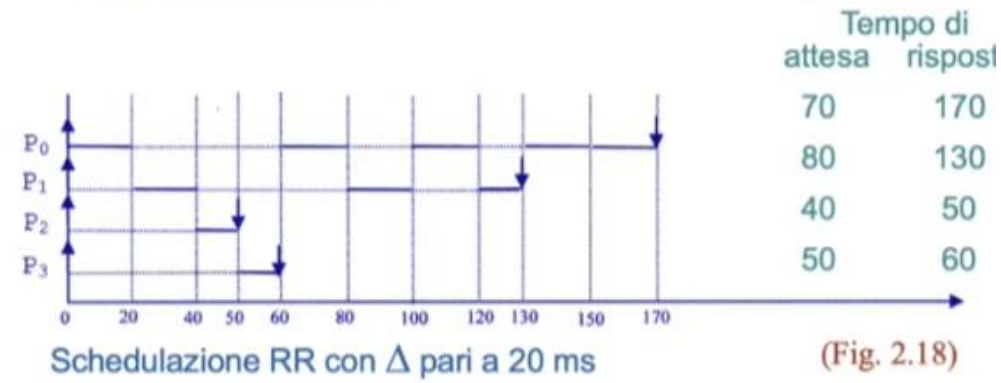
- Nella figura si vede che entrambe le simulazioni terminano all'istante 90. Nella realtà questa cosa non è possibile: questo perché l'overhead di SRTF è sicuramente maggiore dell'overhead di SJF (basta vedere il numero di cambi di contesto effettuati).
- Si osserva che in SRTF sia il turnaround medio che il tempo medio di attesa sono minori. Pare che non ci siano molti dubbi su quale algoritmo sia migliore.
- STRF è affetto da *starvation* per le stesse motivazioni che vedono la *starvation* in SJF. Il fatto è che la priorità non può essere modificata in un processo se questo non può essere portato avanti.

5.3.5.2 Algoritmo Round-Robin (RR)

L'algoritmo è pensato per i processi interattivi (in contrasto con i precedenti ottimi per sistemi batch), cioè per quei processi che hanno una forte interazione con l'utente e/o con altri processi. I processi interattivi hanno CPU burst molto brevi e IO burst più lunghi. Ricordarsi che con CPU burst o I/O burst andiamo a rappresentare intervalli di tempo.

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	100
P_1	0	50
P_2	0	10
P_3	0	10





- Algoritmo simile a *First-Count-First-Served*: la coda dei processi pronti viene servita da Round-Robin allo stesso modo. L'overhead è basso, c'è da fare poco (estrarre la testa della coda). **La differenza principale è la preemption**.
- Quando si supera un tempo limite *quanti* (ricordare i discorsi fatti per i sistemi time-sharing) la CPU viene revocata al processo attuale e ceduta ad un altro processo (secondo logica FCFS).
- Necessaria un'interruzione esterna, quella del timer. L'interruzione sarà gestita da un handler che implementa il Round-Robin.
- Nell'esempio di simulazione abbiamo quattro processi (P0,P1,P2,P3), di ciascuno conosciamo il CPU burst (che serve solo per fare la simulazione, non per prendere decisioni). L'unica cosa rilevante è l'ordine di arrivo dei processi in coda pronti.
- **Osservazioni nel confronto tra SJF ed SRTF**:
  - Si suppone che il tempo limite sia 20 ms. Il primo processo eseguirà 20 ms al di la della durata, a quel punto arriva un'interruzione del timer che riporta il processo in coda pronti. P2 termina prima, così come P3. Alla fine P0 avrà il processore tutto per lui, quindi va oltre i 20 ms e finisce.

- Calcoliamo tempi di attesa e di risposta. Se facciamo le medie in FCFS osserviamo che non ci sono correlazioni tra *CPU burst* e *tempo di attesa necessario*: paradossalmente i processi più semplici hanno atteso di più, e ciò non va bene. In Round-Robin avviene l'esatto opposto.
- C'è *starvation*? No, abbiamo FCFS a parte alcune modifiche. Siamo certi che ogni processo prima o poi sarà eseguito (garanzia per i nostri sistemi).
- Cosa succede quando cambia il numero di processi in coda pronti e il tempo limite? Se conosco il numero medio di processi in coda posso dire che il tempo di attesa medio è il seguente rapporto

$$\text{Tempo di attesa medio} = \frac{\text{Numero processi}}{2} \cdot \text{quanti}$$

  Chiaramente si ha proporzionalità diretta tra tempo di attesa e numero processi.
- Se riduco il *quanti* migliora sia il tempo di attesa che di risposta (in particolare il primo), ma aumentano i cambi di contesto e l'overhead inizia ad essere consistente (non più trascurabile). I processi interattivi, che potrebbero terminare subito, vengono fatti girare in coda troppe volte con un tempo troppo basso.
- Riduciamo il quanti a 10

Schedulazione RR con Δ pari a 10 ms (Fig. 2.19)

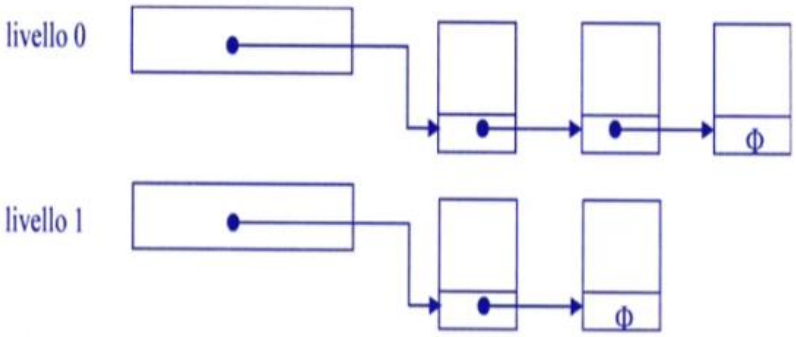
  Tutti i processi iniziano prima: riduco il tempo di attesa a favore dei processi più rapidi. Paga chi ha CPU burst più lungo. I processi si alternano nel tempo con logica FCFS.

5.4 Uso di più algoritmi di scheduling con l'introduzione di più code

Abbiamo introdotto quattro algoritmi, non siamo obbligati ad usarne uno solo. In un sistema operativo multiprogrammato è conveniente una soluzione ibrida.

5.4.1 Schedulazione a code multiple

La *schedulazione a code multiple* introduce più code *pronti*.



- Abbiamo più livelli di code: livello 0, livello 1… Il livello 0 ha priorità più alta rispetto al livello 1.
- A ciascuna coda associo un algoritmo: supponiamo di associare Round Robin alla prima coda, mentre alla seconda associo FCFS.
- Associare una priorità alle code significa servire prima le code con maggiore priorità, e dopo aver svuotato queste le altre.
- Supponiamo che i nuovi processi vengano posti sempre nel livello 0. Emergono i seguenti problemi:
  - *starvation*, le prime code devono essere vuote per considerare le successive, quindi se arrivano nuovi processi non passeremo mai a gestire le code successive;
  - con code di livello superiore associate ad algoritmi senza preemption, ad esempio FCFS, si reintroduce la non preemption e la coda di livello inferiore si riempie nel caso di esecuzione di processi molto lunghi.

  Risolviamo il secondo problema lanciando un'interruzione interna ogni volta che un nuovo processo entra nella coda di livello 0, lanciando così lo scheduler.

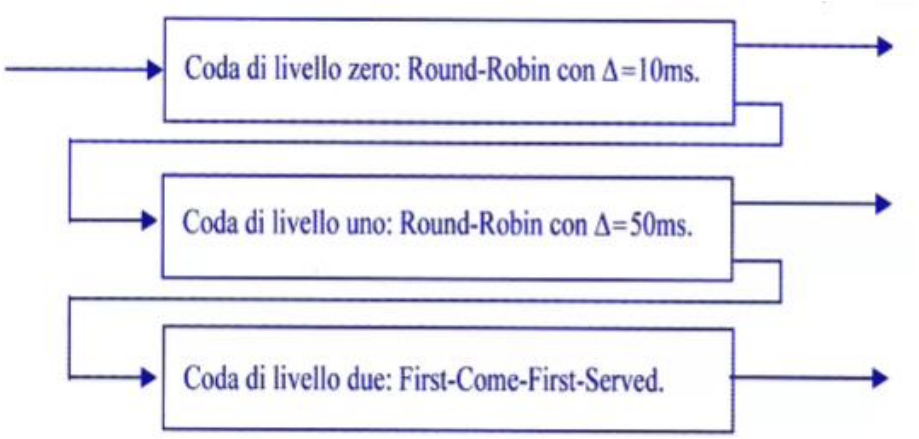
- Un processo lungo ha la CPU e la tiene tutto il tempo, finchè la coda di livello 0 è vuota mi va bene.
- A un certo punto creo un nuovo processo che finisce nella coda di livello 0. L'evento dice al sistema che devo eseguire un nuovo processo: revoco il processo attualmente in esecuzione e inizio ad eseguire quello che avevamo appena inserito in coda pronti.
- Il processo che era in esecuzione che fine fa? Non è né terminato, né sospeso: è pronto, quindi deve andare in una coda pronti. Quale tra le due? La coda di provenienza, ma in testa (quando la CPU sarà nuova libera per la coda di livello 1 dobbiamo riprendere questo processo)!

Un altro meccanismo necessario per evitare la starvation è l'aging, relativo alle code: si spostano processi tra code alterandone la priorità.

5.4.2 Coda multi-level feedback

Una cosa già detta ieri è che al momento della creazione del processo dobbiamo scegliere la coda più giusta. Un approccio potrebbe essere considerare l'interattività e/o la tendenza all'I/O o alla CPU (I/O bound o CPU bound?). A questo punto mettiamo i processi interattivi nella coda di livello 0, mentre quelli CPU-bound nella coda di livello 1. Proviamo a gestire la cosa attraverso una coda multi livello con feedback.

- Abbiamo un sistema di scheduling multilivello, in questo caso tre code (livello zero, livello uno e livello due).
- Quando viene creato un nuovo processo viene sempre messo nella coda di livello 0.
- Nell'eseguire i processi si guardano prima le code con priorità maggiore, e poi quelle con priorità minore.



- **Cosa succede se un processo ha un CPU burst superiore al tempo limite?**
  - Viene revocata la CPU e il descrittore viene posto in una coda di livello inferiore, in questo caso in una coda dove si applica sempre una politica RR, ma con tempo limito maggiore.
  - Se ciò non basta significa che il processo è più complesso di altri, e quindi lo poniamo in una coda di livello due con politica FCFS.

  Si parla di feedback perché è un sistema che si autoregola. Non basta il tempo dato? Ne do di più, ponendolo a priorità minore.
- **Problemi**: i soliti già descritti nella schedulazione a code multiple.

5.5 Schedulazione di sistemi in tempo reale

5.5.1 Introduzione di deadline, campionamenti, processi periodici

Un sistema a tempo reale si basa su un modello con processi, ma va a introdurre vincoli temporali (deadlines) entro i quali i processi devono essere eseguiti: fare diversamente è errore (soprattutto nei sistemi hard-real time). In un sistema in tempo reale le periferiche in ingresso sono sensori, che campionano periodicamente grandezze fisiche (secondo il concetto di tempo percepito dall'essere umano). Dobbiamo decidere:

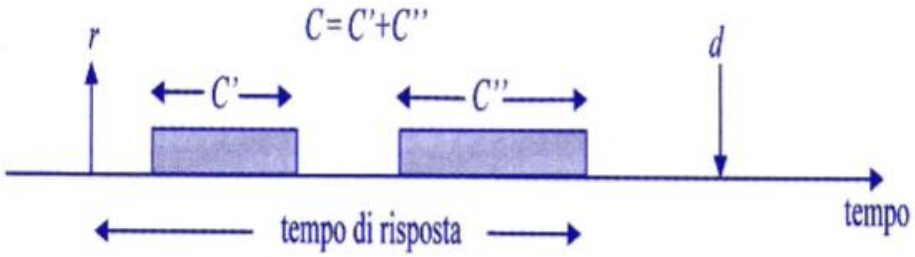
- la frequenza con cui facciamo i campionamenti;
- il numero di bit, la profondità con cui rappresentiamo i vari campioni (il numero di bit dipende molto da quello che facciamo, prendere solo 8 bit per un audio non è sufficiente, mi serviranno almeno 16 o 32 bit).

Quello che noi facciamo a seguito del campionamento è convertire una grandezza analogica in grandezza digitale: ricordare quanto visto a Reti logiche nel convertitore A/D.

A cosa potrebbero servirmi i campioni? I campioni vengono usati, ad esempio, per calcolare una media: tenendo conto di un range il sistema viene impostato per intervenire qualora la media esca da questo range (interviene stampando un output, dunque interagendo con l'ambiente esterno, l'obiettivo è riportare la media all'interno del range coi campionamenti successivi).

Processi periodici. Il fatto che io debba eseguire campionamenti periodici significa che lancerò frequentemente dei processi: si parla di **processi periodici**, ossia processi che si manifestano periodicamente per fare i campionamenti. Sulla base dell'esito di uno di uno di questi processi il sistema decidere se intervenire o no nell'ambiente esterno.

5.5.2 Caratteristiche con cui descriviamo i processi



Da un punto di vista di comportamento temporale descriviamo un qualunque processo attraverso i seguenti parametri:

- r, nell'istante di richiesta, cioè l'istante di ingresso in coda pronti del processo;
- d sta per deadline, cioè l'istante entro il quale l'esecuzione del processo deve essere terminata;
- C è il tempo di CPU necessario per completare l'esecuzione del processo (il CPU burst, sì).

La figura ci fa vedere che nel costruire lo *scheduler* è ammesso l'uso di algoritmi con preemption (il C viene diviso in due parti, per qualche motivo il processo vede a un certo punto la CPU revocata). Si consideri anche che il tempo di esecuzione dipende dai dati di ingresso: questo significa che C non è mai noto a priori.

Requisiti di ingresso. Morale della favola, i requisiti di ingresso sono:

- $C_{\max}$, il tempo massimo (calcolato dall'ingegnere informatico dopo aver studiato i casi peggiori);
- d, la deadline (decisa da chi realizza il sistema esterno, il sensore non lo progettiamo noi e chiaramente presenta dei limiti).

5.5.3 Caratteristiche di un sistema hard-real-time

Il sistema su cui ci interessa ragionare è quello hard-real-time, su cui la violazione dei vincoli temporali è grave errore. Gli algoritmi che ci interessano sono quelli con preemption, dobbiamo garantire i vincoli temporali.

5.5.3.1 Informazioni che dobbiamo conoscere su un processo

Dobbiamo conoscere per ogni processo i-esimo:

- periodo t_i, dopo il quale si rientra in coda pronti (periodicamente ogni t_i);
- deadline d_i, in un sistema hard non deve essere mai violata;
- tempo di esecuzione massimo stimato c_i.

Ci aspettiamo che la deadline sia minore rispetto al periodo complessivo (sarebbe inevitabile la sovrapposizione con i successivi processi nel caso di violazione della deadline, inoltre abbiamo processi sporadici, non possiamo eseguirli se l'asse temporale è occupato da soli processi periodici)

$$d_i \leq t_i$$

Il tempo di esecuzione si riduce usando strutture dati e algoritmi opportuni.

Facciamo un'ipotesi semplificativa. Supponiamo per semplificazione che la deadline relativa a un processo k-esimo coincida con l'ingresso in coda pronti del processo $(k+1)$-esimo: aumentiamo il tempo a disposizione del processo.

$$r_{k+1} = d_k = r_k + t_i$$

Abbiamo un sistema di processi tutti periodici. La fisica ci insegna che a sua volta il sistema risultante è ancora periodico. Il periodo di questo sistema è il minimo comune multiplo di tutti i periodi

$$T = \mathrm{MCM}(t_0, \dots, t_n)$$

Questa cosa mi serve perché devo schedulare il sistema periodico, e il tempo che considero è sempre T!

$$r_{i+1} = r_i + T = r_i + \mathrm{MCM}(t_0, \dots, t_n)$$

In questo modo abbiamo solo due dati da considerare: il periodo t_i e il CPU burst massimo C_i.

5.5.3.2 Definizione di schedulabilità di un insieme di processi

Schedulabilità di un insieme di processi. Dato un sistema di processi affermiamo che questo è schedulabile attraverso un dato algoritmo solo se, utilizzando quell'algoritmo, e al di là del carico del sistema, nessun processo va a generare un *overflow temporale* (cioè non si rispettano i vincoli temporali).

Gravità del non rispetto dei vincoli. La gravità del non rispetto dei vincoli temporali dipende dalla tipologia di sistema in tempo reale: in un hard real-time system la questione è molto seria, in un soft real-time system si cerca di evitare l'errore, ma alla fine l'unica conseguenza è una degradazione della qualità del servizio offerto.

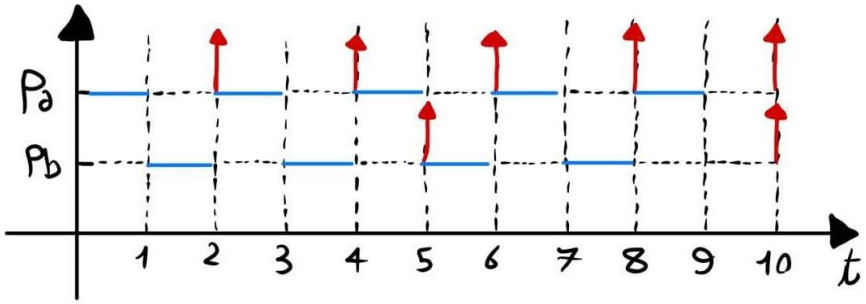
5.5.4 Criterio ottimo con priorità statica: rate monotonic

L'idea che vogliamo seguire è assegnare la priorità ai processi in base al loro periodo, stabiliamo una proporzionalità inversa: più è elevato il tempo minore sarà la priorità. Questo criterio (priorità statica) è noto come *rate monotonic*.

$$P \propto \frac{1}{t_i}$$

Rate monotonic è un criterio ottimo: se un insieme di processi non è schedulabile con *Rate Monotonic* allora non riuscirò a farlo con altri algoritmi basati su priorità statica, ergo risulta necessario passare ad algoritmi aventi priorità dinamica.

Esempio. Consideriamo i processi A e B. Abbiamo $t_a = 2, t_b = 5$ e $C_a = 1$, $C_b = 2$. Si suppone che tutti i processi siano pronti all'istante $t = 0$.

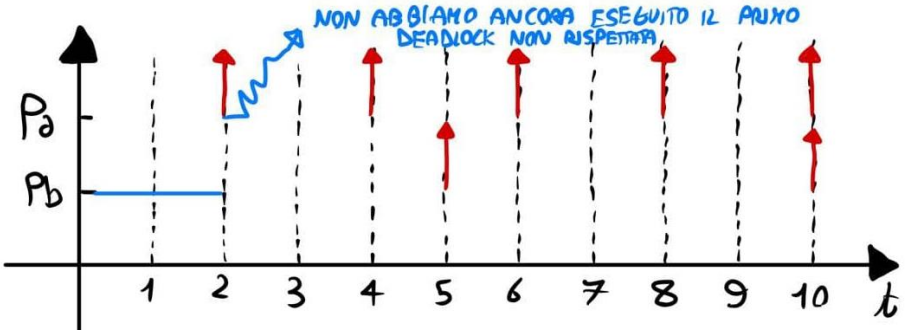


- Otteniamo il periodo del sistema calcolando il minimo comune multiplo
  $$T = \text{MCM}(2,5) = 10$$
  Ci limitiamo a disegnare solo questo periodo, non è necessario studiare le unità di tempo successive perché si ripeterà quanto posto nell'intervallo studiato. Nell'ascisse pongo il tempo, nell'ordinata l'evoluzione dei due processi. Dato il processo **periodico** k-esimo sappiamo ogni quanto si manifesta il processo con t_k e quanto dura con C_k.
- Inizio applicando rate monotonic. Abbiamo
  $$P(P_a) > P(P_b)$$
  Affermiamo ciò poiché $t_a < t_b$, dunque va in esecuzione per primo il processo P_a.
- Cosa manda in esecuzione *rate monotonic*? Le cose classiche: la terminazione del processo, la sospensione del processo, l'ingresso di un nuovo processo in coda pronti. È ammessa la *preemption*.
- Qua studiamo 10 unità di tempo: il processo A entrerà cinque volte, il processo B due volte. Rappresentiamo i momenti in cui i due processi entrano in coda pronti con delle frecce verso l'alto.
- A questo punto siamo pronti per fare la simulazione valutando se si ha *overflow* o no. Consideriamo ogni volta C_a e C_b.
  - Il processo A viene eseguito periodicamente senza necessità di smezzare in più parti l'esecuzione
  - Il processo B viene smezzato e complessivamente eseguito due volte. La prima volta con [1,2] e [3,4], la seconda volta con [5,6] e [7,8]. Ogni volta che questo processo viene smezzato il relativo descrittore torna in coda pronti. Chiaramente si ha preemption.

  All'istante 9 si ha l'ultimo step col processo A che termina: la coda è vuota, quindi lo scheduler non ha più niente da fare. Il sistema è schedulabile, non ho avuto nessuna violazione della deadline!! Non mi serve studiare gli intervalli successivi perchè quanto descritto si ripete ciclicamente.
- Considerando che non abbiamo usato l'intervallo [9,10] abbiamo un'efficienza $E = 90\%$.
  La cosa non ci dispiace: potrei avere bisogno di eseguire un terzo processo, lo faccio tenendo conto dei vincoli temporali stringenti. Questo intervallo libero potrebbe servirmi anche per eseguire processi asincroni, che eseguiamo ogni tanto.

Riprendiamo la stessa situazione di prima e supponiamo che le priorità siano invertite.

$$P(P_a) < P(P_b)$$



- Il periodo e le deadline sono sempre le solite. Parte P_b (priorità invertite), che viene eseguito per due unità di tempo visto che prima non si hanno eventi rilevanti.
- Possiamo già renderci conto che con A si va subito in overflow: ho ancora da eseguire il primo processo pronto all'istante $t = 0$, ma siamo già all'istante $t = 2$ dove dovrebbe entrare in coda il processo A successivo.
- Un sistema che non è schedulabile con *rate monotonic*, ricordiamo, non sarà sicuramente schedulabile con altri algoritmi a priorità statica (*rate monotonic* è ottimo rispetto agli algoritmi con priorità statica).

5.5.4.1 Fattore di utilizzazione

Abbiamo detto che il nostro sistema è periodico, e il periodo è pari al minimo comune multiplo dei periodi dei processi che compongono il sistema. Possiamo dire che nel periodo T ogni processo sarà eseguito un certo numero di volte: troviamo il numero di volte dividendo il periodo complessivo T per il periodo relativo al singolo processo t_i (cioè ogni quanto il processo periodico si presenta in coda)

$$n_i = \frac{T}{t_i}$$

Il risultato ci indica anche il numero di deadline che dovranno essere rispettate. Se io conosco n_i posso calcolare quanto tempo mi serve, nel periodo T, per eseguire tutti i processi che costituiscono il sistema rispettando tutti i vincoli. Iniziamo dal processo 1, e sommiamo i vari processi fino al processo m-esimo:

$$t_{\text{necessario}} = n_1 \cdot C_1 + n_2 \cdot C_2 + \cdots + n_m \cdot C_m = \frac{T}{t_1}C_1 + \frac{T}{t_2}C_2 + \cdots + \frac{T}{t_m}C_m = T\left(\sum_{i=1}^{N}\frac{C_i}{t_i}\right)$$

n_i rappresenta il tempo in senso umano, mentre C_i è il CPU burst (il numero di unità di tempo, in un certo senso). Abbiamo sostituito ad ogni k-esimo n la sua definizione. Si impone che il risultato della somma sia minore o uguale rispetto al tempo T, che è il tempo che abbiamo a nostra disposizione.

$$T\left(\sum_{i=1}^{N}\frac{C_i}{t_i}\right) \leq T$$

Definiamo il contenuto posto tra parentesi come **fattore di utilizzazione**

$$U = \sum_{i=1}^{N}\frac{C_i}{t_i}$$

Dividendo entrambi i membri per T notiamo che la disuguaglianza è valida se

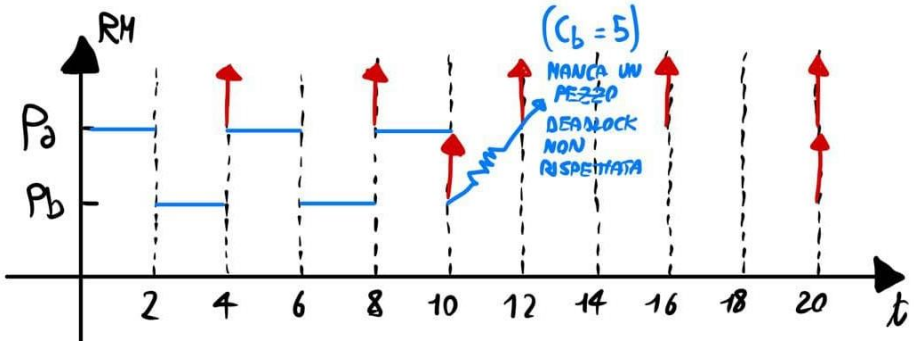
$$U \leq 1$$

La condizione è necessaria ma non sufficiente: se valida potrebbe essere possibile la schedulabilità, altrimenti non lo è di sicuro.

Esempio. Consideriamo i processi A e B. Abbiamo $t_a = 4, t_b = 10$ e $C_a = 2, C_b = 5$. Calcoliamo U

$$U = \frac{2}{4} + \frac{5}{10} = 1$$

La condizione necessaria è soddisfatta poiché $U \leq 1$, proviamo con *rate monotonic*.

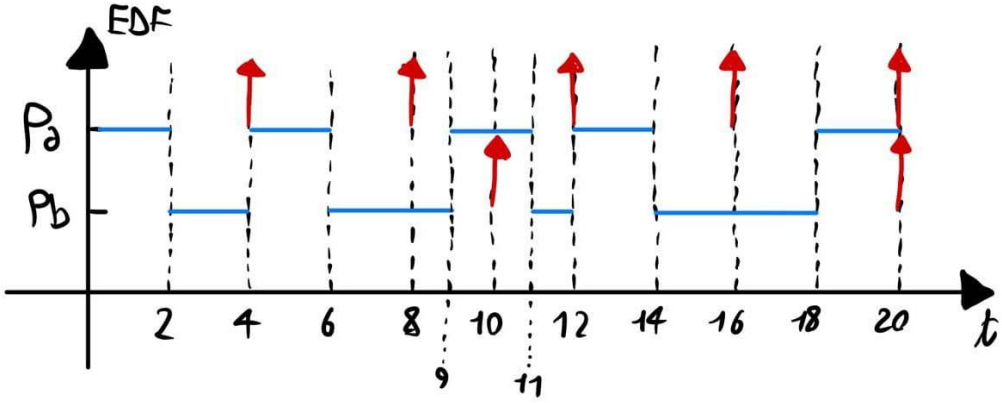


- Il periodo è $T = MCM(4,10) = 20$, quindi considero 20 unità di tempo. Anche in questo caso supponiamo che entrambi i processi siano in coda pronti all'istante $t = 0$.
- Poniamo le varie deadline con le frecce verso l'alto (che sono gli istanti in cui entrano in coda pronti i processi successivi).
- Con *rate monotonic* abbiamo $P(P_a) > P(P_b)$, quindi cominciamo col processo A. All'istante $t = 10$ si ha un overflow: non ho finito il processo B ma sono già all'istante in cui il successivo entra in coda pronti.
- **L'esempio dimostra chiaramente come la condizione sia necessaria, ma non sufficiente**.

5.5.4.2 Criterio ottimo con priorità dinamica: Earliest-Deadline-First

Abbiamo detto che se un sistema non è schedulabile con *rate monotonic* allora non è possibile schedularlo con un qualunque algoritmo basato su priorità statica. L'algoritmo *Earliest-Deadline-First* ha priorità dinamica e pone con maggiore precedenza i processi più vicini alla deadline.

Esempio. Proviamo a fare la simulazione riprendendo lo stesso caso di prima. Abbiamo un processo che nelle venti unità di tempo si ripete più volte, e un altro processo che si ripete per un numero più basso. Ogni volta che inizia un nuovo processo il timer relativo alla vicinanza alla deadline riparte dall'inizio: inizialmente tutto come prima, a un certo punto il processo B sarà più vicino alla deadline rispetto al processo A.



Contrariamente a prima si ha schedulabilità. L'algoritmo *Earliest-Deadline-First* è ottimo rispetto agli algoritmi con priorità dinamica. Un sistema che non è schedulabile con l'algoritmo EDF sicuramente non potrà essere schedulato in altro modo. Contrariamente a quando proviamo con *rate monotonic* non ci sono vie di scampo.

5.5.4.3 Attenzione all'overhead

Si tenga in mente che fino ad ora abbiamo ragionato in via teorica.

Dobbiamo considerare anche l'overhead.

Nella realtà $U = 1$ non è possibile in generale: i calcoli non tengono conto dell'overhead. Ricordiamoci che in presenza di processi aperiodici è necessario avere un po' di tempo libero nelle unità di tempo considerate (cosa impossibile con $U = 1$).

5.5.4.4 Condizione sufficiente per rate monotonic.

$$U \leq N\left(2^{1/N} - 1\right)$$

Dove N è il numero di processi in esecuzione. Se la condizione è soddisfatta sicuramente potrò schedulare con *rate monotonic*. Ovviamente noi siamo contenti perché l'overhead è minore. Si tenga a mente che l'insoddisfattibilità della condizione non indica la non schedulabilità in *rate monotonic*.

5.6 Recap su algoritmi per short-term scheduling

Criterio	Descrizione
Uso della CPU	Occupazione della CPU da parte di un processo.
Turnaround time *(o tempo medio di completamento)*	Tempo che passa dall'ingresso del processo in coda pronti al suo completamento (non necessariamente terminazione, ma restituzione di un risultato). Il tempo medio dipende dall'algoritmo di scheduling adottato, ma anche dalla complessità dell'algoritmo implementato (il task).
Throughput rate *(o produttività)*	Numero di processi completati nell'unità di tempo. Prendo l'intervallo di osservazione, conto i processi che terminano dall'istante 0 dell'intervallo in poi e divido per il tempo.
Tempo di risposta	Tempo che va dall'istante di richiesta a quando si fornisce il risultato.
Tempo di attesa	Somma degli intervalli di tempo in cui il processo attende in coda pronti.
Rispetto dei vincoli temporali	Garantire il rispetto dei vincoli posti in sistemi in tempo reale.

Sistemi multiprogrammati non in tempo reale				
Nome	**Tipo di priorità**	***Preemption***	**Descrizione**	**Problemi**
First-Come-First-Served	Statica	No	Si considerano i processi nell'ordine di arrivo. Dati con esempio visto: - Turnaround medio: 98 - Tempo medio di attesa: 60.5	Algoritmo estremamente inefficiente, l'ordine dei processi può determinare una migliore o una peggiore efficienza (processi di breve durata vengono eseguiti più avanti se prima è arrivato un processo di durata maggiore).
Shortest-Job First	Statica	No	Si considerano i processi col CPU burst totale: eseguo prima i processi più brevi. Dati con esempio visto: - Turnaround medio: 56 - Tempo medio di attesa: 18.5	- Maggiore overhead: calcolo del CPU burst e scorrimento di tutta la lista per decidere a chi assegnare la CPU. - Starvation: se la coda viene manipolata con rate elevato aumenta la probabilità che vengano introdotti di continuo processi con CPU burst basso, quindi processi di lunga durata rimangono in attesa.
Shortest-Remaining-Time-First	Dinamica	Sì	Si considerano i processi col CPU burst rimanente: eseguo il processo a cui rimane meno da eseguire. Ricalcolo ad ogni cambio di contesto la priorità del processo in esecuzione (riduzione del CPU burst rimanente). Dati con esempio visto: - Turnaround medio: 56 - Tempo medio di attesa: 18.5	- Starvation per le solite motivazioni viste per SJF.

Round-Robin	Statica	Sì	Adatto per processi interattivi. Divido la linea temporale in intervalli di tempo quanti: revoco la CPU al processo al termine di ogni intervallo. Nello scheduling si assegna la CPU con logica FCFS. Si osservi proporzionalità diretta tra tempo di attesa medio, numero di processi e intervallo di tempo quanti. $\text{Tempo di attesa medio} = \frac{\text{Numero processi}}{2} \cdot \text{quanti}$	- Sparisce la starvation: con la logica FCFS siamo certi che prima o poi qualunque processo sarà eseguito. - I processi più lunghi vengono penalizzati nell'esecuzione (inevitabile).

Sistemi multiprogrammati in tempo reale				
Nome	**Tipo di priorità**	***Preemption***	**Descrizione**	**Condizioni necessarie [e sufficienti]**
Rate Monotonic *(Criterio ottimo con priorità statica)*	Statica	Per forza!	Dato il periodo dopo il quale si manifesta una nuova "istanza" del processo periodico, affermiamo di dare maggiore priorità ai processi con periodo minore. $P \propto \frac{1}{t_i}$	**C.N. (Non C.N.S.)** affinchè si abbia schedulabilità è $U \leq 1$ Dove U è il fattore di utilizzazione, cioè un coefficiente che segnala quanto è utilizzata la CPU (attenzione all'overhead, $U = 1$ impossibile). $U = \sum_{i=1}^{N} \frac{C_i}{t_i}$ Nel caso in cui la condizione non sia valida la schedulazione non è possibile né con algoritmi a priorità statica, né con algoritmi a priorità dinamica. Nel caso sia valida si verifica la schedulabilità con *Rate monotonic* (criterio ottimo per priorità statica), graficamente e/o con la seguente condizione. **C.S.** affinchè si abbia schedulabilità con *rate monotonic* è $U \leq N\left(2^{1/N} - 1\right)$ Dove N è il numero di processi in esecuzione. **Criterio ottimo (*Rate Monotonic*)**. Un insieme di processi non schedulabile con rate monotonic non sarà sicuramente schedulabile con altri algoritmi aventi priorità statica. In caso di non schedulabilità con *rate monotonic* (priorità statica) si verifica la schedulabilità dell'insieme di processi con EDF (priorità dinamica): se non è schedulabile allora non ci sarà mai schedulabilità (vicolo cieco). **Criterio ottimo (*Earliest Deadline First*)**. Un insieme di processi non schedulabile con EDF non sarà sicuramente schedulabile con altri algoritmi (né priorità statica, né priorità dinamica).
Earliest-Deadline-First *(Criterio ottimo con priorità dinamica)*	Dinamica		Diamo maggiore priorità ai processi che sono più vicini alla loro deadline.	

5.7 Processi pesanti (task) e processi leggeri (thread)

Il processo è in sostanza "un elemento che possiede delle risorse logiche e fisiche" (cit.): un insieme di locazioni per i dati, le variabili globali e locali, lo stack e il suo descrittore, files aperti, processi figli, dispositivi di I/O, la CPU…

Spazio di memoria. Per ogni processo si definisce uno spazio virtuale di memoria appositamente dedicato. Questo spazio di memoria costituisce uno spazio unico di indirizzamento privato: l'accesso è riservato al solo processo che lo possiede. Questo significa che non è possibile per due processi pesanti accedere a un'area di memoria condivisa: i due processi devono poter comunicare. Una soluzione possibile è lo scambio di messaggi (già visto a Calcolatori elettronici con le prove pratiche), ma non è una soluzione estremamente efficiente. Per ottenere una soluzione migliore si introduce la distinzione tra processi pesanti (*task*, quelli considerati fino ad ora) e processi leggeri (***thread***).

5.7.1 Processo leggero (thread)

Il processo leggero, a differenza del processo pesante, vede assegnata solo la CPU: il descrittore contiene solo lo stato di esecuzione e pochissime altre informazioni relative all'accesso ad altre strutture dati. Queste strutture dati sono esterne al *thread*: quello che succede è che io posso avere all'interno dello stesso processo pesante più thread di esecuzione, che condividono lo stesso spazio di indirizzamento. Questo permette di avere due vantaggi:

- introduzione di un'esecuzione concorrente di thread (ogni thread rappresenta un flusso di esecuzione all'interno del processo);
- accesso allo stesso spazio di memoria da parte di più thread (possibile scambio di informazioni in modo più efficiente rispetto allo scambio di messaggi).

Con strutture dati comune si riduce l'overhead relativo al cambio di contesto: ho più elementi comuni e meno elementi specifici, quindi le cose che devo fare durante il cambio di contesto sono minori (maggiore velocità, lo ha ribadito pure Minici).

5.7.2 Multithreading

Con multithreading intendiamo una molteplicità di flussi di esecuzione all'interno dello stesso processo pesante. Potrei avere:

- programmi diversi ciascuno eseguito da un thread, che possono avanzare secondo le leggi dello scheduling e che condividono lo spazio di indirizzamento;
- lo stesso programma eseguito da più thread, che agiscono con velocità diverse.

Tutti i thread definiti in un processo condividono le risorse del processo (spazio di indirizzamento virtuale, quindi tutti i file e risorse connesse). **Ogni thread deve avere il proprio stack e il proprio descrittore**.

5.7.3 Realizzazione dei thread

La realizzazione di un sistema multithread avviene su più livelli. Possiamo muoverci in due modi.

- **A livello utente**
  Il multithreading esiste in un unico processo, cioè ho un processo che a livello di esecuzione esegue l'applicazione multithread, ma il sistema operativo è ignaro a riguardo. La CPU viene assegnata dal sistema operativo a livello di processo: tutti i thread appartenenti allo stesso processo non potranno mai essere eseguiti in parallelo (il sistema vede solo i processi, sarò poi il processo a lavorare sui thread).
  Il limite non è forte: possiamo scrivere un'applicazione concorrente.

- **A livello kernel**
  La schedulazione avviene a livello di thread (alcune versioni di Windows, dalla NT, e di linux). La CPU non viene assegnata a un processo, ma ad un thread!!! Posso sfruttare al massimo il sistema multiprocessore (con core assegnati a singoli thread, che lavorano in parallelo).

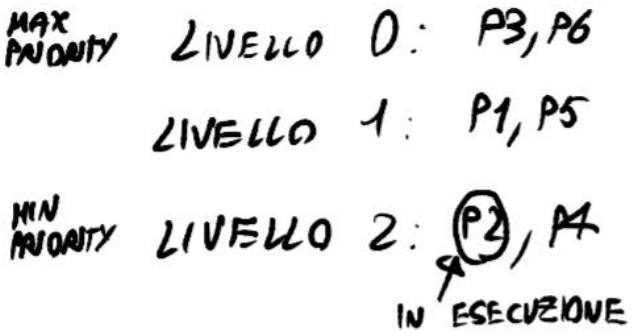
5.8 Alcuni problemi del libro

Primo problema. *Con riferimento all'esempio riportato nel paragrafo 2.8 a proposito del nucleo di un sistema a processi, si descriva il comportamento del meccanismo di trasferimento tra l'ambiente dei processi e l'ambiente del nucleo e viceversa, nell'ipotesi che esista un solo insieme di registri generali utilizzato sia dai processi che dal nucleo*.

- Trasferimento tra l'ambiente dei processi e quello del nucleo
    - Poiché le funzioni del nucleo girano in stato privilegiato mentre il codice dei processi gira in stato non privilegiato, come è stato messo in evidenza nel testo, il trasferimento del controllo tra l'ambiente dei processi e l'ambiente del nucleo **può avvenire soltanto in seguito ad un'interruzione**, che sia esterna o no (per esempio istruzione INT, già vista a Calcolatori).
    - In seguito all'accettazione dell'interruzione da parte della CPU, vengono inseriti i valori dei registri PSW e PC in cima alla pila del processo che era in esecuzione all'arrivo dell'interruzione stessa. Successivamente si pone in questi registri i valori recuperati dal vettore delle interruzioni (rispettivamente stato del registro con flag relativo ai privilegi e indirizzo della prima istruzione della routine).
    - Avendo disponibile un solo insieme di registri generali, al fine di evitare di cancellare i valori presenti nei registri al momento in cui l'interruzione è stata accettata, è necessario salvarli per poterli poi ripristinare. Per prima cosa viene quindi eseguita la funzione di *salvataggio dello stato* del processo interrotto che ha il compito di trasferire i valori di tutti i registri generali dai registri di macchina al campo *contesto* del descrittore del processo stesso. Successivamente, anche i valori di PC e PSW, precedentemente inseriti nello stack, vengono salvati nel descrittore del processo.
    - A questo punto l'intero stato del processore virtuale corrispondente al processo interrotto è presente nel campo *contesto* del suo descrittore e tutti i registri di macchina possono essere utilizzati dalla procedura di risposta alle interruzioni. **Possiamo eseguire la routine**!
    - Tale procedura, una volta terminata la gestione dell'interruzione, deve restituire il controllo all'ambiente processi. A questo punto le strade sono due:
        - riprendendo l'esecuzione del processo interrotto a partire dall'istruzione successiva all'ultima eseguita, se la gestione dell'interruzione non prevede nessuna commutazione di contesto;
        - mandando in esecuzione un diverso processo se, a causa della gestione dell'interruzione, si è reso necessario commutare il contesto dal processo interrotto a un diverso processo da mandare in esecuzione.

- Trasferimento tra l'ambiente del nucleo e quello dei processi
    - Il meccanismo è identico a quello già descritto.
    - Noto il processo a cui deve essere ceduto il controllo (o quello che ha subito interruzione o quello che viene prescelto per essere eseguito in caso di cambio di contesto) vengono prelevati dal campo *contesto* del suo descrittore tutti i valori da ricaricare nei registri generali di macchina.
    - Sempre dal campo contesto vengono prelevati i valori dei registri PSW e PC del processore virtuale del processo e tali valori vengono inseriti in cima allo stack.
    - La procedura termina quindi eseguendo l'istruzione IRET (ritorno da interruzione) che, via hardware carica nei registri di machina PS e PC i valori presenti in cima allo stack. In questo modo riparte l'esecuzione del processo schedulato a partire dall'istruzione successiva all'ultima eseguita, con la parola di stato corrispondente allo stato del processore non privilegiato e trovando nei registri generali di macchina gli stessi valori che erano presenti nel momento in cui il processo ha precedentemente perso il controllo.

Secondo problema. *Si consideri un sistema a processi nel quale a ciascun processo sia stata assegnata al tempo della creazione una priorità. Si supponga che esistano tre livelli di priorità, 0,1, 2 con priorità decrescente verso l'alto. Al livello 0 sono assegnati i processi P3, P6, al livello 1 i processi P1, P5 e al livello 2 i processi P2, P4. Si supponga che entro ciascun livello venga adottato un algoritmo Round-Robin. Nell'ipotesi che a un certo istante sia in esecuzione il processo P2 si indichi per i rimanenti processi lo stato in cui si trovano*.



Sappiamo che il sistema non si dedica ai processi in code con priorità minore fino a quando non saranno stati considerati tutti i processi posti in code con priorità maggiore. Essendo in esecuzione il processo P2, la cui priorità è 2, cioè la più bassa fra le tre priorità previste, ciò significa che tutti i processi con priorità superiore (cioè quelli con priorità 0 oppure 1) sono bloccati. Inoltre essendo P2, fra quelli con priorità 2, il primo processo in coda, ciò significa che il successivo processo nella coda di priorità 2, e cioè P4, è pronto in attesa di ricevere il suo quanto di tempo.

Quarto problema. *Si descrivano le operazioni relative al cambio di contesto tra due processi P1 e P2 nell'ipotesi che il processo P1 in esecuzione venga interrotto dallo scadere del suo quanto di tempo e che il processo P2 sia nello stato di pronto*.

L'ipotesi prevista dal problema è quella in cui il processo P1 sia in esecuzione, che il processo P2 sia in stato pronto e, in particolare, fra tutti i processi pronti sia quello che verrà scelto dall'algoritmo di *short term scheduling* per andare in esecuzione quando P1 perderà il controllo della CPU. Inoltre, è previsto dal testo del problema che il processo P1, in esecuzione, venga interrotto dal timer in seguito allo scadere del suo quanto di tempo. Ciò implica che l'algoritmo di *short term scheduling* sia di tipo preemptive. All'arrivo dell'interruzione di timer deve essere eseguita la transizione di stato relativa alla revoca della CPU al processo P1 e al suo inserimento nella coda dei processi pronti. Possiamo quindi descrivere nel seguente modo le operazioni relative al cambio di contesto fra i due processi P1 e P2:

- il contesto del processo P1 viene salvato nel proprio descrittore. In particolare, ciò significa che tutte le informazioni contenute nei registri di macchina vengono trasferite nell'area *contesto* del descrittore del processo P1;
- il descrittore del processo P1 viene inserito nella coda dei processi pronti[6];
- l'algoritmo di scheduling seleziona il primo processo presente nella coda dei processi pronti (in base all'ipotesi previste dal problema, tale processo corrisponde a P2);
- il contesto di P2 viene caricato nei registri di macchina. In particolare, tutti i valori presenti nel campo *contesto* del descrittore del processo P2 vengono trasferiti nei registri di macchina.

In seguito a questa operazione il processo P2 riprende la sua esecuzione a partire dall'istruzione successiva a quella in cui aveva precedentemente perso il controllo o, nell'ipotesi di prima attivazione, a partire dalla sua prima istruzione.

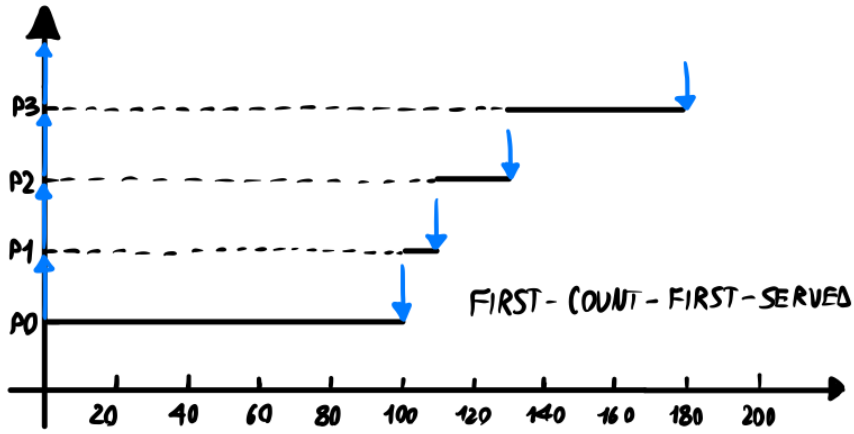
[6] In particolare, con l'algoritmo di short term scheduling di tipo Round-Robin, tale inserimento viene effettuato in modo tale che P2 sia l'ultimo della coda (logica FCFS).

Sesto problema. *Supponendo che i seguenti processi siano arrivati nell'ordine P0, P1, P2 e P3, e siano tutti presenti al tempo 0 nella coda dei processi pronti. Supponendo inoltre di conoscere le durate degli intervalli di esecuzione richiesti, come indicato nella seguente tabella*

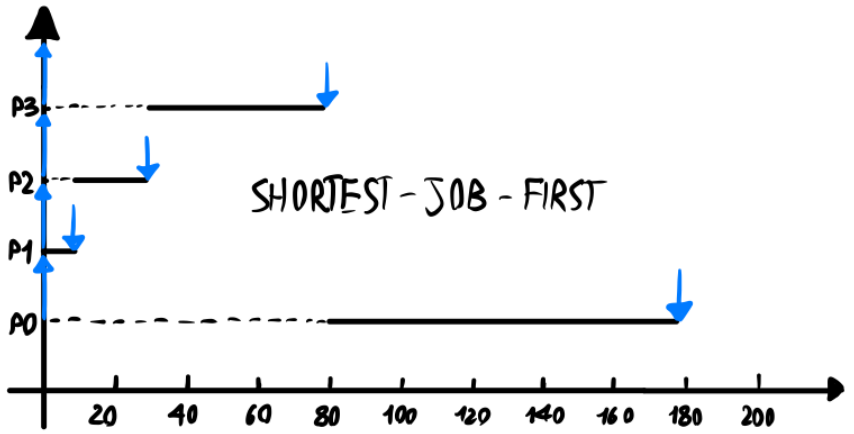
Processo	Istante di arrivo	Durata del CPU burst
P_0	0	100
P_1	0	10
P_2	0	20
P_3	0	50

- *disegnare i diagrammi temporali che illustrano l'esecuzione dei processi con algoritmi di scheduling FCFS, SJF e RR (questo con quanto = 10);*
- *calcolare il tempo di completamento di ciascun processo e il tempo di completamento medio per ciascuno dei precedenti algoritmi di scheduling;*
- *calcolare il tempo di attesa di ciascun processo e il tempo di attesa medio per ciascuno dei precedenti algoritmi di scheduling.*

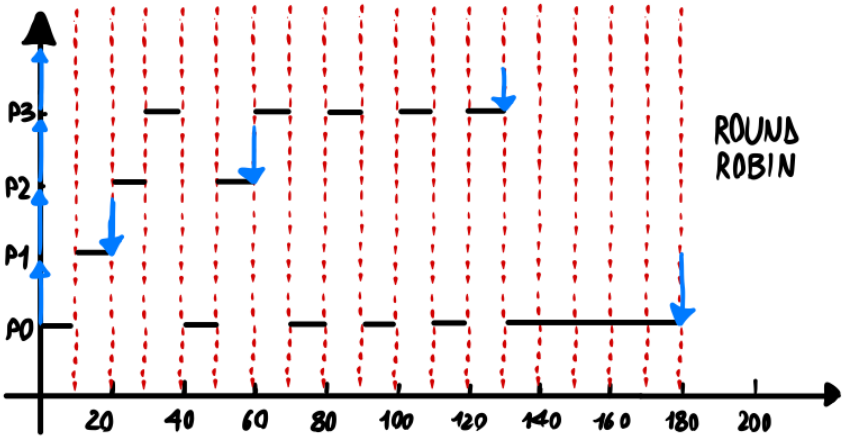
Il tempo di completamento di ciascun processo è il tempo che va dall'ingresso in coda pronti fino al termine del processo. Il tempo di attesa è il tempo in cui il processo non viene eseguito, da quando è posto in coda pronti fino al suo termine (per capire la definizione occhio a Round Robin).



Tempo di completamento	
$T_0 = 100$	$T_1 = 110$
$T_2 = 130$	$T_3 = 180$
$T = \frac{100 + 110 + 130 + 180}{4} = 130$	
Tempo di attesa	
$A_0 = 0$	$A_1 = 100$
$A_2 = 110$	$A_3 = 130$
$A = \frac{0 + 100 + 110 + 130}{4} = 85$	



Tempo di completamento	
$T_0 = 80$	$T_1 = 10$
$T_2 = 30$	$T_3 = 180$
$T = \frac{80 + 10 + 30 + 180}{4} = 75$	
Tempo di attesa	
$A_0 = 80$	$A_1 = 0$
$A_2 = 10$	$A_3 = 30$
$A = \frac{80 + 0 + 10 + 30}{4} = 30$	



Tempo di completamento	
$T_0 = 180$	$T_1 = 20$
$T_2 = 60$	$T_3 = 130$
$T = \frac{180 + 20 + 60 + 130}{4} = 97.5$	
Tempo di attesa	
$A_0 = 80$	$A_1 = 10$
$A_2 = 40$	$A_3 = 80$
$A = \frac{80 + 10 + 40 + 80}{4} = 52.5$	

Ottavo[7] problema. *Dati due processi hard real-time periodici P0 e P1 con i seguenti parametri temporali*

$$C_0 = 3, C_1 = 4, T_0 = 6, T_1 = 10$$

- *calcolare il fattore di utilizzazione della CPU di tale insieme di processi e indicare qual è la percentuale di idle time della CPU (cioè la percentuale del tempo durante il quale la CPU resta inutilizzata);*
- *disegnare il diagramma temporale utilizzando il criterio di assegnazione delle priorità RM e verificare che i processi sono schedulabili;*
- *supponiamo di introdurre un terzo processo P2 e di voler raggiungere con tale processo un fattore di utilizzazione della CPU pari a 1, mantenendo comunque schedulabili con RM i tre processi. Quali dovrebbero essere i parametri temporali di P2?*
- *Invece di introdurre un terzo processo, esiste una possibilità di incrementare il tempo di esecuzione di P1 fino a portare il fattore di utilizzazione della CPU a 1 mantenendo i processi ancora schedulabili con RM?*
- *Con riferimento ai dati del precedente punto, disegnare il diagramma temporale utilizzando il criterio di assegnazione delle priorità EDF e verificare la schedulabilità dei processi.*

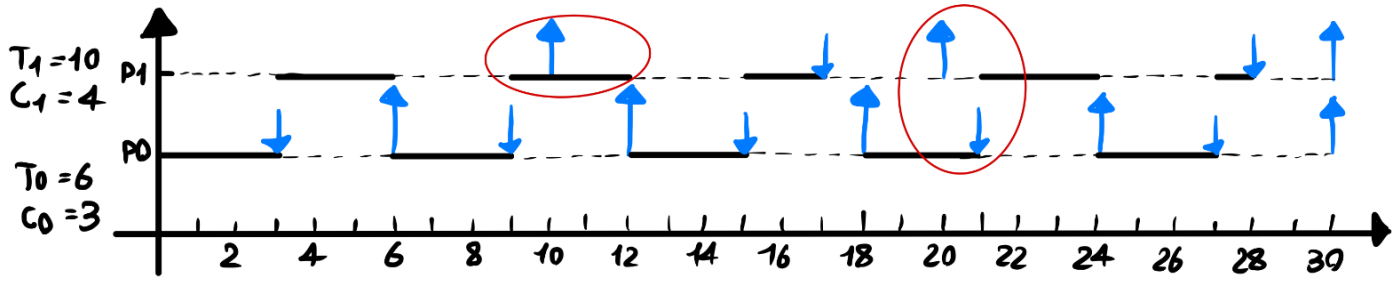
Primo punto. Per quanto riguarda il fattore di utilizzazione andiamo a sommare i rapporti.

$$U = \sum_{0}^{n-1} \frac{c_i}{t_i} = \frac{3}{6} + \frac{4}{10} = \frac{15+12}{30} = \frac{27}{30} = \frac{9}{10} = 0.9$$

La percentuale di idle time è pari al 10%. Il minimo comune multiplo dei due periodi è 30. In tale periodo P_0 utilizza per 5 volte 3 unità di tempo pari ad un totale di 15 unità di tempo, mentre P_1 utilizza per 3 volte 4 unità di tempo per un totale di 12 unità. La somma delle unità di tempo utilizzate complessivamente è quindi 27 su 30 e cioè pari al 90%, come indicato dal valore del fattore U di utilizzazione. Quindi la percentuale di idle time è del 10%.

Secondo punto. Dal risultato deduciamo osserviamo anche che la condizione necessaria (ma non sufficiente) affinchè si abbia schedulabilità statica e/o dinamica ($U \leq 1$) è soddisfatta! Disegniamo il diagramma temporale con criterio Rate Monotonic per verificare la schedulabilità con priorità statica. Nel grafico studiamo il periodo seguente

$$T = MCM(6,10) = 30$$



Attenzione ai punti evidenziati:

- Nel primo abbiamo il processo P1 che termina all'istante 10. In presenza di entrambi i processi *Rate Monotonic* avrebbe dato precedenza a P0, ma questo è stato terminato nell'istante 9 e non si manifesterà nuovamente fino all'istante 12. Segue che va avanti P1 per due unità di tempo.
- Con la freccia verso l'alto non indichiamo il momento in cui inizia l'esecuzione del processo, ma il momento in cui questo appare in coda pronti. Contrariamente a prima abbiamo entrambi i processi da eseguire, dunque diamo priorità a PO poiché $T0 < T1$.

L'insieme di processi è schedulabile con *Rate Monotonic*. La cosa è verificata pure dalla condizione sufficiente

$$U \geq N\left(2^{1/N} - 1\right) \implies 0.9 \geq 2(2^{1/2} - 1) = 0.82$$

Terzo punto. Vogliamo introdurre un terzo processo P2 e mantenere la schedulabilità insieme a P0 e P1, portando il fattore di utilizzazione a 1. Osserviamo che abbiamo delle unità di tempo in cui la CPU non è utilizzata: [17,18] e [28,30]. Otteniamo: $C_2 = 3, T_2 = 30$.

$$U = \sum_{0}^{n-1} \frac{c_i}{t_i} = \frac{3}{6} + \frac{4}{10} + \frac{3}{30} = \frac{15+12+3}{30} = 1$$

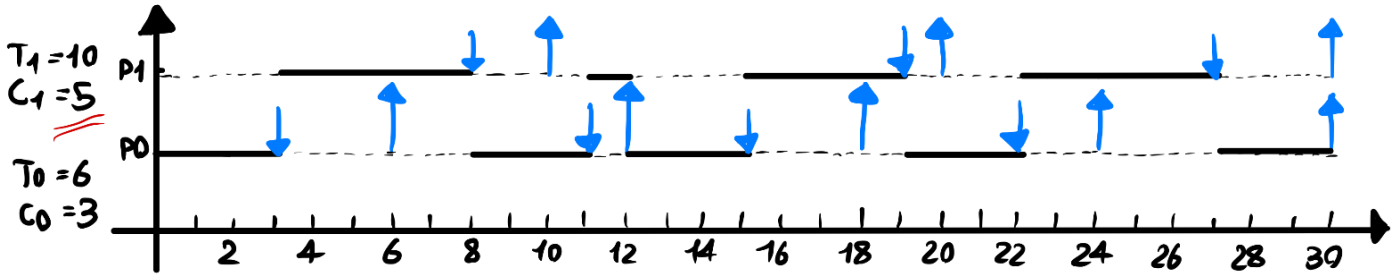
Quarto punto. Non è possibile utilizzare il fattore di utilizzazione a 1 mantenendo solo P0 e P1: non ho unità di tempo sufficienti per espandere nessuno dei due processi (si osservi che espandendo P1 emergerebbe subito un overflow all'istante temporale 10).

[7] Rimosso il settimo problema per dubbi sulla correttezza (aggiornamento del 17/12/2022)

Quinto punto. Abbiamo detto precedentemente che

$$U = \sum_{0}^{n-1} \frac{c_i}{t_i} = \frac{3}{6} + \frac{4}{10} = \frac{15+12}{30} = \frac{27}{30} = \frac{9}{10} = 0.9$$

Per ottenere $U = 1$ agiamo su $\frac{c_1}{t_1}$, ponendo $c_1 = 5$ invece di $c_1 = 4$. A questo punto l'unica cosa che ci rimane da fare è il diagramma temporale: lo facciamo applicando il criterio EDF (Earliest Deadline First, priorità di tipo dinamico).



L'insieme di processi non è schedulabile in *Rate monotonic* (priorità statica), ma è invece schedulabile in *Earliest-Deadline-First* (priorità dinamica).

6 Sincronizzazione tra processi

6.1 Modelli di interazione tra processi interagenti

Precedentemente abbiamo distinto processi indipendenti da processi interagenti. Classifichiamo le tipologie di interazione tra questi ultimi:

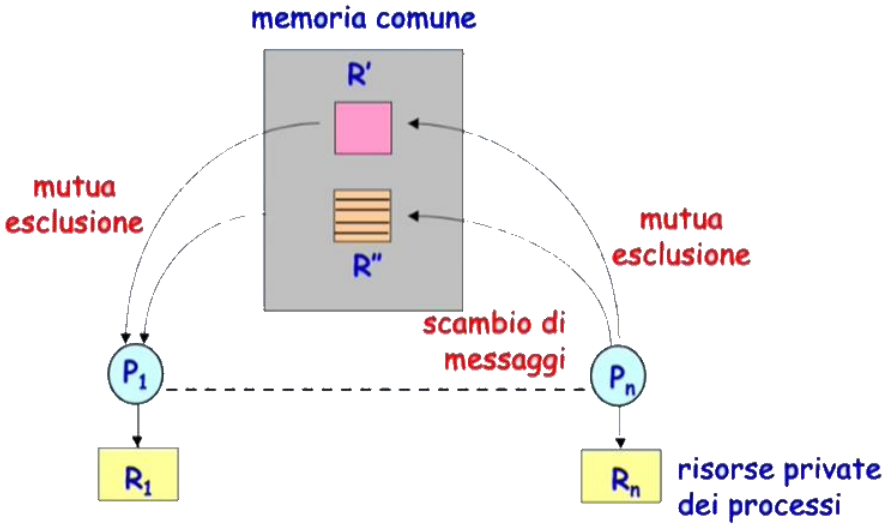
- **Competizione** (mutua esclusione, sincronizzazione indiretta/implicita)
  Uso di risorse comuni che non possono essere utilizzate contemporaneamente (mutua esclusione, il sistema deve garantire l'accesso alla risorsa stabilendo un ordine – non determinabile a priori - dei processi).
  In presenza di una competizione i processi sono ignari: abbiamo solo istruzioni che richiedono l'uso di risorse, la questione è in mano solo al sistema operativo (non gestisco la cosa nel codice del programma).
- **Cooperazione** (comunicazione, sincronizzazione diretta/esplicita)
  Processi il cui programma chiede esplicitamente di avere uno scambio di informazioni con un altro processo.
- **Interferenza**
  Tipo di interazione che non vogliamo avere, porta ad errori dipendenti dal tempo (si pensi a un buffer, potrei avere problemi se non rispetto alcuni vincoli, ad esempio necessità di scrivere nel buffer prima di scrivere).

Emerge anche il problema della sincronizzazione, che richiede un certo ordine nell'esecuzione di istruzioni.

- Nella competizione un solo processo alla volta deve avere accesso alla risorsa comune (sincronizzazione indiretta o implicita).
- Nella cooperazione le operazioni con le quali produttore e consumatore operano su un buffer devono seguire una sequenza prefissata (sincronizzazione diretta o esplicita).

6.1.1 Modello di interazioni in sistema a memoria comune

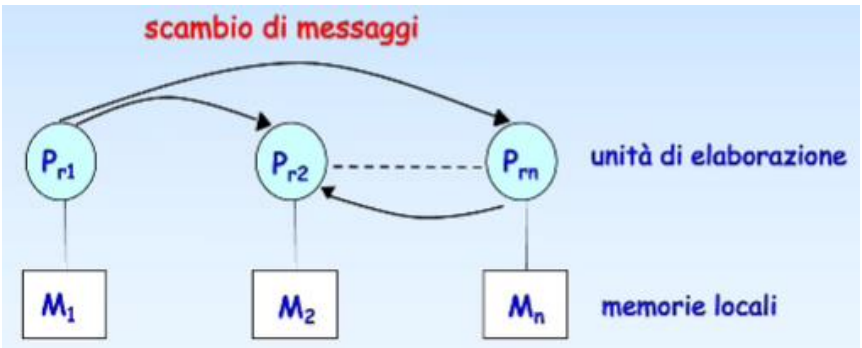
Esistono i processi $P_1, \dots, P_n$, ciascuno con le sue risorse private $R_1, \dots, R_n$. Esiste un terzo spazio, detta memoria comune, dove sono allocate risorse condivise atte alla cooperazione tra processi.



L'accesso deve essere regolato dalla mutua esclusione, nel senso che solo un processo alla volta deve accedere a questa memoria comune. L'overhead è dovuto alla mutua esclusione (ogni volta che si ha qualcosa di condiviso è necessario garantire la mutua esclusione), ma mi risparmio copie del messaggio a differenza del metodo successivo: scriviamo in una memoria e quella è l'unica utilizzata dai processi coinvolti.

6.1.2 Modello di interazioni in sistemi a memoria locale

Esistono i processi $P_1, \dots, P_n$, ciascuno con la sua memoria privata $M, \dots, M_n$.



Non esiste una memoria comune, l'unico modo per cooperare (come già anticipato) è lo scambio di messaggi. Lo scambio di messaggi avviene attraverso la chiamata di una primitiva del sistema operativo, che trasferisce fisicamente il contenuto dalla memoria del processo mittente a quella del processo destinatario, eventualmente utilizzando una rete. Quando il messaggio arriva a destinazione questo dovrà essere copiato nella memoria locale del processo destinatario. L'overhead, per le operazioni dette, è maggiore (numero di accessi in memoria maggiore).

6.2 Problema della mutua esclusione (garantire atomicità delle sezioni critiche)

Supponiamo di avere una risorsa richiesta da più processi: se il numero di richieste è superiore al numero di istanze delle risorse dovrò decidere una qualche politica di allocazione dinamica delle risorse che offra certe garanzie.

La garanzia fondamentale è che non ci siano errori nell'uso di una certa risorsa.

6.2.1 Esempio introduttivo al problema, concetto di sezione critica

Prendiamo il seguente codice, relativo a una pila con funzioni push (*Inserimento*) e pop (*Prelievo*)

```
T stack[n];
int top=-1;

void Inserimento(T y) {
  top++;
  stack[top]=y;
}

T Prelievo() {
  T temp;
  temp=stack[top];
  top--;
  return temp;
}
```

Possibile sequenza di esecuzione:

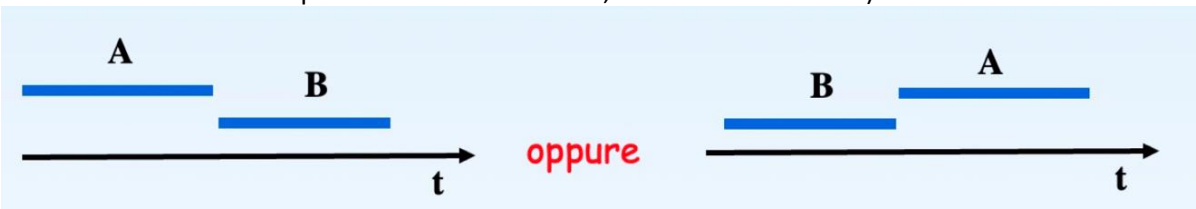
```
t0: top++;              (P1)
T1: temp=stack[top];    (P2)
T3: top--;              (P2)
T4: stack[top]=y;       (P1)
```

Supponiamo che questo stack sia condiviso da due processi (cosa ragionevole, potrebbe essere nella memoria comune detta prima). Lo scheduler potrebbe determinare la sequenza di esecuzione posta in figura: è abbastanza chiaro che le istruzioni provocano problemi:

- In P1 si annulla l'incremento di top a causa del decremento in P2;
- In P2 si copia una cosa indefinita in temp.

Si ha uno stato inconsistente. **Le funzioni Inserimento e Prelievo sono due sezioni critiche rispetto allo stack**: nella sequenza di esecuzione abbiamo problemi perché non abbiamo eseguito i due blocchi in modo atomico!!! Dobbiamo garantire la mutua esclusione sulle sezioni critiche.

Attenzione: nel garantire la mutua esclusione non ci interessa l'ordine, nell'esempio garantendo l'atomicità si garantisce consistenza, ma l'ordine delle operazioni può comportare una diversa semantica (eseguire Inserimento prima di Prelievo o viceversa comporta un risultato diverso, anche se consistente).



Le operazioni con le quali i processi accedono a una risorsa condivisa (sezioni critiche) devono escludersi mutuamente nel tempo. Chi si occupa dell'ordine? L'algoritmo di scheduling.

6.2.2 Proposta di soluzione (sbagliata)

La soluzione è implementare un protocollo che richiede un prologo e un epilogo:

P_1

```
prologo: while(occupato==1);
         occupato=1;
         <sezione critica A>;
epilogo:       occupato=0;
```

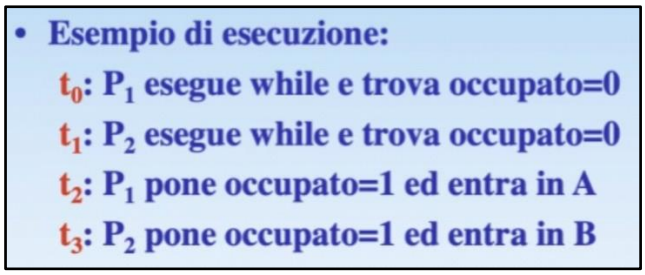
P_2

```
prologo: while(occupato==1);
         occupato=1;
         <sezione critica B>;
epilogo:       occupato=0;
```

- alla base del meccanismo si ha una variabile, che assume come valore 1 o 0;
- nel prologo verifichiamo se la risorsa risulta occupata o no;
- nell'epilogo segnaliamo il rilascio della risorsa.

Soluzione apparentemente corretta, ma in realtà abbiamo due problemi.

- Non è garantita la consistenza. Supponiamo di avere i processi P1 e P2, ciascuno si ferma sul while. Se le istruzioni sono eseguite nell'ordine qua indicato entrambi i processi entrano in area critica: l'atomicità nel prologo non è garantita.



  Quindi? *occupato* è una risorsa condivisa, **il prologo e l'epilogo sono sezioni critiche rispetto alla variabile**.
- Si ha attesa attiva (*busy wait*). La cosa non è sbagliata per definizione, ma è meglio evitarla (perché ciclare quando potrei sospendere il processo e fare qualcos'altro?).

6.2.3 Prima soluzione: primitive *lock* e *unlock*

Molti processori prevedono un'istruzione *test-and-set* per modificare il contenuto delle variabili in modo atomico. Supponiamo l'esistenza di una istruzione assembler a due parametri detta TSL[8], che copia il sorgente nel destinatario e pone il sorgente uguale ad 1.

L'atomicità è garantita per definizione: l'esecuzione della singola istruzione non sarà mai interrotta.

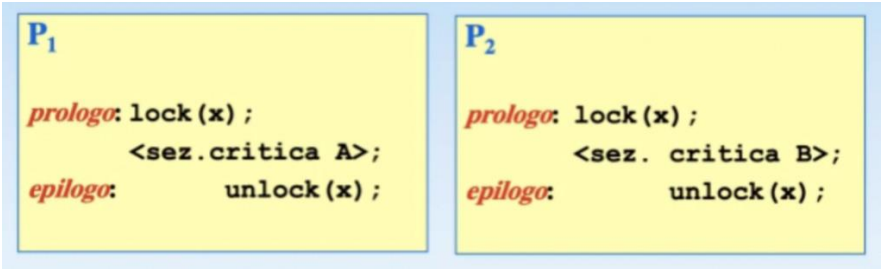
Sfruttando la cosa vado a scrivere le primitive *lock* e *unlock*.

```
lock(x):
    TSL registro,x  (copia x nel registro e pone x =1)
    CMP registro,0  (il valore di x era 0 ?)
    JNE lock        (se no, ricomincia il ciclo)
    RET             (ritorna al chiamante; accesso alla sezione critica)

unlock(x):
    MOVE x,0        (inserisce 0 in x)
    RET             (ritorna al chiamante)
```

Comportamento della `lock` con $x = 0$	Comportamento della `lock` con $x = 1$
- TSL pone come valore del registro 0 e imposta $x = 1$ - Si verifica con CMP e JNE se il valore del registro se questo è uguale a 0. Lo è quindi vado alla RET e ho finito.	- TSL pone come valore del registro 1 e imposta $x = 1$ - Si verifica con CMP e JNE se il valore del registro è uguale a 1. Non lo è quindi ritorno a TSL. Si entra in *busy wait* finchè il valore non sarà zero. - La cosa non è un problema perché x era già uguale ad 1 prima che intervenisse la TSL.

In *unlock* mi limito a porre la variabile di controllo uguale a zero. A questo punto possiamo riscrivere il codice ponendo le due primitive.



Rimane la questione *busy wait*, ma la consistenza è garantita (atomicità su prologo ed epilogo).

6.2.4 Seconda soluzione [mhm]: ordine dei processi

Imporre un certo ordine ai processi (osservo le priorità e impongo uno stretto rispetto di quell'ordine).

6.2.5 Terza soluzione [mhm]: mascherare interruzioni

Mascherare le interruzioni. La soluzione non è apprezzabile: eseguire le cose come routine di sistema comporta vanificare gli sforzi dello scheduling.

[8] **Extra**: per implementare TSL è necessario che il processore possa bloccare il bus per due cicli. Introduciamo una linea aggiuntiva, un attivo basso con cui blocchiamo o attiviamo il bus.

6.2.6 Quarta soluzione: semaforo

Il semaforo è un meccanismo di sincronizzazione, che ha svariate applicazioni (tra queste la mutua esclusione). Esso consiste in una struttura dati contenente:

- un valore intero maggiore o uguale a zero;
- una coda di descrittori di processo.

Il semaforo è caratterizzato da due funzioni che permettono la verifica dello stato del semaforo: *wait* e *signal*.

```
void wait(s) {
    if(s.value == 0)
        <il processo viene sospeso e il suo descrittore inserito in s.queue>
    else
        s.value = s.value – 1;
}

void signal(s) {
    if(<esiste almeno un processo nella coda s.queue>)
        <descrittore del primo processo estratto e inserito in coda pronti>
    else
        s.value = s.value + 1;
}
```

- Possiamo dare qualunque valore a value (da 0 a infinito). Contrariamente a Calcolatori il semaforo proposto da Avvenuti ha un value con valore minimo zero, non si va sottozero.
- La `wait` può assumere comportamento bloccante, sospendendo il processo (si cambia lo stato del processo da esecuzione a bloccato, finchè non faremo qualcosa con la `signal`. Ripensiamo alla similitudine di Lettieri della gettoniera: se non trovo gettoni (`s.value == 0`) non posso andare in bagno e mi fermo, se trovo gettoni ne prendo uno (`s.value = s.value – 1`) e vedo in bagno.
- La `signal` viene lanciata in corrispondenza di un particolare evento (in questo caso il termine di esecuzione della sezione critica). Se sono presenti processi nella coda dei processi bloccati pone quello in testa nella coda dei processi pronti (si altera lo stato da bloccato a pronti). Se non ci sono processi in coda incremento value (nella similitudine di Lettieri ciò equivale all'aggiunta di un nuovo gettone).

Attenzione: le due funzioni devono essere primitive garantendo l'atomicità. Come facciamo ciò:

- mascheriamo le interruzioni, oppure
- usiamo le primitive *lock* e *unlock*.

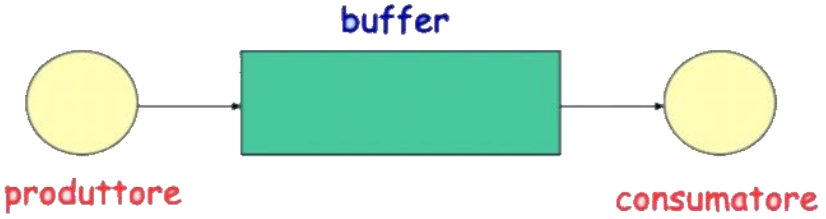
Per quanto riguarda l'implementazione del programma molto banalmente facciamo nel seguente modo (valore iniziale del contatore di mutex è 1):

```
prologo: wait(mutex);
         <sezione critica>;
epilogo: signal(mutex);
```

Attenzione: lo strumento è potente, ma si scaricano sul programmatore grandi responsabilità. Errori nella sezione critica e uso sbagliato delle primitive possono comportare un blocco del sistema.

6.3 Problemi di sincronizzazione *bounded-buffer* (produttore e consumatore)

Nella sincronizzazione esplicita e diretta non basta determinare un'esecuzione sequenziale: bisogna farlo tenendo conto di un ordine dipendente dalla logica dell'applicazione. Il caso classico è quello della comunicazione: abbiamo due processi concorrenti che devono scambiarsi delle informazioni.



- Si ha un modello a memoria condivisa, dove è presente una struttura dati condivisa *buffer* che dovrà essere acceduta in mutua esclusione.
- Un processo assume il ruolo di produttore, è colui che a un certo punto decide di produrre un messaggio.
- Dopo che il produttore realizza il messaggio deve poterlo immettere nel buffer, accedendo a una struttura dati condivisa. Non basta la regola della mutua esclusione: **dobbiamo assicurarci che il consumatore abbia letto il messaggio precedente**.
- Il consumatore è un processo che attende per prelevare/ricevere un messaggio dal buffer.

6.3.1 Risoluzione mediante primitive semaforiche (modello a memoria condivisa)

6.3.1.1 Implementazione *con buffer con unica posizione*

In un sistema a memoria condivisa è ragionevole ricorrere alle primitive semaforiche. Creiamo due semafori:

- `spazio_disponibile`, per segnalare quando il produttore può scrivere nel buffer;
- `messaggio_disponibile`, per segnalare al consumatore che può leggere il buffer.

Il valore iniziale del primo semaforo è 1, il secondo è 0 (la prima mossa viene fatta dal processo produttore).

```
// Processo produttore
do {
    <produzione del nuovo messaggio>; -> Produco il messaggio
    wait(spazio_disponibile); -> Verifico se lo spazio è disponibile (non lo è se il consumatore deve ancora leggere)
    <deposito del messaggio nel buffer>; -> Manipolo il buffer
    signal(messaggio_disponibile); -> Segnalo al consumatore che è disponibile nuovo contenuto
} while(!fine);
```

- Inizia il processo produttore, che produce il nuovo messaggio.
- Il processo produttore lancia la wait sul semaforo `spazio_disponibile`: essendo il produttore colui che farà la prima mossa esso troverà il semaforo con contatore ad 1, dunque si segnala la disponibilità del buffer per il produttore e non c'è necessità di sospendere il processo. Il buffer non è disponibile nel caso in cui il processo consumatore non abbia usufruito del contenuto precedentemente posto nel buffer: si osservi a tal proposito che la signal relativa a `spazio_disponibile` viene lanciata dal processo consumatore (e non dal processo produttore) solo dopo aver usufruito del contenuto del buffer.
- Il processo produttore deposita il messaggio nel buffer e alla fine lancia la signal sul semaforo messaggio_disponibile, segnalando al processo consumatore la disponibilità di contenuto all'interno del buffer.

```
// Processo consumatore
do {
    wait(messaggio_disponibile); -> Verifico se ci sono cose nuove da leggere (se non ci sono significa che
                                    il produttore non ha ancora eseguito la signal)
    <prelievo del messaggio dal buffer>; -> Usufruisco del nuovo contenuto
    signal(spazio_disponibile); -> Segnalo al produttore che ho consumato il messaggio
    <consumo messaggio>;
} while(!fine);
```

- Prosegue il processo consumatore, che per prima cosa lancia la wait su `messaggio_disponibile`. Se il processo produttore ha posto nuovo contenuto nel buffer allora il processo consumatore troverà un semaforo con contatore ad 1, dunque prosegue. Nel caso in cui non siano presenti nuovi contenuti il processo si sospende.
- Il processo consumatore preleva il messaggio dal buffer e alla fine lancia la signal su `spazio_disponibile`, segnalando così al processo produttore che è possibile scrivere sul buffer.

6.3.1.2 *Implementazione con buffer ad n posizioni*

Cosa succede se abbiamo un buffer con n posizioni, un produttore e un consumatore? Aggiungiamo un semaforo `mutex`, inizializzato a 1, mentre `spazio_disponibile` viene inizializzato a n e non più ad 1.

```
// produttore
wait(spazio_disponibile); -> Verifico se lo spazio è disponibile (semaforo inizializzato a n e non 1)
wait(mutex); -> MUTUA ESCLUSIONE
<ins>;
signal(mutex); -> MUTUA ESCLUSIONE
signal(msg_disponibile); -> Segnalo al consumatore che è disponibile nuovo contenuto

// consumatore
wait(msg_disponibile); -> Verifico se ci sono cose nuove da leggere
wait(mutex); -> MUTUA ESCLUSIONE
<prel>;
signal(mutex); -> MUTUA ESCLUSIONE
signal(spazio_disponibile); -> Segnalo al produttore che ho consumato il messaggio (semaforo inizializzato a n e non 1)
```

- L'introduzione del semaforo `mutex` garantisce la mutua esclusione nella sezione critica (perché il semaforo `spazio_disponibile` è posto con valore n). Creiamo una coda di produttori.
- Supponiamo che un po' di produttori abbiano prodotto dei messaggi. Il produttore si blocca o per `mutex` o per `spazio_disponibile`. Quando viene eseguita la signal il produttore potrebbe andare in esecuzione.
    - Perché metto `mutex` prima di `spazio_disponibile`? Uso lo spazio a fisarmonica: prima lo riempio e poi lo svuoto. Facciamo questo rinunciando alla flessibilità di alternare produzioni e consumi.
    - In aggiunta distribuisco i processi tra varie code, invece che concentrali sulla coda `mutex`. Quello che facciamo è:
        - considerare la logica del problema, e
        - solo dopo sistemare la questione della mutua esclusione.

Cerchiamo di rendere più chiara la cosa. Il nostro interesse è permette l'accesso da parte di più produttori e/o più consumatori al buffer: è palese che produttori e consumatori non dovranno interagire nella stessa area del buffer. Per avere un'idea di ciò che abbiamo detto immaginiamo il buffer organizzato come una coda circolare. Abbiamo due puntatori (testa e coda) che inizialmente coincidono. Riformuliamo il codice precedente introducendo questa gestione:

```
// produttore
wait(spazio_disponibile); -> Verifico se lo spazio è disponibile (semaforo inizializzato a n e non 1)
vettore[coda] = x;
coda = (coda+1)%N;
signal(msg_disponibile); -> Segnalo al consumatore che è disponibile nuovo contenuto

// consumatore
wait(msg_disponibile); -> Verifico se ci sono cose nuove da leggere
x = vettore[testa];
testa = (testa+1)%N;
signal(spazio_disponibile); -> Segnalo al produttore che ho consumato il messaggio (semaforo inizializzato a n e non 1)
```

Si osservi che abbiamo tolto il semaforo mutex: questo perché le operazioni di deposito e prelievo agiscono sulla stessa porzione di buffer se

$$p_1 = p_2 \% N$$

È dimostrabile che tale condizione non può mai verificarsi: se il buffer è pieno il processo produttore si blocca sul semaforo `spazio_disponibile`, mentre se è vuoto il processo consumatore si blocca sul semaforo `messaggio_disponibile`.

6.3.2 Risoluzione con primitive *send* e *receive* (modello a memoria locale)

La soluzione proposta si ha nel caso di un modello a memoria locale.

Abbiamo due primitive:

```
send(destinazione, messaggio)
receive(origine, messaggio)
```

La prima è invocata dal processo mittente: la destinazione consiste nell'identificativo del processo destinatario. La seconda è invocata dal processo destinatario: l'origine consiste nell'identificativo del processo mittente. Questo parametro non è obbligatorio in molte applicazioni, in particolari in quelle client-server (il server passa tempo ad aspettare di ricevere messaggi, non pensa a quale client potrebbe trasmettergli un messaggio[9]).

6.3.2.1 Formato del messaggio

Quando facciamo scambio di messaggi dobbiamo definire un formato.
Abbiamo:

- **Intestazione**
  - Origine (identificatore univoco del processo del sistema su cui stiamo agendo, il Process ID). Si tenga conto che in una rete il PID non è più univoco. Una soluzione potrebbe essere accompagnare il PID con l'indirizzo IP (in realtà risolveremo la cosa con le porte).
  - Destinazione, valgono gli stessi discorsi fatti prima.
  - Lunghezza del messaggio, importante per i socket.
- **Corpo** (messaggio)

6.3.2.2 Tipologie di primitive send/receive

- **Primitive non bloccanti**:
  - **send**: dopo la trasmissione del messaggio l'esecuzione del processo continua (chiaramente la cosa implica assenza di meccanismi di sincronizzazione). La cosa garantisce maggiore flessibilità: dopo aver mandato un messaggio posso mettermi subito a fare qualcos'altro, per esempio trasmettere altri messaggi.
  - **receive**: la receive è non bloccante e permette l'esecuzione del processo anche in assenza di messaggi.
- **Primitive bloccanti**:
  - **send**: la send bloccante **sospende il processo** finchè il messaggio non arriva a destinazione (solitamente si prende a riferimento l'inserimento in una coda).
    - **Attenzione**: con arrivo a destinazione non si intende il consumo da parte del processo destinatario.
    - **Versione alternativa**: esiste un altro tipo di send bloccante (in contrapposizione alla *send sincrona* appena vista), molto meno flessibile, che blocca il processo finchè questo non viene consumato dal processo destinatario (si va oltre l'arrivo a destinazione, è detta send RPC – *Remote Procedure Call*).
  - **receive**: la receive sincrona e bloccante sospende il processo nel caso in cui il buffer sia vuoto (il processo rimane sospeso finchè questo non riceverà un messaggio dal processo P indicato).

Una soluzione interessante può essere avere una send non bloccante e una receive bloccante: si scrivono applicazioni distribuite molto più intuitive dal punto di vista semantico (vita più semplice per il programmatore, ripensare a come si incrociavano gli occhi con le prove pratiche di Calcolatori elettronici). L'aggettivo bloccante ci piace anche perché implica la sospensione del relativo processo (quindi CPU utilizzabile per altri processi).

Scheduler non preemptive, send e receive non bloccanti non sono una cosa buona insieme: rimango a ciclare finchè non si hanno cambiamenti, senza cambiare il processo. Con lo scheduler preemptive sorridiamo: risolviamo il problema dell'attesa attiva, ma potrei avere sistemi con diverse code e alcune di queste di tipo non preemptive.

Si hanno svantaggi in entrambi gli scheduler.

[9] Al più si fa riferimento a una porta, non tanto a un processo.

Overhead.

- Le primitive send e receive non bloccanti implicano minore overhead nel sistema.
  - Si pensi a cosa dobbiamo fare per implementare delle primitive bloccanti: si copia il messaggio da trasmettere su un buffer di sistema, prima o poi questo contenuto sarà copiato in un altro buffer (che sta nello stesso nodo, o in un altro se si lavora in rete).
  - L'overhead consiste nel copiare da una zona di memoria a un'altra. Se i due processi sono su due nodi diversi contribuisce all'overhead anche il passaggio dalla rete.
- Detto in altre parole: una send o una receive di tipo non bloccanti di fatto delegano al programmatore lo scambio dei messaggi necessario per capire se il messaggio è effettivamente arrivato al destinatario. Si pensi alla comunicazione verbosa del protocollo TCP: tanti messaggi per permettere al mittente di capire se il messaggio è arrivato al destinatario.

6.3.2.3 Comunicazione diretta simmetrica con send e receive

La comunicazione simmetrica richiede che il processo produttore e il processo consumatore si indichino a vicenda attraverso i parametri di ingresso delle primitive (per esempio indicando i Process ID).

```
// Processo produttore P
pid c = …;
main() {
    msg M;
    do {
        produci(&M);
        …
        send(c,M);
    } while(!fine);
}
```

```
// Processo consumatore C
pid p = …;
main() {
    msg M;
    do {
        receive(p, &M);
        …
        consuma(M);
    } while(!fine);
}
```

Cosa deduciamo in termini di sincronizzazione? Distinguiamo due tipologie di primitive, sia *send* che *receive*:

6.3.2.4 Comunicazione diretta asimmetrica con send e receive

```
// Processo produttore p
pid c = …;
main() {
    msg M;
    do {
        produci(&M);
        …
        send(c,M);
    } while(!fine);
}
```

```
// Processo consumatore c
…
main() {
    msg M; pid id;
    do {
        receive(&id, &M);
        …
        consuma(M);
    } while(!fine);
}
```

Nella comunicazione asincrona il processo produttore continua a indicare il destinatario. **Il consumatore, invece, non indica più il mittente**: l'unica cosa che fa rispetto al mittente è leggere chi sia sfruttando un riferimento. Anche in questo caso facciamo riflessioni sul bloccare o meno:

- la send tutto come prima
- la receive, se bloccante, si blocca finchè non riceve un messaggio da qualcuno (chiunque, non si indica più un mittente in particolare). Sotto questo aspetto la receive bloccante asimmetrica è uno strumento di programmazione decisamente più flessibile della receive bloccante.

6.3.2.5 Modello cliente/servitore (client-server)

Il modello client-server si basa sulla presenza di porte, separato rispetto a quello di Process ID. In questo caso il client utilizza una primitiva `send` che fa riferimento a una porta, e non a un processo destinatario.

Perché è vantaggiosa? Perché una porta risulta meno variabile rispetto a un process ID (si pensi alla porta 80 del web, è quella e usiamo sempre quella). La lunghezza del messaggio permette a chi riceve di sapere quando termina il messaggio trasmesso.

Quando viene inviata una `send` dal browser? Nel momento in cui premiamo un link: usiamo la `send` per lanciare una richiesta, secondo protocollo TCP (il browser conosce il nodo e sa anche quale porta controllare, la 80). Ora chiediamoci: che tipo di comunicazione abbiamo? **È asincrona, con timeout**. Quello che facciamo è abbandonare le richieste nel caso si superino i tempi massimi consentiti.

6.4 Problema di sincronizzazione *Readers and writers* (Lettori e scrittori)

Problema tipico che si presenta con un certo numero di processi concorrenti:

- lettori, che leggono ma non possono scrivere;
- scrittori, che possono leggere e scrivere.

Vogliamo permettere a più lettori di fare letture allo stesso tempo. In presenza di lettori non devo avere scrittori, in presenza di scrittori non devo avere lettori. Ovviamente un solo scrittore alla volta può agire sul buffer.

Risolviamo il problema con:

- Una struttura dati detta *data set* (il buffer a cui accedono i processi)
- Un semaforo *mutex* inizializzato a 1 (atomicità di prologo ed epilogo nel rettore)
- Un semaforo *writer* inizializzato a 1 (limito il numero di scrittori contemporanei)
- Un contatore, un intero, con cui conto i lettori presenti in questo momento sulla sezione critica.

6.4.1 Implementazione dello scrittore

Nello scrittore, banalmente, verifichiamo solo che non siano presenti altri scrittori (mutua esclusione con semaforo `mutex`, inizializzato a 1).

```
do {
   wait(writer);

   // writing is performed

   signal(writer);
} while(true);
```

6.4.2 Implementazione del lettore

Il lettore è chiaramente più complesso: vogliamo dare libero accesso ai lettori, ma in contemporanea dobbiamo stare attenti alla eventuale presenza di uno scrittore (scrittore e lettori non possono coesistere).

```
do {
    wait(mutex);
    readcount++;

    if(readcount == 1)
        wait(writer);
    signal(mutex);

    // reading is performed

    wait(mutex);
    readcount—;
    if(readcount == 0)
        signal(writer);
    signal(mutex);
} while(true);
```

- Le sezioni di codice dove noi ci assicuriamo la non-presenza simultanea di lettori e scrittori sono sezioni critiche. Segue che entrambe le sezioni presenti nel codice saranno “incapsulate” da wait e signal su semaforo `mutex` (inizializzato ad 1 per garantire la mutua esclusione).
- **Prima sezione**:
  - Per prima cosa il lettore incrementa la variabile contatore *readcount* per segnalare la presenza di un nuovo lettore.
  - Se il lettore arrivato è il primo (e quindi abbiamo *readcount* uguale ad 1) si lancia la wait su writer: questo perché dobbiamo verificare la presenza di scrittori all’interno dell’area di lettura. Se è presente uno scrittore la wait è bloccante e il processo si sospende.
  - Si esegue la wait solo all’accesso del primo lettore poiché in presenza di più lettori è automatica l’assenza di uno scrittore (mi risparmio operazioni inutili).
- **Seconda sezione**:
  - Il lettore ha finito di leggere, decremento la variabile contatore *readcount*.
  - Nel caso in cui la variabile *readcount* sia uguale a zero significa che il lettore che ha finito è l’ultimo rimasto: segue la necessità di lanciare la signal sul semaforo writer, in modo tale da permettere a uno degli eventuali scrittori in attesa di poter agire sul buffer.

6.5 Problema di sincronizzazione *Dining-Philosophers* (Problema dei cinque filosofi)

La metafora permette di spiegare un problema di sincronizzazione, che è preludio di una situazione di deadlock.

Siamo filosofi che parlano di oriente (al centro del tavolo c'è il riso). Il riso è una risorsa, i cinque filosofi seduti attorno al tavolo devono fare delle operazioni:

1. si procurano le bacchette
2. mangiano
3. rilasciano le bacchette

Attenzione: abbiamo cinque filosofi, cinque bacchette, **ma ne servono due**!

Perché è un problema di sincronizzazione?

- I filosofi rappresentano cinque processi/thread. Questi si comportano tutti allo stesso modo e possono trovarsi in tre stati possibili:
  - **pensante**, il filosofo/thread non ha bisogno di risorse;
  - **affamato**, stato transitorio durante il quale il filosofo/thread deve trovare le risorse necessarie per poter mangiare;
  - **mangiante**, mangia, ha la bacchetta a sinistra e quella a destra.
- Quando un thread passa dallo stato pensante allo stato affamato fa due operazioni
  - si procura la bacchetta a destra;
  - si procura la bacchetta a sinistra

  Dopo avere ottenuto le due bacchette passa allo stato mangiante.
- Il problema si ha con le bacchette: il numero di risorse è inferiore rispetto al numero di bacchette necessarie affinchè mangino in contemporanea tutti e cinque i filosofi.

6.5.1 Implementazione del filosofo

Immaginiamoci un vettore *chopstick* di cinque semafori: associamo ciascun semaforo ad ogni bacchetta. Il filosofo fa wait sulle bacchette e si pone in attesa nel caso in cui una bacchetta richiesta non sia disponibile.

```
do {
    wait(chopstick[i]); // Il filosofo prende la bacchetta alla sua sinistra
    wait(chopstick[(i+1)%5]); // Il filosofo prende la bacchetta alla sua destra

    // eat

    signal(chopstick[i]); // Il filosofo rilascia la bacchetta alla sua sinistra
    signal(chopstick[(i+1)%5]); // Il filosofo rilascia la bacchetta alla sua destra

    // think
} while(true);
```

Dopo aver superato le wait siamo in sezione critica: il filosofo può prendere il riso e mangiarlo usando le bacchette. Alla fine il filosofo chiama le signal sulle due bacchette (prima quella a sinistra e poi quella a destra) per rilasciarle.

Problema: a causa delle diverse velocità dei processi (per ragioni esterne all'applicazione) questi potrebbero trovarsi in sezione critica insieme sulla seconda wait. Chiaramente si entra in un'*attesa ciclica*: tutti i processi sono in attesa della wait interna e il sistema si congela.

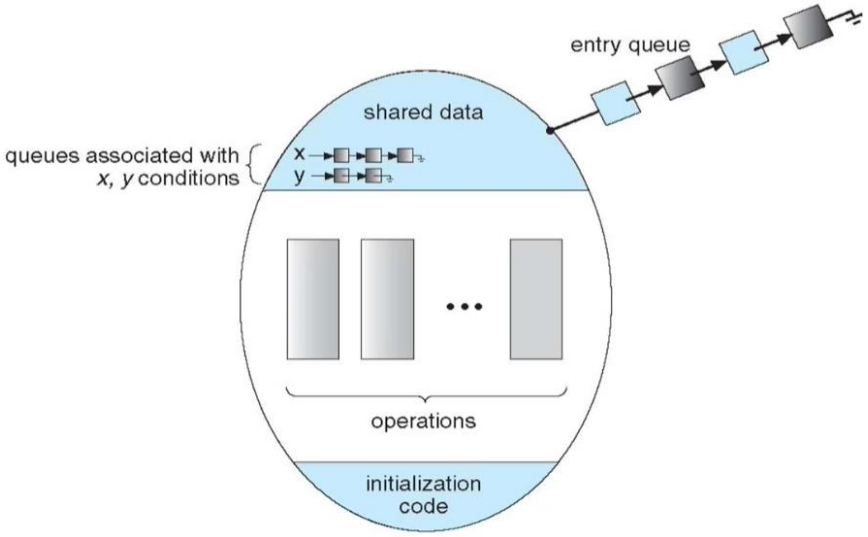
6.6 Costrutto *monitor*

Il monitor è un costrutto che fornisce un meccanismo più complesso rispetto al semaforo. Permette una programmazione priva di alcuni errori detti precedentemente. Prevede un certo numero di strutture dati e delle procedure che agiscono su queste strutture (SOLO QUESTE PROCEDURE).

La cosa interessante è che tutte le procedure definite sul monitor vengono eseguite in mutua esclusione, non devo ricordarmi di fare wait e signal perché la cosa è già implementata nel costrutto.

```
Monitor monitor-name {
    // shared variable declarations
    Procedure P1(…) {…}
    Procedure Pn(…) {…}
    Initialization code(…) {…}
}
```

Possiamo rappresentare il monitor attraverso la seguente immagine:



Nell'implementazione dobbiamo distinguere i processi/thread che sono in attesa fuori dal monitor e quelli che sono in attesa all'interno del monitor in caso di sospensioni con variabili condition (si consideri che questi hanno già acquisito la mutua esclusione).

6.6.1 Variabili *condition*

All'interno del monitor possono essere definite variabili condition. Su queste variabili sono predefinite due operazioni. Supponiamo di avere una variabile `x`:

- `x.wait()`, il processo che la invoca si sospende SEMPRE (differenza col semaforo propriamente detto) finchè non viene lanciata la `x.signal();`
- `x.signal()`, si riesuma uno dei processi che ha precedentemente invocato la `x.wait()`. Nel caso in cui nessun processo abbia invocato la `x.wait()` sulla condizione allora non si hanno effetti.

Prima di implementare il costrutto monitor dobbiamo chiederci quale sia la politica da utilizzare quando un processo

1. ***signal and wait* (politica adottata nell'implementazione)**
   Il processo P, che ha eseguito la signal, si mette in attesa finchè Q (che si era sospeso con la wait) non lascia il monitor o si mette in attesa su qualche condizione.
2. ***signal and continue***
   Il processo P, che ha eseguito la signal, continua. Q (che si era sospeso con la wait) attende che P lasci il monitor o si metta in attesa su una qualche condizione.

6.6.2 Implementazione del costrutto

6.6.2.1 Esempio di uso del costrutto monitor come premessa: problema dei cinque filosofi

Proviamo a implementare il problema dei cinque filosofi usando il monitor, in modo da evitare la *busy wait*. Si osservi che la logica di sincronizzazione risulta implementata nel monitor (la cosa è un bene, siamo sollevati dal gestire un qualcosa di complesso, su cui è facile commettere errori).

```
monitor diningPhilosophers {
    enum { THINKING; HUNGRY; EATING } state[5];
    condition self[5];

    initialization_code() {
        for(int i = 0; i < 5; i++)
           state[i] = THINKING;
    }

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if(state[i] != EATING) self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        /// test left and right neighbors
        test((i+4) % 5);
```

Con `initialization_code` inizializziamo tutti gli n filosofi ponendoli nello stato pensante (All'inizio non mangia nessuno).

```
        test((i+1)%5);
    }
    void test(int i) {
        if((state[(i+4)%5] != EATING) && (state[i] == HUNGRY) && (state[(i+1)%5] != EATING)) {
            state[i] = EATING;
            self[i].signal(); // Significativa solo nella chiamata di test da parte di putdown
        }
    }
}
// Codice del filosofo
DiningPhilosophers.pickup(i); // Prologo
// EAT
DiningPhilosophers.putdown(i); // Epilogo
```

- L'utente ha a disposizione due procedure:
  - `void pickup(int i)`
    Col parametro di ingresso i indichiamo il filosofo che vogliamo porre nello stato EATING (acquisizione della risorsa).
    - Il filosofo viene inizialmente posto nello stato HUNGRY.
    - Successivamente viene lanciata la funzione `test`. La funzione può determinare un cambiamento nello stato del filosofo: questo viene posto nello stato EATING qualora sia possibile acquisire entrambe le bacchette.
    - Dopo aver chiamato la funzione test verifichiamo se è avvenuto il cambiamento o no. Se lo stato del filosofo a seguito dell'esecuzione della funzione non è EATING allora il processo deve essere sospeso (ricordarsi che la wait è sempre bloccante).
  - `void putdown(int i)`
    Col parametro di ingresso i indichiamo il filosofo che vogliamo porre di nuovo nello stato THINKING (rilascio della risorsa).
    - Il filosofo viene subito posto nello stato THINKING.
    - Dopo aver rilasciato la risorsa dobbiamo "avviare" i controlli che risveglieranno i processi/filosofi adiacenti che non hanno potuto acquisire entrambe le bacchette, e quindi accedere alla risorsa riso. Eseguo la test due volte: una volta sul filosofo alla sinistra e un'altra sul filosofo alla destra.

- **Cosa fa la funzione test?**
  `void test(int i)`
  La funzione test è chiamata sia in pickup che putdown e permette di verificare lo stato delle bacchette che richiede il filosofo i. Pone il filosofo indicato nello stato EATING se può acquisire entrambe le risorse. Risveglia eventuali processi/filosofi che si sono posti in attesa a causa della mancanza di bacchette (la signal non fa nulla se non ci sono filosofi/processi in attesa, per questo possiamo porla anche nella test di pickup nonostante sia inutile).
  - Quello che facciamo non è tanto verificare lo stato delle bacchette, ma quello dei filosofi adiacenti al filosofo i: se uno dei filosofi adiacenti, o entrambi, stanno mangiando significa che il filosofo i non può acquisire le due bacchette necessarie e quindi non può porsi nello stato EATING. La condizione in test è vera se il filosofo può porsi nello stato EATING.

- La soluzione previene il deadlock.
  L'accesso al monitor avviene in mutua esclusione, il processo si sospende all'interno del monitor solo invocando la wait su una condizione (l'implementazione della wait ci garantisce che non avremo attese cicliche). Siamo certi di rompere la catena anche perché dopo aver risvegliato un filosofo si verifica se ci sono filosofi adiacenti da risvegliare.
- La soluzione non previene la *starvation*.
  Qualche filosofo potrebbe essere ritardato nell'accesso alla risorsa: la cosa è dovuta alla località della funzione test, che non controlla tutti i filosofi ma solo quelli adiacenti. Filosofi adiacenti a filosofi che lavorano molto velocemente verranno risvegliati prima, mentre filosofi adiacenti a filosofi che lavorano lentamente saranno risvegliati dopo molto più tempo.

6.6.2.2 Cosa ci serve per implementare il monitor?

Per implementare il monitor utilizziamo i meccanismi di sincronizzazione che già conosciamo (primitive semaforiche). Definiamo quanto segue:

- `semaphore mutex;`
  Semaforo posto inizialmente ad 1, lo utilizziamo per gestire la mutua esclusione all'interno del monitor.
- `semaphore next;`
  Semaforo posto inizialmente a zero, lo utilizziamo per gestire i processi che si sono posti in attesa all'interno del monitor (sia quello sospesi a causa della wait, che quelli sospesi a causa della signal).
- `int next_count = 0;`
  Variabile contatore posta inizialmente a zero, si conteggiano i processi che rimangono all'interno del monitor a causa della signal.
- `int x_count = 0;`
  [Definita per ogni variabile condition] Variabile contatore posta inizialmente a zero, si conteggiano i processi che rimangono all'interno del monitor a causa della wait.
- `semaphore x_sem;`
  [Definito per ogni variabile condition] Semaforo posto inizialmente a 0, si usa per sospendere i processi che hanno chiamato la wait dopo aver eseguito eventuali processi all'interno del monitor. Il fatto che si esegua la wait su questo semaforo è testimonianza del fatto che la wait su condition blocca sempre.

Attenzione all'implementazione della politica signal and wait negli snippet seguenti.

6.6.2.3 Procedura generica implementata nel monitor

Il corpo della procedura generica diventa la sezione critica da proteggere

```
wait(mutex);

// body della procedura

if(next_count > 0)
    signal(next); // Risvegliamo uno dei processi che si sono sospesi dentro il monitor
else
    signal(mutex); // Risveglio i processi che aspettano fuori (m.escl.) o esco dal monitor
```

- Abbiamo la mutua esclusione su ogni procedura/operazione definita nel monitor grazie al semaforo mutex.
- Alla fine dovrei rilasciare il monitor, ma dobbiamo tenere a mente che col meccanismo delle variabili condition potrebbero esserci processi in attesa all'interno del monitor: con processi in attesa intendiamo quei processi che si sono sospesi dopo aver invocato la signal, **a causa della politica *signal and wait***.
- Diamo priorità a quelli, dobbiamo uscire dal monitor solo quando avremo svuotato la coda. Verifichiamo se ci sono processi del genere in attesa sulla coda del semaforo `next`, se ci sono (`next_count > 0`) invochiamo la signal su next.
- Prima o poi la signal su mutex sarà eseguita per forza, precisamente quando non ci saranno più (all'interno del monitor) processi sospesi a causa della signal. In quel caso avremo uno tra i seguenti eventi: l'esecuzione dei processi/thread in attesa fuori dal monitor (che si sono sospesi a causa della mutua esclusione intrinseca nel costruttore), oppure l'uscita dal monitor e l'esecuzione di cose che non hanno a che vedere col monitor.

6.6.2.4 Implementazione della wait su una variabile condition

Ricordiamo le cose definite per ciascuna variabile condition (dette prima). Analizziamo il codice:

```
x_count++;
if(next_count > 0)
    signal(next); // wait nell'implementazione della signal
else
    signal(mutex);
wait(x_sem);
x_count—;
```

- Abbiamo detto che la wait ha carattere sempre bloccante, quindi in ogni caso il thread dovrà sospendersi (lo farà sul semaforo x_sem). Per prima cosa incremento `x_count` per ricordarmi (nell'implementazione della signal) che ci sono processi/thread in attesa.
- Anche qua stabiliamo che devono avere priorità i thread che si sono sospesi all'interno della signal. Se ci sono (`next_count > 0`) lancio la signal su `next`. L'alternativa è la signal su mutex, che significa eseguire

i thread che aspettano fuori dal monitor o andare subito alla wait (se non c'è nulla incremento solo il contatore).

- Concludiamo con la wait su `x_sem`, che sospende il processo. Il processo sarà risvegliato con il lancio della signal, implementato successivamente. Al risveglio del processo si rende consistente il contatore `x_count` decrementandolo.

6.6.2.5 Implementazione della signal su una variabile condition

La signal risulta coerente con la wait

```
if(x_count > 0) { // Se ci sono processi sospesi sulla variabile condition
    next_count++; // Conto i processi che sono rimasti nel monitor a seguito di signal
    signal(x_sem); // SIGNAL AND
    wait(next);    // WAIT, POLITICA ADOTTATA
    next_count—;
}
```

- Verifichiamo se sono presenti processi che si sono sospesi a causa della wait. Se non ci sono processi del genere la signal non fa niente.
- Nel caso in cui vi siano processi in attesa a causa della wait (all'interno del monitor) incremento la `next_count`, visto che abbiamo un nuovo processo che si sospende a causa della signal (politica signal and wait).
- Col lancio della signal su x_sem non solo completiamo l'esecuzione della wait (dove rendiamo consistente il contatore), ma concludiamo l'esecuzione del thread che si era sospeso sulla wait (salvo ulteriori sospensioni).
- Abbiamo detto che viene implementata la political signal and wait: risveglio il processo che si era sospeso a causa della wait, il processo che ha lanciato la signal rimane sospeso fino a quando il processo risvegliato non uscirà dal monitor. Questo avviene negli if in fondo all'implementazione della procedura vista poco fa (quando lanciamo la signal su next).

6.6.2.6 Esempio di uso del costrutto monitor: `ResourceAllocator`

Vediamo un altro esempio di uso nel monitor nel regolare l'accesso alle risorse. Vogliamo scrivere un monitor che permetta di gestire l'accesso a una risorsa.

```
monitor ResourceAllocator {
    boolean busy;
    condition x;

    void acquire(int time) }
        if(busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization_code() {
        busy = FALSE;
    }
}
```

Ho bisogno di un booleano per sapere se la risorsa è libera o utilizzata, ma anche di una variabile condition per porre in attesa chi richiede la risorsa e non può acquisirla subito.

Inizialmente la risorsa non è stata acquisita da nessuno.

- **Funzione *acquire***.
  Funzione semplice a due step: per prima cosa verifico se la risorsa è disponibile o meno, nel caso in cui non lo sia sospendo il processo; se la risorsa è disponibile pongo busy uguale a true. Ritorneremo a questo punto della procedura acquire quando il processo sarà risvegliato.
  - **A cosa serve il parametro time, che incontriamo ora per la prima volta?**
    Per avere una wait con priorità, un processo aspetta ma solo per un certo tempo.
- **Funzione *release***.
  Pongo la busy uguale a false per rilasciare la risorsa, e lancio la signal per risvegliare eventuali processi in attesa. Ribadiamo che la signal è come se non ci fosse nel caso in cui non ci siano processi in attesa.

6.7 *deadlock* / Stallo / Blocco critico

In un sistema multiprogrammato più processi possono competere per l'uso di un numero definito di risorse. La cosa non può essere analizzata in modo semplice, dobbiamo tener conto del problema del deadlock (detto anche *stallo* o *blocco critico*).

6.7.1 Definizione di deadlock

Consideriamo un insieme di processi: ogni processo possiede almeno una risorsa e ha bisogno di almeno un'altra risorsa per arrivare a completamento. Il problema *deadlock* consiste nell'acquisizione di risorse in mano ad altri processi dello stesso insieme. Un processo che richiede una risorsa è posto in attesa, e non può fare altro. Il fenomeno dipende soprattutto dalla velocità relativa di esecuzione dei processi (che non dipende dal codice dei nostri programmi).

6.7.2 Situazione tipica: *hold and wait*

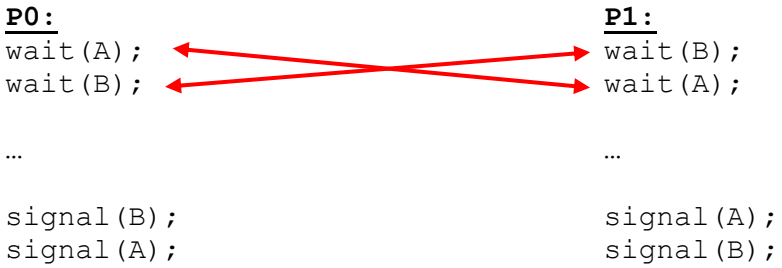
Pensiamo alla situazione dove un processo che detiene certe risorse si pone in attesa per ottenerne altre: i tempi si allungano. Le risorse già in mano al processo sospeso cosa fanno? Se non interveniamo queste rimangono in mano al processo sospeso, e non possono essere usate da altri processi durante l'attesa per la nuova risorsa.

6.7.2.1 Esempio banale e semplificativo

- Ho due hard disk.
- Il processo P1 possiede il primo hard disk e il processo P2 possiede il secondo.
- Per proseguire il processo P1 ha bisogno del secondo hard disk, mentre il processo P2 ha bisogno del primo

6.7.2.2 Esempio con i semafori

- Supponiamo di avere i processi P0 e P1, e i semafori A e B (entrambi inizializzati a 1).
- L'esecuzione delle seguenti primitive (a sinistra in P0 e a destra in P1) potrebbe provocare situazioni di deadlock



- Immaginiamo la seguente sequenza temporale di azioni

```
P0 esegue wait(A) e acquisisce la risorsa A
P1 esegue wait(B) e acquisisce la risorsa B
P0 esegue wait(B) e si blocca
P1 esegue wait(A) e si blocca
```

  - Il processo P0 si sospende sulla seconda wait in attesa di ricevere B, attualmente in mano a P1.
  - Il processo P1 si sospende sulla seconda wait in attesa di ricevere A, attualmente in mano a P0.
  - Se uno è in attesa dell'altro e viceversa siamo in deadlock.

6.7.3 Modello di sistema: tipi di risorse e istanze di tipi di risorse

Parleremo di tipi di risorse e non di risorse, senza fare distinzione tra risorse fisiche e logiche.

$$R_1, R_2, \dots, R_m$$

Ciascun tipo di risorsa è caratterizzato da un certo numero di istanze (si pensi alla CPU, con un solo processore si ha una sola istanza, mentre della memoria abbiamo più istanze se la concepiamo in partizioni).

Costituisce parte integrante del modello il solito protocollo caratterizzato da:

- richiesta (***prologo***),
- uso (***sezione critica***), e
- rilascio (***epilogo***).

Abbiamo già visto diverse implementazioni del protocollo: primitive, monitor, …

6.7.4 Condizioni necessarie (ma non sufficienti) affinchè si manifesti il deadlock

Dato un insieme di processi $P_1, \dots, P_n$ e un insieme di risorse $R_1, \dots, R_m$ si può verificare una condizione di deadlock se sono valide **tutte** le seguenti condizioni:

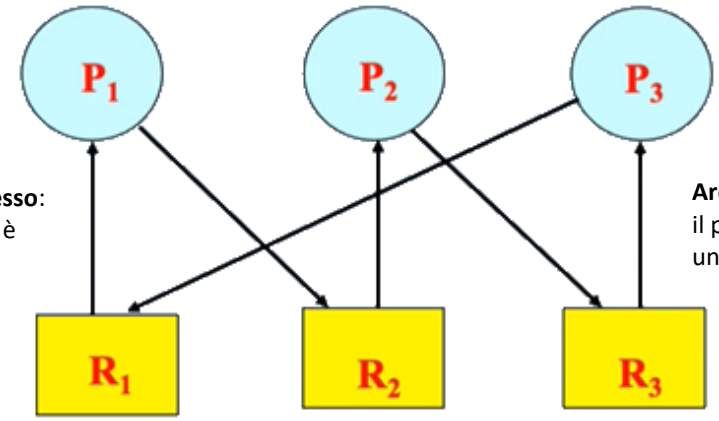
- **Mutua esclusione**.
  Solo un processo alla volta può utilizzare una risorsa. Si tenga a mente che rinunciare alla mutua esclusione implica rinunciare al modello di sistema appena descritto.
- ***Hold and wait*** (possesso e attesa di una risorsa, già anticipato).
  Abbiamo processi che possiedono altre risorse e si pongono in attesa per ottenere risorse aggiuntive. La risorsa richiesta è già posseduta da altri processi, dunque si pone in attesa per il carattere bloccante della primitiva.
- ***No preemption***.
  Il sistema operativo ha la possibilità di revocare risorse già assegnate ai processi. La possibilità di revoca è strettamente collegata alla condizione precedente: per risolverlo il sistema operativo potrebbe revocare le risorse e assegnarle ad altri processi.
- ***Circular wait*** (attesa circolante).
  Dato un insieme di processi $\{P_0, P_1, \dots, P_n\}$. Se si manifestasse un ciclo in un grafo di allocazione delle risorse allora potrebbe esserci deadlock.
  - **Esempio**: il processo P_0 attende una risorsa posseduta da P_1, P_1 attende una risorsa posseduta da P_2 e così via fino a P_n, che è in attesa di una risorsa posseduta da P_0.

Se una sola di queste condizioni risulta non verificata allora non avremo deadlock.

6.7.5 Grafo di allocazione delle risorse per descrivere l'attesa circolare

L'attesa circolare può essere descritta attraverso un grafo di allocazione delle risorse (detto anche ***grafo possesso-attesa***). Esso descrive come le risorse sono allocate ai processi.

I **cerchi** rappresentano i processi $P = \{P_1, P_2, \dots, P_n\}$

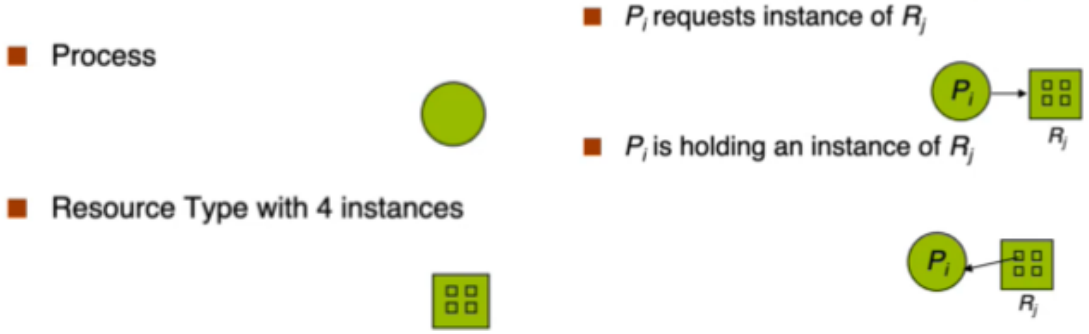


Arco da risorsa a processo: un'istanza della risorsa è assegnata al processo.

Arco da processo a risorsa: il processo è sospeso in attesa di un'istanza della risorsa

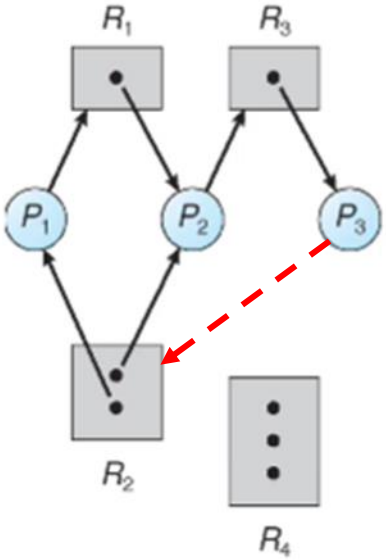
I **rettangoli** rappresentano le risorse $R = \{R_1, R_2, \dots, R_n\}$

Il fatto che gli archi siano orientati ci permette di individuare cicli in modo agile. Nell'esempio in foto potrei avere deadlock (ho il ciclo *R1-P1-R2-P2-R3-P3-R1*). Consideriamo anche i seguenti simboli (più avanti introdurremo pure il *claim edge*):



6.7.5.1 Esempio di grafo con ciclo e deadlock

Il processo P2 detiene un'istanza del tipo di risorsa R1, e un'istanza del tipo di risorsa R2. Ha richiesto un'istanza del tipo di risorsa R3, ma l'unica disponibile è in mano a P3, quindi P2 si sospende.



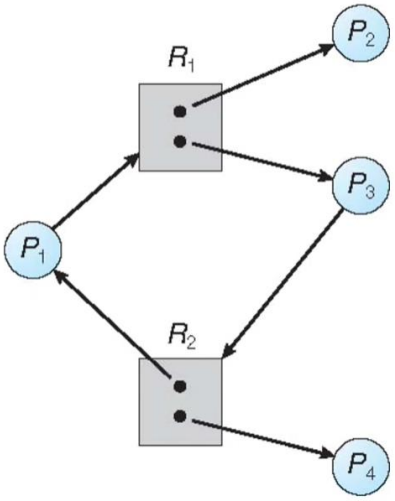
Il processo P1 detiene un'istanza del tipo di risorsa R2, e ha richiesto un'istanza del tipo di risorsa R1. Quest'ultima ha entrambe le istanze già detenute da altri processi, quindi P1 si sospende.

Il processo P3 detiene un'istanza del tipo di risorsa R3, e ha richiesto un'istanza del tipo di risorsa R2 (se si considera la freccia rossa tratteggiata). Quest'ultima ha entrambe le istanze già detenute da altri processi, quindi P3 si sospende.

Consideriamo l'esempio a lato (inizialmente senza la freccia rossa) con tre processi e quattro tipi di risorse.

- R1 ed R3 hanno una sola istanza, R2 ne ha due, R4 ne ha tre. Il modello prevede che un processo si sospenda se la risorsa richiesta è già utilizzata da altri (cioè se tutte le istanze sono state acquisite da altri processi).
- **P1**: ha richiesto e ottenuto un'istanza di R2; ha lanciato una wait per assumere R1, ma l'unica istanza di R1 è già posseduta da P2, quindi P1 si sospende.
- **P2**: ha richiesto e ottenuto un'istanza di R2; ha lanciato una wait per assumere R3, ma l'unica istanza di R3 è posseduta da P2, quindi P2 si pone in attesa.
- R4 non ha istanze controllate da processi, né tantomeno vi sono richieste da parte dei processi.
- Non abbiamo un ciclo: il deadlock non è possibile.
- **Consideriamo adesso la freccia tratteggiata in rosso**
  - Si manifesta la condizione di attesa circolare, ma ciò non basta per dire se c'è deadlock o meno.
  - Tutti i processi sono tutti in attesa (hanno archi in uscita), tutte le istanze delle risorse coinvolte sono possedute. Posso affermare che il sistema è in deadlock rispetto all'insieme di processi e risorse da noi indicato.

6.7.5.2 Esempio di grafo con ciclo ma senza deadlock



Il processo P2 detiene un'istanza del tipo di risorsa R1.

Il processo P1 detiene un'istanza del tipo di risorsa R2, e ha richiesto un'istanza del tipo di risorsa R1. Quest'ultima ha entrambe le istanze già detenute da altri processi, quindi P1 si sospende.

Il processo P3 detiene un'istanza del tipo di risorsa R1, e ha richiesto un'istanza del tipo di risorsa R2. Quest'ultima ha entrambe le istanze già detenute da altri processi, quindi P3 si sospende.

Il processo P4 detiene un'istanza del tipo di risorsa R2.

- Abbiamo i processi P1, P2, P3 e P4. Abbiamo le risorse R1 ed R2, ciascuna con due istanze.
- La condizione di attesa circolare è soddisfatta: abbiamo un ciclo R2-P1-R1-P3-R2.
- **C'è il deadlock?**
  - P1 e P3 sono i due processi coinvolti nel ciclo, entrambi in attesa.
  - R1 ed R2 hanno più istanze, una di queste due (per ciascuna) è detenuta da processi non in attesa.
  - Siamo certi che P2 e P4 rilasceranno la risorsa a un certo punto: quando P2 rilascia R1 l'arco da P1 ad R1 si inverte (diventa possesso), dunque il ciclo viene meno. Non abbiamo deadlock.

6.7.5.3 Conclusioni dagli esempi: quando si ha deadlock e quando no

1. Se il grafo non contiene cicli sicuramente non ho deadlock (una delle CN è falsa).
2. La condizione di attesa ciclica diventa condizione sufficiente per il deadlock **se tutte le risorse coinvolte sono a singola istanza**.

6.7.6 Metodologie per gestire il *deadlock*

Il sistema può adottare uno dei seguenti approcci per trattare il deadlock:

1. ***deadlock prevention* (prevenzione statica) e *deadlock avoidance* (prevenzione dinamica)**.
   Assicurarsi che il sistema non entri mai in uno stato di deadlock. Nella prima ci assicuriamo che una delle CN dette prima non si valida, nella seconda simuliamo lo stato in cui entrerebbe il sistema operativo concedendo una risorsa, e la concediamo solo se possiamo parlare di *stato sicuro*. Nella seconda ci servono algoritmi per poter simulare lo stato e arrivare a una conclusione.
2. ***deadlock detection***
   Permettere al sistema di entrare in uno stato di deadlock e quindi di recuperare. Introdurremo algoritmi in grado di individuare il deadlock "a giochi ormai fatti".
3. Ignorare il problema e pretendere che i deadlocks non si manifestino mai nel sistema.

L'ultimo approccio è quello adottato dalla maggior parte dei sistemi operativi, incluso UNIX.

6.7.6.1 Deadlock prevention (prevenzione statica)

La *deadlock prevention* consiste nell'assicurarsi che almeno una delle quattro condizioni necessarie per il deadlock non sia valida. Chiediamoci per ogni condizione cosa possiamo fare affinchè questa diventi falsa.

Mutua esclusione	Come già anticipato la mutua esclusione è irrinunciabile: risorse non condivisibili, per esempio strutture dati sensibili, non possono fare a meno della mutua esclusione. Esistono particolari sistemi dove il programmatore ha la facoltà di dire che non ci saranno problemi di mutua esclusione su certe variabili (attenzione, una grossa responsabilità).
Hold and wait	Dobbiamo garantire che quando un processo richiede una risorsa questo non ne tenga altre con sé. Una soluzione potrebbe essere far richiedere al processo, all'inizio, tutte le risorse di cui ha bisogno. La cosa non è semplice, ma soprattutto è dispendiosa: - è necessaria una procedura appositamente dedicata; - il programmatore è obbligato a conoscere in anticipo tutte le risorse necessarie al processo; - L'uso delle risorse è basso (*starvation*), rimangono assegnate al processo anche quando non vengono utilizzate.
no preemption	Il meccanismo della *preemption* può salvarci dal deadlock: quello che dobbiamo fare è rompere il deadlock revocando risorse ad alcuni processi coinvolti nel deadlock. Queste risorse saranno rese disponibili ad altri processi. Complicato perché dobbiamo garantire il recupero successivo delle risorse e il salvataggio di uno stato delle risorse.
Attesa circolare	Per evitare cicli in fase statica dobbiamo imporre una politica che regoli l'acquisizione delle risorse da parte dei processi. Immaginiamo una gerarchia di livelli: - i processi hanno un ordine; - se un processo possiede una risorsa appartenente al livello d nel seguito può richiedere solo risorse di livello $k > d$. L'acquisizione delle risorse in un ordine prestabilito comporta un utilizzo inefficiente delle risorse (che ad esempio vengono acquisite in anticipo e non usate subito).

6.7.6.2 Deadlock avoidance (prevenzione dinamica)

La *deadlock avoidance* consiste nell'uso di algoritmi che verificano il manifestarsi di deadlock nel caso di concessione di una risorsa a un processo. Per fare ciò dobbiamo conoscere a priori alcune informazioni sui processi.

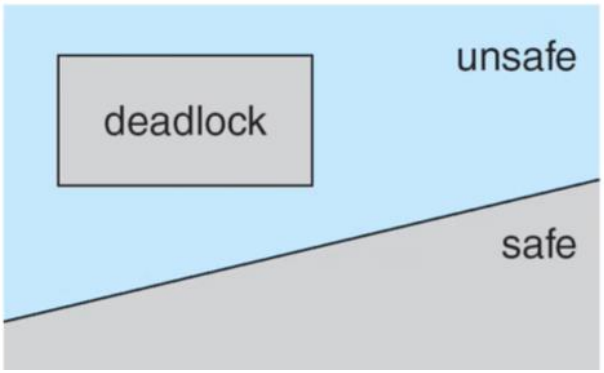
- Richiedo che ciascun processo/thread dichiari il numero massimo di istanze per ogni tipo di risorsa che intende utilizzare (per esempio quante *open file* saranno lanciate…).
- Ragionamento:
  - Devo poter descrivere ad ogni istante lo stato di allocazione delle risorse nel sistema.
  - Abbiamo introdotto poco fa il grafo di allocazione delle risorse: lo utilizzo per verificare runtime lo **stato ipotetico** in cui si troverebbe il sistema nel caso venisse concessa la risorsa al processo.
  - Se lo stato ipotetico è uno **stato sicuro** allora possiamo concedere la risorsa.

Lo stato è descritto non solo dal grafo, ma anche da ulteriori variabili che tengono conto del numero massimo di risorse che il processo chiederà, delle risorse già allocate (e che quindi possiede) e il numero di risorse di cui ha ancora bisogno (la differenza tra i due valori precedenti).

- **Definizione di stato sicuro**.
  Un sistema si trova in uno stato sicuro se esiste una sequenza di istruzioni di tutti i processi tale che un processo generico P, dopo aver ottenuto e usato tutte le risorse di cui ha bisogno, termina e rilascia tutte le risorse rendendole disponibili agli altri processi. Se le richieste di un particolare processo non sono immediatamente disponibili allora questo si pone in attesa finché il processo antecedente nella sequenza non avrà rilasciato le relative risorse in suo possesso.

- **Quindi**:
  - Se il sistema si trova in uno stato sicuro sicuramente non avrò deadlock (una CN diventa falsa)
  - Se il sistema non si trova in uno stato sicuro **potrei** avere deadlock.

  La prevenzione dinamica non riesce a distinguere l'*unsafe* senza deadlock dall'*unsafe* con blocco, dunque il confine che dobbiamo considerare è quello tra *safe* e *unsafe*.



Morale della favola: dobbiamo assicurarci che il sistema non entri mai in uno stato non sicuro.

Nell'introduzione dei seguenti algoritmi distinguiamo due casi: quello in cui la condizione di attesa ciclica è necessaria e sufficiente e quello in cui la stessa è solo necessaria.

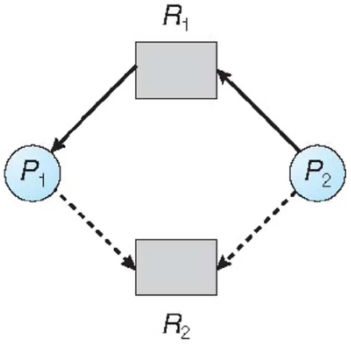
6.7.6.2.1 Algoritmo di avoidance per risorse a singola istanza

Ricordiamo quanto detto: se si ha un ciclo e tutte le risorse hanno una sola istanza allora l'attesa circolare è condizione sufficiente.

- **Nuovo arco**.
  Introduciamo un nuovo arco: il *claim edge*, che va da un processo a una risorsa ed è raffigurato tratteggiato. L'arco indica che il processo ha intenzione di richiedere la risorsa, ma non l'ha ancora fatto.
- Come faccio a disegnare i claim edge? Devo sapere quante risorse richiederà ogni processo.
- Durante l'esecuzione del sistema i *claim edge* si trasformano: diventa arco di richiesta quando il processo effettivamente lancia la primitiva di richiesta, infine l'arco di richiesta diventa un arco di possesso quando la risorsa è libera e viene allocata al processo richiedente. Se la risorsa non è disponibile l'arco di richiesta diventa arco di attesa.

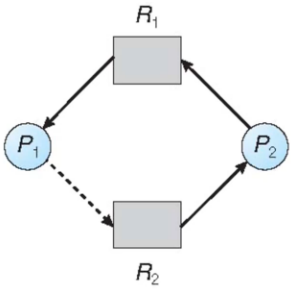
Esempio.

- Abbiamo due processi (P1 e P2) e due risorse (R1 e R2).
- Entrambi i processi richiederanno sia R1 che R2 (immaginiamo di avere all'inizio quattro *claim edge*, due che escono da P1 e due che escono da P2).
- Nell'istante rappresentato abbiamo P1, che ha richiesto R1 prima di P2. R1 era libera, quindi il claim edge è diventato arco di possesso. Successivamente P2 ha fatto la richiesta su R1, ma ha trovato R1 occupato e quindi l'arco è diventato arco di attesa. Gli altri due claim edge sono in attesa dell'evoluzione futura.



- **Cosa succede andando oltre? Potrei creare un ciclo e finire in uno stato *unsafe*?**
  P2 è sospeso, quindi c'è solo P1. R2 è libera quindi P1 trasformerà il suo claim, prima o poi, in possesso. Se questo è lo stato posso stare tranquillo: sono e rimarrò in uno stato safe.

- Nell'algoritmo dobbiamo analizzare lo stato ipotetico (in un certo senso significa disegnare il grafo con i claim edge) e decidere sulla base di quello se assegnare o meno la risorsa. In questo caso non ci sono problemi e la risorsa viene assegnata al processo.



- **Quando si rileva uno stato unsafe?**
  Supponiamo che dopo la wait di P1 su R1 si abbia richiesta di P2 su R2. R2 è libera l'arco di claim si trasforma in arco di possesso (va da R2 a P2) e si introduce un ciclo. Lo stato è non sicuro, quindi blocco il processo P2 e nego la risorsa.

6.7.6.2.2 Algoritmo di *avoidance* per risorse a plurime istanze (c.d. algoritmo dei banchieri, con esempio)

In presenza di risorse a plurime istanze ricorriamo all'algoritmo dei banchieri. In questo caso l'attesa circolare è solo condizione necessaria, quindi l'algoritmo precedente non basta più.

- Ogni processo deve dichiarare a priori il numero massimo di istanze di risorse di cui ha bisogno per completarsi.
- Ogni processo che richiede una risorsa può ritrovarsi in una situazione di attesa, anche se la risorsa è libera (potrebbe non essere assegnata per evitare il problema del ciclo).
- Consideriamo n come numero dei processi ed m come numero dei tipi di risorse.

- **Sono necessarie delle strutture dati**:
  - Una matrice di dimensioni $n \times m$ con cui indichiamo il numero massimo di istanze che un processo i −esimo richiederà del tipo di risorsa j −esimo
    $$Max[i,j] = k$$
  - Un vettore di lunghezza m, contenente il numero k di risorse ancora disponibili (si decrementa e incrementa quando istanze vengono acquisite o rilasciate).
    $$\text{Available}[j] = k$$
  - Una matrice di dimensioni $n \times m$ con cui indichiamo il numero di istanze di risorse j −esime attualmente allocate per il processo i −esimo
    $$\texttt{Allocation}[i,j] = k$$
  - Un ulteriore matrice ottenuta con la differenza tra gli elementi delle due matrici precedenti (il numero di istanze di risorse ancora necessarie per completare il processo)
    $$\texttt{Need}[i,j] = \texttt{Max}[i,j] - \texttt{Allocation}[i,j]$$

- **Algoritmo**:
  - **Vettore contenente le richieste da parte di un processo, per ogni tipo di risorsa**.
    Considero un vettore Request relativo al processo P_i. Nel caso in cui la tipologia di risorsa j −esima sia richiesta si otterrà un intero $k > 0$.
    $$\texttt{Request}_i[j] = k$$
  - **Verifica del rispetto delle ipotesi poste nella matrice Need**
    Se $\texttt{Request}_i[j] < \texttt{Need}_i$
    Passo allo step successivo. Altrimenti devo sollevare un errore, poiché il processo ha superato in numero di richieste il massimo che si era posto
  - **Verifica della disponibilità delle risorse richieste, sospensione se le istanze non soddisfano**.
    Se $\texttt{Request}_i \leq \texttt{Available}_i$
    Passo allo step successivo. Se il numero di richieste è superiore al numero di risorse effettivamente disponibili allora il processo P_i si pone in attesa.
  - **Simulazione dello stato ideale**.
    Pretendo di allocare le risorse richieste al processo P_i generando lo stato ideale
    $$\texttt{Available} = \texttt{Available} - \texttt{Request}_i$$
    $$\texttt{Allocation}_i = \texttt{Allocation}_i + \texttt{Request}_i \quad \forall i$$
    $$\texttt{Need}_i = \texttt{Need}_i - \texttt{Request}_i \quad \forall i$$
  - **Verificare se lo stato ideale è anche stato sicuro**.
    A questo punto applico la procedura per la verifica della sicurezza, che restituisce 0 o 1:
    - se lo stato ideale è *sicuro* la risorsa viene allocata al processo;
    - se lo stato è *unsafe* il processo deve aspettare, e lo stato di allocazione originario viene ripristinato.

- **Procedura per la verifica della sicurezza** (applichiamo ogni volta che si deve verificare lo stato sicuro):
  - Inizializziamo le strutture dati necessarie.
    - **Vettore di supporto**. Un vettore temporaneo `Work` tale che
      $$Work = Available$$
    - Un vettore `Finish` che per ogni processo i −esimo mi dice se sono terminati o meno.
      $$Finish[i] = false \quad \forall i = 0, \dots, n-1$$
      Attenzione, a false si mettono anche gli elementi di finish che hanno need diverso da zero.
  - **Condizioni richieste**. Faccio un ciclo con cui individuo il processo i −esimo che rispetta quanto segue
    $$\texttt{Finish}[i] = false$$
    $$\texttt{Need}_i \leq \texttt{Work}$$
    Se non esiste un processo del genere vado all'ultimo punto
  - Per ogni processo i −esimo pongo quanto segue (lo mettiamo nelle condizioni per terminare):
    $$\texttt{Work} = \texttt{Work} + \texttt{Allocation}_i$$
    $$\texttt{Finish}[i] = true$$
    e torno allo step precedente.
  - Se $\texttt{Finish}[i] == true$ per tutti i processi i −esimi allora il sistema si trova in uno stato sicuro.

Esempio. Vista la verbosità/complessità dell'algoritmo poniamo subito un esempio di applicazione. Abbiamo i processi P_0, P_1, P_2, P_3, P_4 (cinque processi) e tre tipi di risorse: A (10 istanze), B (5 istanze), C (7 istanze). Inizialmente abbiamo il seguente snapshot (lo snapshot è rappresentato dalle strutture dati dette all'inizio):

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Per prima cosa calcoliamo l'altra struttura dati necessaria (vediamo le necessità che devono essere soddisfatte affinchè i vari processi possano terminare):
  $$\texttt{Need}[i,j] = \texttt{Max}[i,j] - \texttt{Allocation}[i,j] = \begin{bmatrix} 7-0 & 5-1 & 3-0 \\ 3-2 & 2-0 & 2-0 \\ 9-3 & 0-0 & 2-2 \\ 2-2 & 2-1 & 2-1 \\ 4-0 & 3-0 & 3-2 \end{bmatrix} = \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$
  Ovviamente tutti i processi hanno necessità pendenti, nessun processo in questo momento è terminato. Applichiamo la procedura per la verifica della sicurezza dello stato: cerchiamo tutti i processi non ancora terminati e cerchiamo di soddisfarne le necessità.
  - P_0 chiede troppo rispetto alle risorse disponibili, mentre P_1 può essere coperto. Il processo P_1 termina in un tempo finito e le disponibilità diventano [5,3, 2] (P_1 termina e restituisce ciò che ha allocato).
  - P_2 non può essere gestito (richiede sei istanze della prima), ma P_3 può essere gestito. Quando P_3 termina questo restituisce le sue allocazioni: le disponibilità diventano [7, 4, 3].
  - Gestiamo il processo P_4: questo termina e le disponibilità diventano [7, 4, 5].
  - Torno ad esaminare P_0: adesso può essere terminato! Con le risorse rilasciate le disponibilità diventano [7, 5, 5].
  - Stavolta P_2 può essere terminato! Con le risorse rilasciate otteniamo come disponibilità [10, 5, 7]
- Adesso entra in azione l'algoritmo vero e proprio. Supponiamo di avere una richiesta di [1, 0, 2] (vettore *Request*) da parte del processo P_1.
  - Partiamo da uno stato sicuro (lo abbiamo appena visto).
  - Per prima cosa verifichiamo che le richieste siano minori rispetto alle disponibilità
    $$[1,0,2] \leq [3,3,2]$$
    La condizione è soddisfatta!
  - Andiamo a creare il cosiddetto stato ideale: immaginiamo di soddisfare la richiesta.
    - Available: [3,3,2] -> [2, 3, 0] (Abbiamo allocato le risorse a P_1)
    - Need_1: [1, 2, 2] -> [0, 2, 0]

        - Allocation_1 : [2, 0, 0] -> [3, 0, 2]
    - Come prima verifichiamo che lo stato sia safe.
        - Il primo processo che può terminare è P_1: quando questo termina le disponibilità diventano [5, 3, 2]
        - Consideriamo successivamente P_3: quando termina le disponibilità diventano [7, 4, 3]
        - Adesso possiamo far terminare il processo P_0: le disponibilità diventano [7, 5, 3]
        - Possiamo far terminare P_2: a seguito le disponibilità diventano [10, 5, 5]

      Tutti i processi sono terminati, dunque esiste una sequenza di processi per cui si ha uno stato sicuro. Possiamo confermare l'allocazione richiesta da P_1.
- Altra richiesta: P_4 richiede [3, 3, 0]. Possiamo acconsentire alla richiesta? No, perché non c'è disponibilità di risorse.
- Altra richiesta: P_0 richiede [0, 2,0]. Possiamo acconsentire alla richiesta? Il fabbisogno c'è, calcoliamo come prima che lo stato sia safe.

6.7.6.3 Deadlock detection

Nella deadlock detection permettiamo al sistema di entrare in uno stato di deadlock. Nella deadlock detection consideriamo:

- algoritmi che individuano i deadlock (eseguiti periodicamente, più avanti è spiegato quando;
- schemi di recupero (a cui accenniamo velocemente solo alla fine).

Ci basiamo nuovamente sul grafo di allocazione delle risorse.

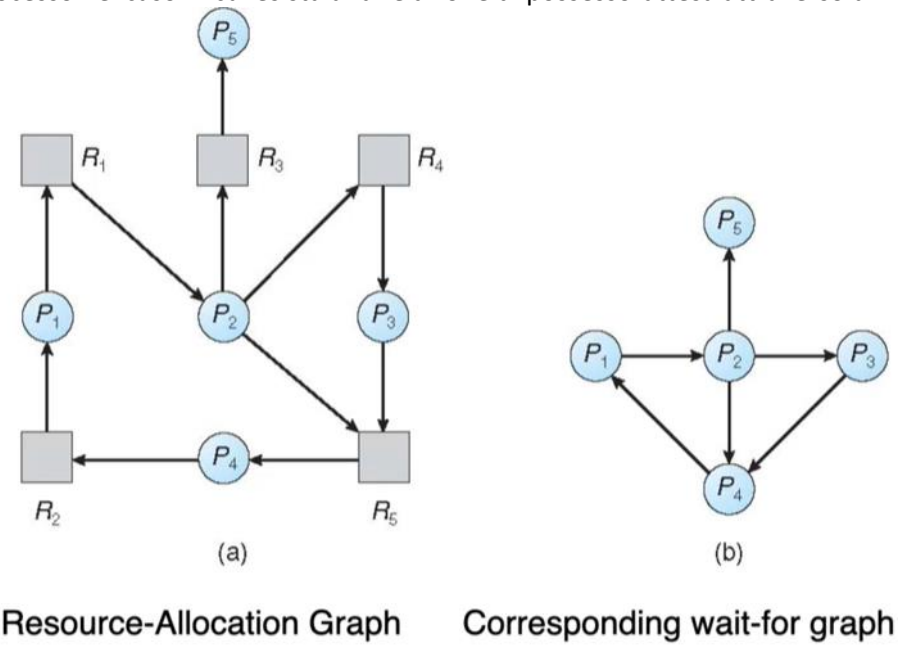
6.7.6.3.1 Algoritmo wait-for-graph (risorse a singole istanze, equivalente del precedente)

Abbiamo risorse a singola istanza. Per ridurre il numero di nodi da analizzare si studia un altro nodo: il *wait-for graph*, dove i nodi sono solo processi. L'arco rappresenta uno stato di attesa

$$P_i \rightarrow P_j \text{ se Pi sta aspettando Pj}$$

Risulta immediato che si ha un deadlock in presenza di un ciclo.

Esempio di corrispondenza. La trasformazione si ottiene con un algoritmo che percorre il grafico a sinistra e collega tutti i nodi di tipo processo nel caso in cui esista una relazione di possesso-attesa attraverso un'istanza di risorsa.



6.7.6.3.2 Algoritmo con risorse a plurime istanze (con approccio simile al banchiere)

Abbiamo risorse a plurime istanze. Si adotta un approccio simile a quello del banchiere. Abbiamo le seguenti strutture dati:

- ***Available***, un vettore di lunghezza m che indica il numero di istanze disponibili per ogni risorsa.
- ***Allocation***, una matrice $n \times m$ che indica il numero di risorse di ogni tipo allocate per ciascun processo.
- ***Request***, una matrice $n \times m$ che indica il numero di richieste relative ad ogni processo per ogni tipo di risorsa. Diremo che:

$$\texttt{Request}[i][j] = k$$

Il processo i −esimo sta richiedendo k istanze di risorse j −esime. Non è ciò che immaginiamo avvenga nel futuro, ma ciò che è effettivamente avvenuto: il processo ha invocato una primitiva richiedendo delle risorse.

Rispetto al banchiere l'algoritmo è più semplice perché non devo verificare se lo stato è sicuro.

1. Inizializziamo le strutture dati. Prendiamo il solito vettore ausiliario Work, e poniamolo IL contenuto di available
$$\texttt{Work} = \texttt{Available}$$
Inizializziamo anche Finish, altro vettore con cui indichiamo se un processo ha finito o no. Per ogni i relativo a processi che presentano richieste di allocazioni ($\texttt{Allocation}_i \neq 0$)
$$\texttt{Finish}[i] = false$$
Altrimenti poniamo $\texttt{Finish}[i] = true$
2. Vogliamo trovare un indice i tale che
$$\texttt{Finish}[i] == false$$
$$\texttt{Request}_i \leq \texttt{Work}$$
Se l'elemento non esiste saltiamo direttamente al quarto step.
3. Interveniamo nelle strutture dati
$$\text{Work} = \text{Work} + \text{Allocation}_i$$
$$\text{Finish}[i] = true$$
Torno al secondo step.
4. Se nell'array troviamo dei valori falsi
$$\text{Finish}[i] = false$$
Allora il sistema si trova in uno stato di deadlock. I processi i −esimi sono i processi incriminati, che a causa del deadlock non saranno mai completati.

6.7.6.3.3 Quanto frequentemente eseguiamo l'algoritmo?

- Una prima risposta potrebbe essere "tutte le volte che un processo si sospende". Si tenga conto che altamente probabile la presenza di un deadlock se un processo si sospende e possediamo già delle risorse.
- Purtroppo, l'esecuzione dell'algoritmo non è irrilevante, dunque un'esecuzione troppo frequente comporta un aumento di overhead. Chiediamoci: ogni quanto si manifesta un deadlock? Supponiamo che per calcoli a noi ignoti si abbia un deadlock in media ogni ora. Sfruttando questo dato lancio l'algoritmo una volta ogni ora.
- Basta quanto detto prima? No, potrei avere sempre casi in cui aumentiamo l'overhead inutilmente.
- **Dobbiamo introdurre un ulteriore criterio: l'uso della CPU**. Se si ha un uso della CPU, ad esempio, sceso sotto il 40% ed è passata un'ora allora eseguo l'algoritmo.

6.7.6.3.4 Recovery from deadlock (accenni)

Gli approcci possibili sono due:

- terminare i processi in deadlock;
- revocare le risorse in mano ai processi (resource preemption).

Primo approccio. Abortire tutti i processi è chiaramente distruttivo: quello che facciamo è abortire un processo alla volta, faccio questo finchè il deadlock non viene meno. Sulla base di quali criteri abortisco un processo alla volta?

- Priorità dei processi (abortisco quelli con minore priorità
- Da quanto tempo il processo è in esecuzione e quanto manca per completare il processo (attenzione, il secondo parametro è una semplice stima).
- Risorse già utilizzate dal processo (molte? stabilisco una soglia e se la supero non abortisco, se il numero di risorse già usate è alto il processo potrebbe essere vicino a completamento)
- Risorse che deve ancora utilizzare un processo.

Secondo approccio. Il secondo approccio consiste nel fare resource preemption (cioè revocare le risorse in mano a un processo). Per prima cosa dobbiamo identificare una vittima (non perderemo tempo su come individuare una vittima).

- La cosa è abbastanza costosa a causa della procedura di rollback (devo fare in modo che i processi si pongano in uno stato sicuro, overhead alle stelle cit.).
- Altro problema potrebbe essere la starvation: se eseguo l'algoritmo troppo frequentemente le vittime scelte potrebbero essere sempre gli stessi processi.

7 Gestione della memoria principale

7.1 Introduzione

7.1.1 Promemoria: necessità di una memoria cache

Perché abbiamo bisogno di un ulteriore componente per gestire la memoria? Con gestione della memoria alludiamo alla memoria principale, che è volatile e tipicamente implementata mediante banchi di memoria RAM (con un sottoinsieme implementato con tecnologia ROM per l'esecuzione del programma bootstrap).
Abbiamo già osservato a Calcolatori elettronici che la memoria richiede tempi di trasferimento maggiori di qualche ordine di grandezza rispetto ai tipici tempi di esecuzione della CPU. Abbiamo una differenza di tempo che deve essere gestita adeguatamente, a tal proposito abbiamo introdotto livelli di memoria intermedi: le memorie cache, che presentano una velocità più elevata anche se con una capacità minore rispetto alla memoria principale.

Trasparenza della cache. La gestione della cache è trasparente: non è visibile al programma in esecuzione, ma è gestita dal sistema operativo attraverso hardware dedicato. Ricordiamoci la visione che abbiamo della memoria principale: un array di elementi dove non indichiamo il tipo ma solo la dimensione.

7.1.2 La memoria virtuale

La memoria virtuale non è altro che un sistema costituito da memorie, meccanismi e politiche che permettono di avere una visione più estesa della memoria principale. Ricordiamo che nel modello di Von Neumann è necessario che tutti i processi siano caricati in memoria principale: chiaramente la dimensione della memoria fisica non è sufficiente. Per aumentare "virtualmente" la capacità della memoria principale ricorriamo all'uso di una memoria persistente, tipicamente l'hard disk.

Meccanismo di swapping. Il meccanismo di swapping consiste nella possibilità di depositare temporaneamente gli spazi di memoria dei processi nell'hard disk, in attesa di disponibilità sulla memoria principale. Lo swapping, vedremo, si distingue in *swap-in* e *swap-out* (rispettivamente passaggio di un processo da *swap area* a memoria principale e viceversa).

7.1.3 Analogie e differenze tra gestione della CPU e gestione della memoria, conseguenze

Il sistema operativo coordina il regolare svolgimento dei processi, garantendo a ciascuno il corretto accesso alle risorse di cui ha bisogno. Il sistema operativo fornisce gli strumenti per coordinare gli accessi alle risorse da parte dei processi secondo precise politiche di allocazione. Dobbiamo considerare che il processo perde di significato se non sono a disposizione le seguenti risorse:

- la CPU;
- un'area di memoria principale.

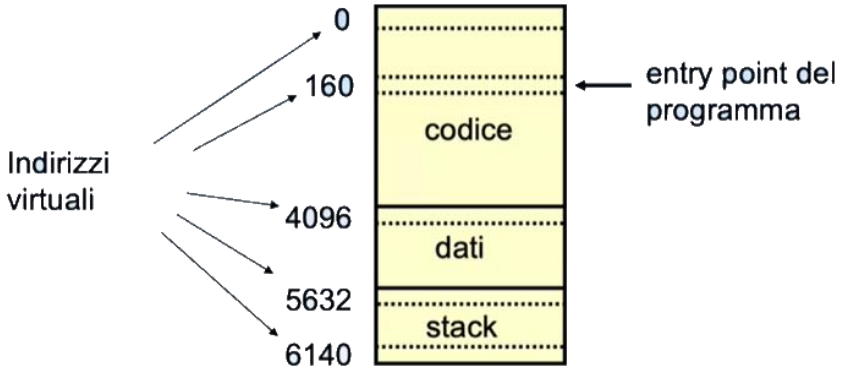
Partendo da questa osservazione vogliamo identificare analogie e differenze tra gestione della CPU e gestione della memoria.

Analogie
- Sono entrambe risorse necessarie per l'esecuzione di un processo. - La loro allocazione è implicita, cioè non è scritta nei programmi (salvo allocazione dinamica). - Contrariamente a tutte le altre risorse non ha senso dire che il processo richiede la CPU e la memoria. Questo perché richiedere una risorsa implica che il processo sia già in esecuzione, ma se il processo richiede la CPU significa che non ce l'ha e che quindi non può essere in esecuzione. Stessa cosa la memoria: il processo non può richiede la memoria perché se non ce l'ha significa che non può essere in esecuzione. - In entrambi i casi è utile prevedere la *preemption*: non è vitale, ma rende la gestione più flessibile.

L'ultima questione viene risolta attraverso la virtualizzazione della CPU: quello che facciamo al momento della creazione del processo (quando questo ancora non è associato alla CPU) è di creare un **processore virtuale** (l'array contesto del relativo descrittore di processo, che contiene i valori di tutti i registri del processore virtuale). A questo punto è il sistema che alloca la CPU al processo, e non viceversa (sulla base dell'algoritmo di scheduling). Attraverso il meccanismo del cambio di contesto (parte del kernel e mai modificato, se non a causa di bug) trasferiamo il contesto dal relativo descrittore di processo al processore fisico.

Lo stesso approccio viene adottato per la memoria principale: al momento della creazione del processo si associa ad esso una **memoria virtuale** dove viene caricato il programma da eseguire. Servono ulteriori strutture dati per descrivere lo stato della memoria virtuale. La memoria virtuale è uno spazio di indirizzamento contiguo che possiamo concepire come un vettore di elementi (dove il tipo è in sostanza la dimensione in byte degli elementi).

L'immagine del processo può essere anche più grande della memoria fisica (quindi alloco un po' alla volta le partizioni in base alle richieste del processore).



Ricordarsi che

Compito del sistema operativo è allocare la memoria reale per fornire supporto alle memorie virtuali.

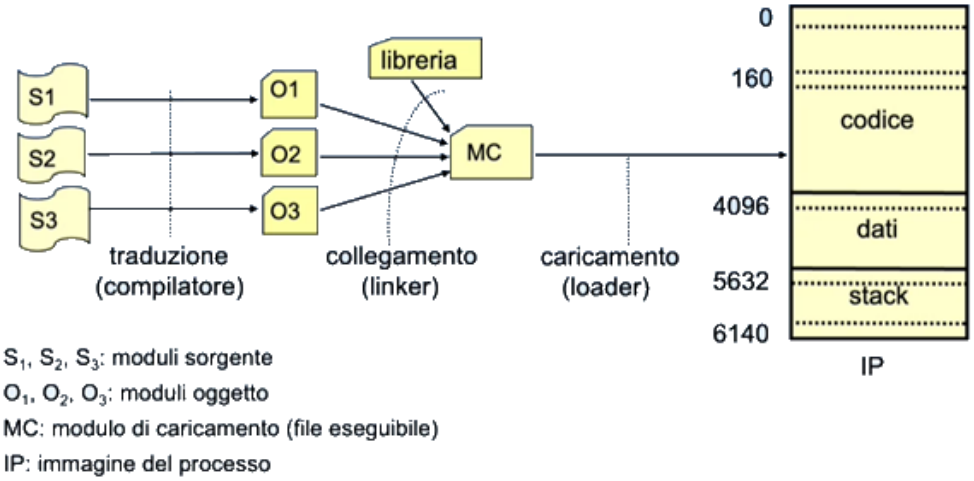
Differenze

- Le due risorse hanno una diversa struttura fisica.
  o Nel caso della CPU intendiamo una struttura dati che rappresenta lo stato della risorsa fisica (la CPU) quando questa non è allocata al processo. La struttura dati deve riflettere lo stato del processore, e deve permettere il recupero di quanto già fatto dopo la sospensione del processo.
  o Nel caso della memoria principale abbiamo bisogno di una memoria diversa da quella principale, appunto la swap area dell'*hard disk*).
- La CPU si assegna per intero, la memoria può essere assegnata per parti a un processo (parti che si suppongono contigue).

Emerge il problema della protezione: abbiamo più processi che hanno allocato uno spazio di indirizzamento (o parte di esso) in una memoria. I processi hanno accesso alla memoria in lettura e scrittura, dobbiamo fare in modo che i processi non accedano a ciò che non gli appartiene. In tutto questo è necessario permettere ai processi di condividere codice e dati con altri processi. Teniamo anche conto che potrei avere più istanze dello stesso programma (processi diversi che eseguono lo stesso programma, e quindi accedono alla stessa partizione di memoria).

7.2 Aspetti caratterizzanti la gestione della memoria

7.2.1 Preparazione di un programma, spazio di indirizzamento virtuale e fisico



Sappiamo da Calcolatori elettronici come un programma viene preparato per l'esecuzione:

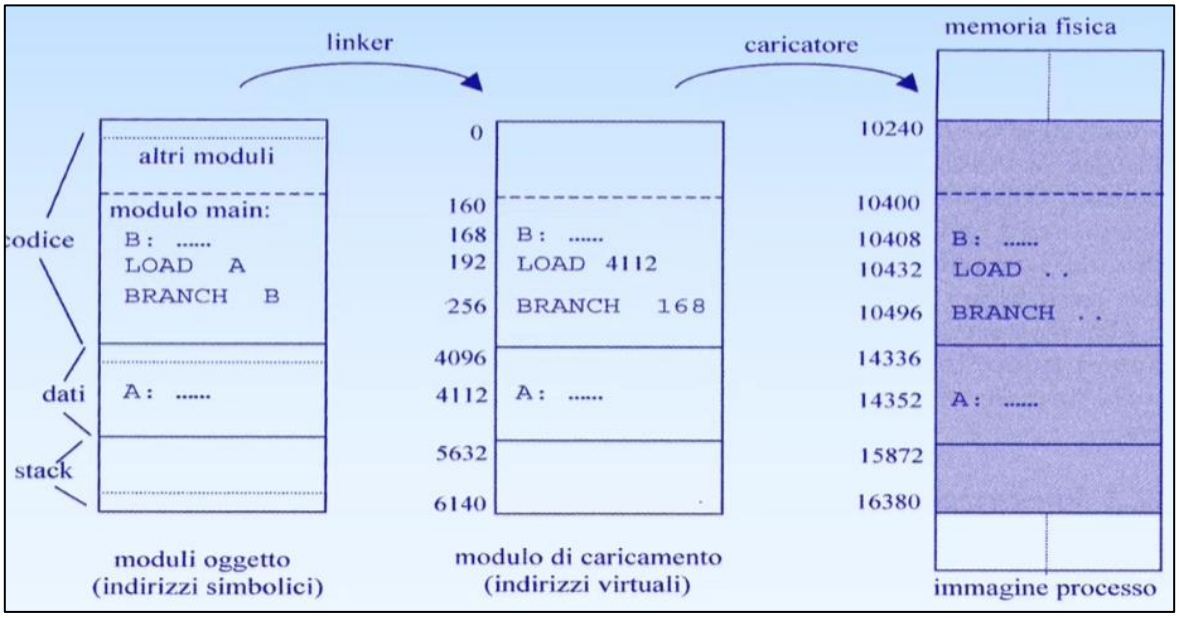
- si parte da *moduli sorgente* che contengono il codice;
- ciascun *modulo sorgente* viene passato dal compilatore ottenendo un corrispondente *modulo oggetto* (non tutte le informazioni poste nel modulo sorgente possono essere tradotte in linguaggio macchina poiché si hanno riferimenti simbolici a informazioni contenute in altri moduli compilati separatamente);
- si uniscono i *moduli oggetti* dando origine al modulo di caricamento (il file eseguibile, a questo punto vengono meno i riferimenti simbolici).

Contribuiscono al modulo di caricamento anche le librerie (anch'esse già viste a Calcolatori elettronici).

Il modulo di caricamento è una descrizione di quello che sarà il processo. Deve essere chiaro fin da subito l'esistenza di due spazi di indirizzamento distinti: quello virtuale, che fa riferimento all'immagine del processo; quello fisico, che fa riferimento alla memoria fisica (come è fatta, quanto è grande, a partire da quale indirizzo viene allocata l'immagine del processo).

7.2.1.1 Premessa alla rilocazione: caricatore

Nella figura si ripropone il modulo di caricamento facendo riferimento a un processo di esempio, che richiede uno spazio virtuale avente una certa dimensione



- Abbiamo bisogno di 6140 byte per rappresentare tutto ciò che è contenuto nell'immagine del processo.
- Si pone enfasi alla sezione del codice, che inizia all'indirizzo virtuale 160 e termina all'indirizzo 4095. In modo contiguo si ha la sezione dedicata ai dati, che inizia all'indirizzo virtuale 4096 e termina all'indirizzo 5631. Infine abbiamo lo stack, che va dall'indirizzo virtuale 5632 all'indirizzo 6140.
- Abbiamo un'etichetta simbolica B, che a seguito di compilazione e linking viene trasformata in un indirizzo dello spazio di indirizzamento virtuale, appartenente alla sezione dati. In questo caso si fa riferimento all'indirizzo 168. Il valore posto a questo indirizzo è operando dell'istruzione alla riga 256.

Dobbiamo caricare il modulo di caricamento, cioè dobbiamo allocare la memoria virtuale nella memoria fisica. L'operazione è svolta dal cosiddetto *caricatore*. Un caricatore è detto ***rilocante*** se svolge rilocazione statica.

7.2.2 Rilocazione statica (*caricatore rilocante*)

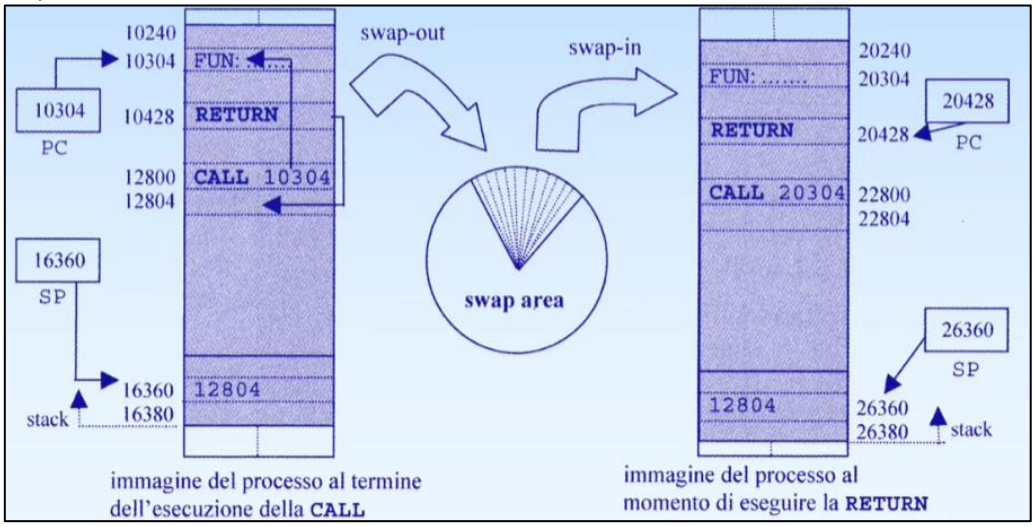
La rilocazione statica consiste nel tradurre tutti gli indirizzi del modulo caricamento in indirizzi fisici, in conseguenza della posizione in cui l'immagine del processo verrà caricata in memoria. Andiamo a definire una *funzione di rilocazione* $y = f(x)$ dove x è l'indirizzo virtuale ed y il corrispondente indirizzo fisico.

- **Esempio**: supponiamo di porre la partizione a partire dall'indirizzo fisico 10240. Questo significa che l'indirizzo virtuale 160 dovrà diventare l'indirizzo fisico 10400: l'unica cosa che facciamo è sommare una costante (dipendente appunto dalla posizione della partizione). La funzione di rilocazione è la seguente
$$y(x) = x + 10240$$
- Nei sistemi con rilocazione statica esiste un modulo detto ***caricatore rilocante*** (come già anticipato) che:
  - **trova una partizione di memoria fisica abbastanza grande per contenere la memoria virtuale** (tenerselo a mente per capire la differenza tra memoria unica e memoria segmentata);
  - calcola l'indirizzo di partenza;
  - calcola tutti gli indirizzi successivi applicando una funzione di rilocazione (in questo caso la somma) che trasforma lo spazio di indirizzamento virtuale in uno spazio di indirizzamento fisico.

  Nella rilocazione statica gli indirizzi virtuali vengono rilocati nei corrispondenti indirizzi fisici prima dell'esecuzione del processo. La traduzione già fatta e una funzione di rilocazione semplicissima comportano un overhead praticamente nullo. I registri PC e SP verranno inizializzati con indirizzi fisici, rispettivamente l'indirizzo della prima istruzione da eseguire nel codice del processo e l'indirizzo della base della pila.
- **Domanda: dove si trova il modulo di caricamento (file eseguibile)?**
  Nella memoria persistente. Poniamo il modulo in memoria principale solo quando dovrà essere eseguito.

7.2.2.1 Meccanismo di swapping e rilocazione statica

Cosa succede se non è disponibile spazio nella memoria principale? Abbiamo già citato il meccanismo di swapping, con *swap-in* e *swap-out*



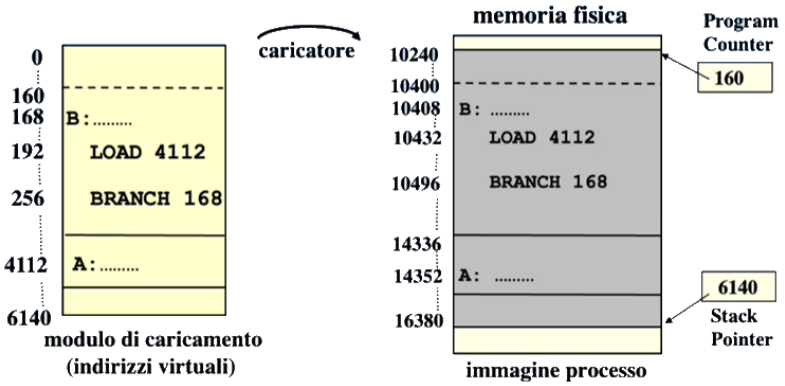
- **Memorizzazione dell'indirizzo nella pila**
  Nell'esecuzione della CALL l'indirizzo di ritorno viene salvato in pila, successivamente viene posto nel PC l'indirizzo posto come operando nella CALL. Nell'esecuzione di RETURN recuperiamo l'indirizzo della pila e torniamo al punto dove abbiamo chiamato la CALL (l'istruzione successiva).
- ***swap-out***
  Supponiamo che per qualche motivo avvenga swap-out: l'immagine caricata in memoria fisica viene caricata nella swap-area dell'hard disk. Nel momento in cui il processo subisce lo swap-out si perde la possibilità di schedularlo. Si distinguono più stati in sistemi col meccanismo: *pronto* e *pronto swapped*, *bloccato* e *bloccato swapped* (vediamo più avanti).
- ***swap-in***
  A seguito dello swap-out faremo, prima o poi, swap-in: un algoritmo deciderà quale processo ricaricare in memoria. Supponiamo di rimettere in memoria lo stesso processo. Quando io vado ad allocare uno spazio virtuale nella memoria fisica devo trovare, nuovamente, una partizione disponibile: non è affatto detto che sia la stessa di prima. Dobbiamo fare nuovamente la rilocazione statica (ricalcolare tutti gli indirizzi).
- **Problema dovuto alla pila, rilocazione statica e swapping incompatibili**
  La CALL e la RETURN funzionano correttamente? Sì, perché abbiamo ricalcolato gli indirizzi fisici. Potrebbero emergere problemi se facciamo lo swapping tra la CALL e la RETURN: quando andiamo ad eseguire la RETURN l'indirizzo contenuto in pila potrebbe non essere più corretto. La cosa non è banale, dunque lo swapping non può essere fatto in presenza di rilocazione statica.

7.2.3 Rilocazione dinamica

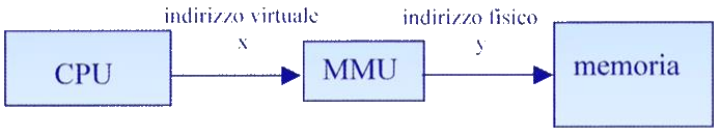
Il caricatore non è più rilocante. Supponiamo di voler allocare un processo in memoria fisica. Il caricatore:

- cerca una partizione abbastanza grande per contenere il processo,
- si fa una copia da disco a memoria principale senza ricalcolare gli indirizzi fisici.

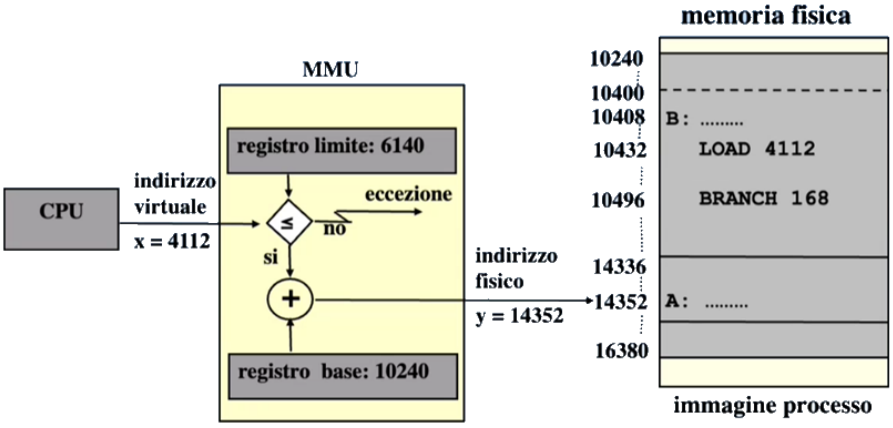
Si consideri il seguente esempio…



Contrariamente a prima abbiamo la rilocazione degli indirizzi virtuali in indirizzi fisici ***runtime***. L'operazione è svolta dal sistema operativo attraverso un modulo hardware detto ***Memory Management Unit*** (MMU), che si occupa di applicare la funzione di rilocazione.



Implementazione della MMU. L'implementazione della MMU è molto semplice, caratterizzata da due registri e un po' di logica.



I registri sono:

- Il registro base, che contiene il primo indirizzo della partizione dove è allocato lo spazio virtuale del processo.
- Il registro limite, che pone la dimensione della partizione riservata al processo.

Nella logica si applica la funzione di rilocazione, ottenendo così l'indirizzo fisico. Nel caso in cui l'indirizzo virtuale fornito esca dai limiti indicati dai due registri si ha un'eccezione e la si gestisce secondo una particolare politica. Si osservi che questi due registri sono parte del contesto del processo (aumenta leggermente l'overhead a causa dei trasferimenti da memoria a MMU e viceversa).

7.2.4 Organizzazione dello spazio virtuale

Lo spazio virtuale può essere **unico** o **segmentato**.

7.2.4.1 Memorie virtuali uniche

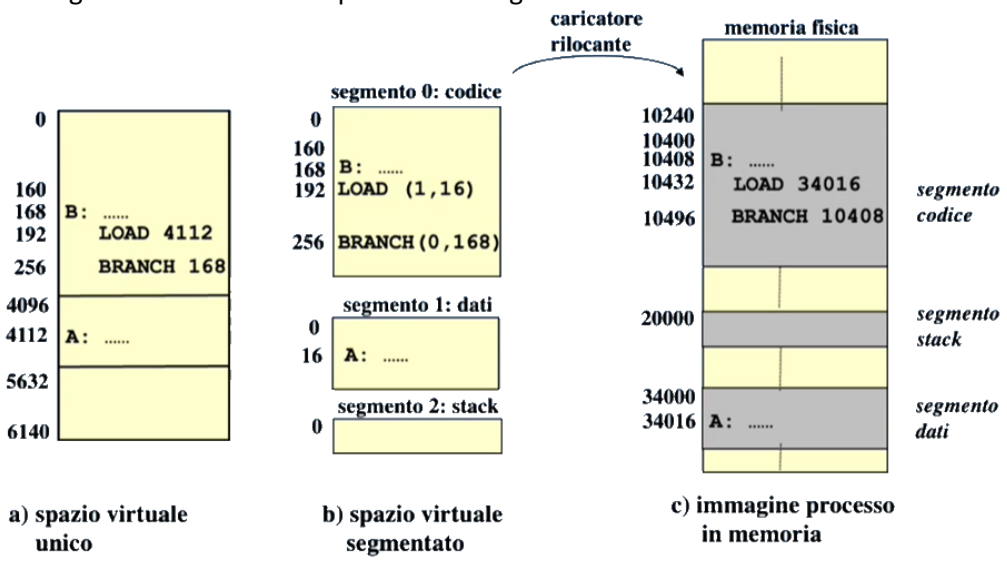
Fino ad ora abbiamo lavorato con memorie virtuali uniche: il modulo di caricamento permette di caricare uno spazio virtuale contiguo, unico, indivisibile, e avente una certa dimensione. Quali possono essere gli svantaggi?

- Essendo unica presenta una dimensiona massima: tutto ciò che serve al processo deve starci. Lo spazio è più grande, più difficile da gestire con più processi dentro (trovare partizioni grandi). RIVEDERE
- Non sono condivisibili spazi virtuali tra processi diversi. Ricordare il meccanismo di protezione.

Supponiamo che ai fini della condivisione del codice due o più processi vogliano accedere alla regione del codice. La cosa principale che potrei fare è replicare più volte in memoria il codice (una volta per processo). Dare accesso allo spazio virtuale rende complicato, se non impossibile, gestire l'accesso alla parte rimanente dello spazio di memoria. Per questi problemi si passa a un'organizzazione segmentata dello spazio virtuale.

7.2.4.2 Memorie virtuali segmentate

Una prima forma di segmentazione naturale prevede tre segmenti.



- segmento codice (che potrei chiamare segmento 0);
- segmento dati (che potrei chiamare segmento 1);
- segmento stack (che potrei chiamare segmento 2).

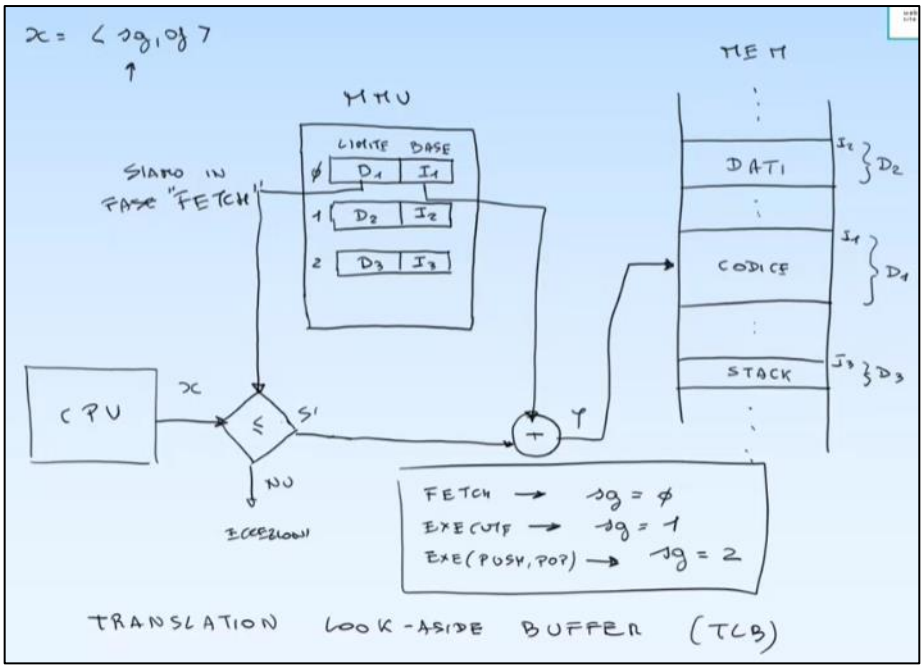
Attenzione alla variazione nelle due istruzioni: l'indirizzo virtuale è a due dimensioni, devo indicare segmento e offset. Cosa succede quando allochiamo in memoria fisica?

- Dobbiamo allocare tre partizioni, **il vantaggio immediato è che queste partizioni sono più piccole, dunque è più facile trovare spazio disponibile**.
- Un altro elemento interessante è la rottura della contiguità degli indirizzi nello spazio virtuale (ogni volta si ricomincia da 0).
- Il caricatore rilocante fa le stesse cose di prima rispetto al singolo segmento (funzione rilocante è sempre la somma, cambia la costante per ogni segmento). **Il fatto che agisca sul singolo segmento permette di allocare i segmenti in memoria fisica in modo non contiguo tra loro**.

Chiaramente sul segmento permangono i problemi della rilocazione statica (in primis l'impossibilità dello swapping).

7.2.4.2.1 Memory Management Unit (MMU) e memoria segmentata

In uno spazio diviso in tre segmenti abbiamo una MMU disposta come in foto: un limite e una base per ciascun segmento. I valori vanno caricati ogni volta che si cambia processo: l'overhead aumenta un pochino visto che lavoriamo su quattro registri in più (in realtà ciò che cambia ogni volta è soprattutto il registro base).



Per quanto riguarda la protezione non verifichiamo solo che l'offset sia nel limite, ma che l'indice del segmento sia valido (esistenza, ma anche permessi validi per manipolare il segmento).

Perché fermarsi a solo tre segmenti, andiamo oltre!

La segmentazione semplifica la vita, ma soprattutto permette di condividere segmenti con altri processi (il segmento isola semanticamente il contenuto dal resto dello spazio virtuale). Ciò è possibile ponendo per più processi lo stesso valore del registro base (lo stesso indirizzo fisico di partenza).

- **Implicazioni sulla MMU**:
  - il numero di registri non è più fisso, ma dipende dal processo.
  - Non posso più avere una MMU implementata interamente via hardware (dove il numero di registri è fisso). Vedremo più avanti la *tabella dei segmenti*, che contiene queste informazioni. Ogni processo avrà la sua tabella.
  - La MMU continua ad esistere, ma assume tutt'altra funzione e tutt'altro nome: *Translation Lookaside Buffer* (già vista a Calcolatori), che assume la funzione di una memoria cache e permette di abbattere i tempi di traduzione degli indirizzi virtuali in fisici.
- **Indirizzi monodimensionali con segmentazione a tre**
  Se io mantengo solo tre segmenti non ho bisogno di indirizzi virtuali strutturati in due campi: l'indice di segmento diventa implicito, la CPU è in grado di calcolarselo runtime. Come?
  - Indirizzi calcolati in fase di fetch appartengono al segmento codice
  - Indirizzi calcolati in fase di esecuzione appartengono al segmento dati
  - Nel caso di istruzioni PUSH e POP gli indirizzi appartengono al segmento stack.

7.2.4.3 Allocazione della memoria fisica e problema della frammentazione

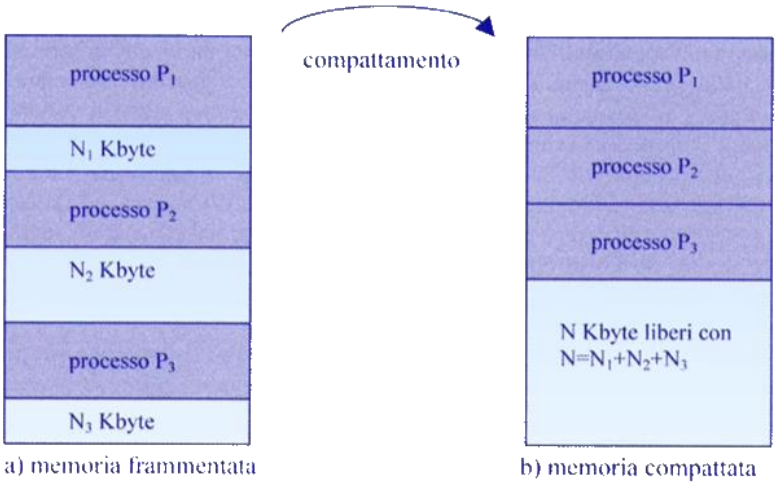
La figura introduce uno dei problemi importanti della gestione della memoria: la frammentazione.

7.2.4.3.1 Frammentazione interna

In alcune tecniche di gestione della memoria si divide la memoria fisica in partizioni avente dimensione fissa (non può essere modificata in base alle effettive esigenze dei processi). Questo significa che è alta la probabilità che il sistema riservi a un processo una partizione di dimensione superiore al fabbisogno, quindi avere un'area di memoria inutilizzata. La questione viene risolta introducendo partizioni di dimensione variabile.

7.2.4.3.2 Frammentazione esterna e introduzione della paginazione

Ogni volta che allochiamo uno spazio virtuale in memoria fisica dobbiamo trovare una partizione della memoria fisica abbastanza grande.



a) memoria frammentata b) memoria compattata

Tra le partizioni dei processi P1, P2, P3 abbiamo partizioni libere (ciascuna occupa un certo numero di Kbyte). A un certo punto viene richiesta una partizione che ha dimensione superiore a tutte le partizioni libere disponibili. Osserviamo che se io sommassi le partizioni otterrei una partizione di dimensione sufficiente per la richiesta. **In questo caso si parla di memoria frammentata esternamente**.

Come si risolve? La soluzione ovvia è un'operazione di compattamento degli spazi liberi, in modo da generare un'unica partizione libera la cui dimensione è la somma delle dimensioni delle precedenti partizioni non contigue. Tuttavia:

- non è possibile fare compattazione con rilocazione statica (per la questione degli indirizzi in pila);
- il processo è costoso perché devo fermare il sistema.

Non è la soluzione migliore, quindi come risolviamo? Rinuncio ad avere un'allocazione contigua in memoria fisica. Lo posso fare perché ho una distinzione tra indirizzi fisici e virtuali, legati da una funzione di rilocazione. Chiaramente cambia la funzione di rilocazione (non ho più una somma) e il modo con cui memorizziamo le corrispondenze. Vedremo che la cosa è abbastanza raffinata: non posso pensare a una tabella che contiene una corrispondenza per ogni singolo byte, altrimenti otterrei una tabella che paradossalmente è più grande dello spazio virtuale stesso. L'allocazione avviene per piccole partizioni: le pagine (già viste a Calcolatori)!!

Attenzione a non confondere paginazione e segmentazione: sono due cose distinte, che possono coesistere. La segmentazione è una divisione dello spazio virtuale senza vincoli su numero e dimensione (divisione in segmenti), la paginazione divide lo spazio virtuale in pagine con vincoli su numero di pagine e dimensione delle pagine stesse.

7.2.4.4 Caricamento unico o a domanda dello spazio virtuale

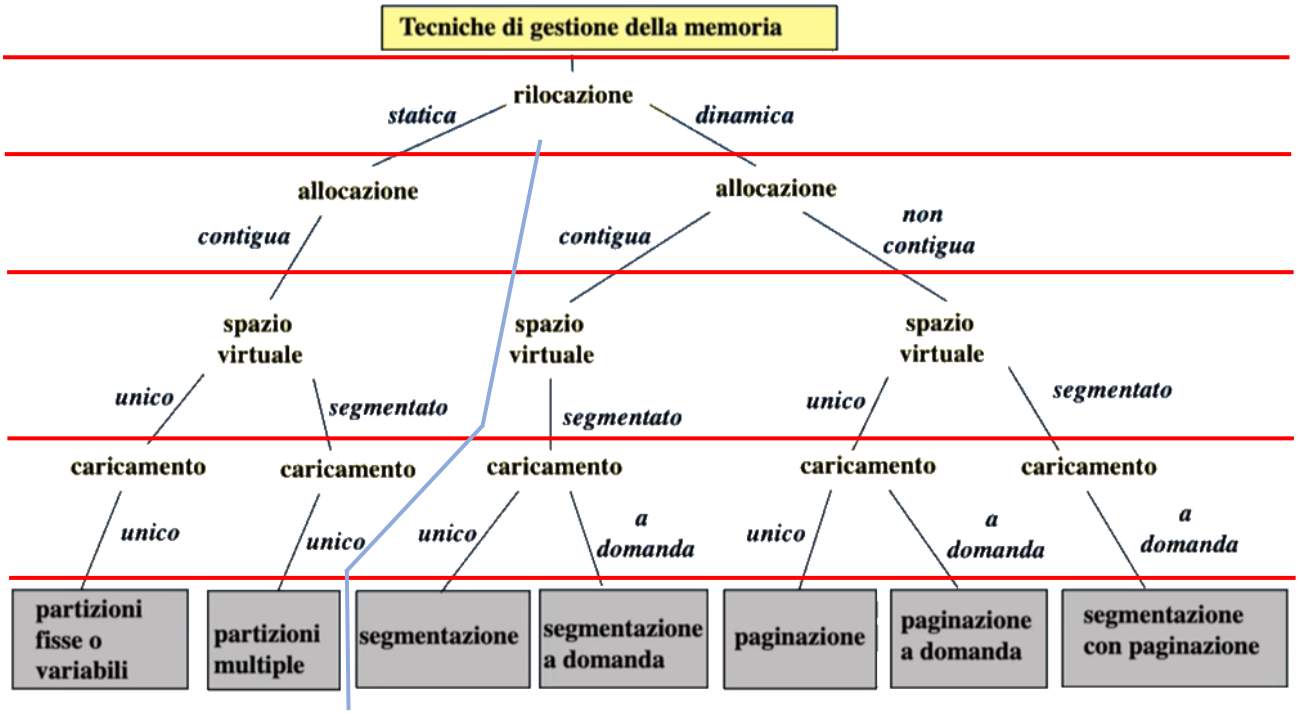
Lo spazio virtuale può essere caricato in due modi.

- **Caricamento unico**.
  Prendiamo lo spazio virtuale, unico o segmentato che sia, e prima di mettere il processo in coda pronti lo metto tutto in memoria fisica. Presenti dei limiti:
  - la memoria deve avere lo spazio necessario per tutto lo spazio virtuale;
  - non sono ammissibili processi che hanno spazio virtuale più grande della memoria fisica.
- **Caricamento su domanda**.
  Il processo va in coda pronti anche se non è avvenuta l'allocazione dello spazio virtuale in memoria. Nel momento in cui verrà generato il primo indirizzo virtuale il sistema provvederà a caricare il segmento codice che contiene quell'indirizzo, e così via… L'overhead aumenta, ma i vantaggi superano gli svantaggi.

7.3 Albero riassuntivo delle tecniche di gestione della memoria

L'albero riassume tutte le possibili tecniche di gestione della memoria. È organizzato in quattro livelli secondo i quattro aspetti caratterizzanti appena visti:

- rilocazione (statica o dinamica);
- allocazione (contigua o non contigua);
- spazio virtuale (unico o segmentato);
- caricamento (unico o a domanda).



- **Rilocazione statica**.
  - La rilocazione statica con allocazione non contigua non è possibile: gli indirizzi sono fisici e vengono calcolati tutti in anticipo.
  - In una rilocazione statica con allocazione contigua sono orientato a uno spazio segmentato per mitigare il problema della frammentazione esterna.
  - Le alternative sono:
    - partizione unica per ogni processo, di dimensione fissa o variabile (spazio virtuale unico e caricamento unico);
    - partizioni multiple per ogni processo grazie alla segmentazione.

    In entrambi i casi abbiamo caricamento unico, in quanto il caricamento su domanda si basa su swapping e questo è incompatibile con la rilocazione statica.
- **Rilocazione dinamica**.
  - **Osservazione banale**: quando si parla di tecniche di paginazione e/o segmentazione (nelle foglie dell'albero riepilogativo) la rilocazione è sempre dinamica.
  - In caso di allocazione contigua la segmentazione è obbligatoria, altrimenti avrei partizioni troppo grandi e notevoli difficoltà di allocazione. Rilocazione dinamica e caricamento a domanda sono compatibili, dunque la scelta è tra segmentazione (caricamento unico) e segmentazione a domanda (caricamento a domanda).
  - In caso di allocazione non contigua introduco la paginazione. In caso di spazio virtuale unico introduco paginazione (caricamento unico) o paginazione a domanda (caricamento a domanda). Se lo spazio è segmentato l'unica strada possibile è l'unione di segmentazione e paginazione: caricamento segmentato a domanda.

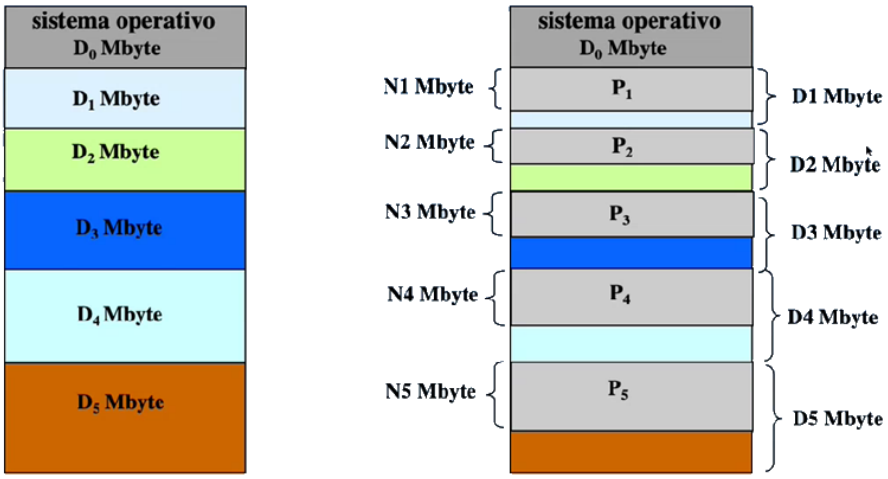
Per quello che abbiamo visto le soluzioni migliori sono le ultime tre citate (paginazione, paginazione a domanda e segmentazione con paginazione). Questo non ci impedisce di parlare delle altre soluzioni, visto che potrebbero essere applicate in sistemi operativi *non general purpose*.

7.4 Tecniche di gestione della memoria sotto rilocazione statica

7.4.1 Partizioni fisse

Rilocazione statica -> Allocazione contigua -> Spazio virtuale unico -> Caricamento unico

Abbiamo rilocazione statica (quindi caricatore rilocante, nel momento in cui allochiamo uno spazio virtuale in memoria fisica ricalcoliamo tutti gli indirizzi virtuali in indirizzi fisici attraverso una semplice funzione di rilocazione). Una partizione, ricordiamo, è un insieme di byte di memoria che può essere fissa o variabile in dimensione. L'allocazione è contigua, dunque non faremo uso (per ora) della paginazione: in altri termini avremo indirizzi virtuali contigui e indirizzi fisici sempre contigui dopo la rilocazione. Lo spazio virtuale è unico (anche se è possibile ricorrere alla segmentazione), il caricamento totale (non posso fare il caricamento su domanda).



a) partizioni libere

b) frammentazione interna = $(D_1-N_1)+(D_2-N_2)+(D_3-N_3)+(D_4-N_4)+(D_5-N_5)$

La memoria fisica deve essere divisa in un certo numero di partizioni, la cui dimensione è fissa e determinata a priori. Una di queste partizioni presenti è dedicata al codice del sistema operativo, le rimanenti ai processi (siamo in un sistema multiprogrammato, dobbiamo ospitare le immagini di più processi).

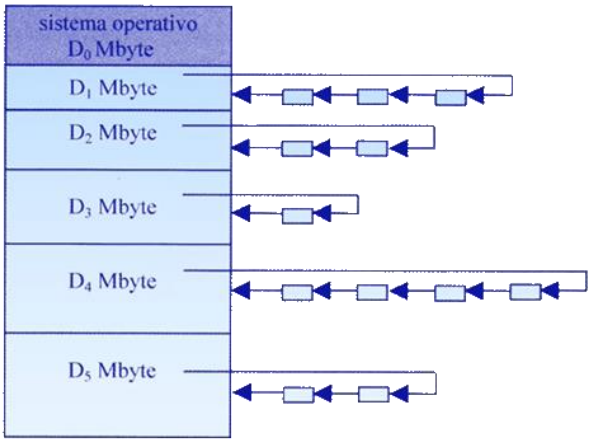
Viene creato il processo P1, il caricatore è in grado di calcolare la dimensione del caricatore (in questo caso N1 Mbyte). Il sistema cerca tra le partizioni libere una che sia abbastanza grande per contenere l'immagine del caricatore. Supponiamo che la partizione D1 sia quella giusta, dunque $D1 \geq N1$.

Abbiamo rilocazione statica: una volta che abbiamo verificato che l'indirizzo base della partizione ha un certo valore useremo questo per rilocare indirizzi (funzione di rilocazione).

Nella figura è raffigurato il problema base di questa tecnica: tutte le partizioni fisse occupate da un processo, si ha una frammentazione interna dovuta alla differenza tra la dimensione dell'immagine di un processo e la dimensione della partizione associata al processo (in alcuni casi questa differenza può essere sostanziosa).

Quindi:

- *overhead* basso (complessità dell'allocazione bassa, ci serve una struttura dati dove indichiamo per ogni partizione la dimensione e il suo stato attuale – libera o non libera).
- Frammentazione interna abbastanza rilevante, non possiamo farci molto se le partizioni sono fisse.

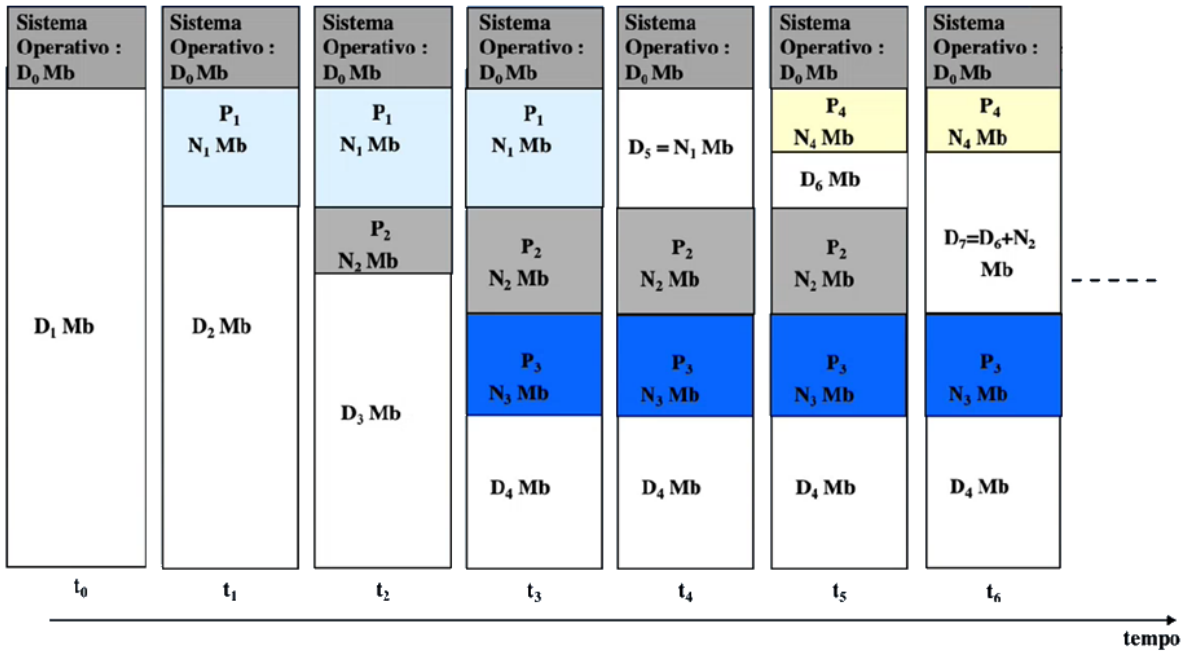


Si ha un vantaggio: quando non ho più partizioni libere posso ricorrere al **meccanismo di swapping**: trasferimento da memoria principale a memoria secondaria dell'immagine di un processo scelto come vittima (swap-out e poi swap-in per ripristinare la situazione iniziale). Normalmente in rilocazione statica lo swapping non funziona (abbiamo già evidenziato le motivazioni): questa è un'eccezione. Abbiamo per ogni partizione una coda di descrittori, dove l'elemento in testa è quello effettivamente caricato in memoria, mentre gli altri sono processi non caricati e destinati prossimamente a quella partizione. Applichiamo una politica Round-Robin (quindi logica FCFS nell'ordine dei descrittori): eseguo un processo, dopo un intervallo di tempo quanti revoco la CPU, pongo il processo a cui è stata revocata in fondo alla coda della relativa partizione e passo ad eseguire il nuovo processo in testa.

7.4.2 Partizioni variabili

Rilocazione statica -> Allocazione contigua -> Spazio virtuale unico -> Caricamento unico

L'unico modo per poter trattare il problema della frammentazione interna è abbandonare la dimensione fissa delle partizioni, cioè pensare a partizioni variabili.



- All'istante t_0 la memoria fisica è suddivisa in due partizioni: una dedicata al sistema operativa e la rimanente libera e disponibile per i processi.
- All'istante t_1 arriva il processo P_1 di dimensione N_1. Si ha $N_1 < D_1$, quindi P_1 può essere allocato (lo faccio nella parte alta della partizione). Fatta questa allocazione statica (con caricatore rilocante, secondo le regole viste prima) si calcola quali partizioni rimangono a seguito dell'allocazione: in questo caso rimane una partizione avente dimensione $D_2 < D_1$.
- Le cose si ripetono in maniera identica agli istanti t_2 e t_3.
- All'istante t_4 succede una cosa nuova: il primo processo che abbiamo allocato termina. Il processo viene deallocato, dunque si libera una nuova partizione subito dopo quella dedicata al sistema operativo, di dimensione D_5.
- Alloco un processo P_4 nello spazio di memoria appena liberato: posso farlo poiché $N_4 < D_5$. Il delta derivante dalla differenza tra D_5 ed N_4 da luogo a una nuova partizione di dimensione D_6.
- Nel tempo il numero di partizioni di memoria tende ad aumentare, ma andando avanti la dimensione delle partizioni è sempre minore. Abbiamo eliminato la frammentazione, ma il fenomeno appena descritto riduce le probabilità che un nuovo processo possa essere allocato (dimensione di ogni partizione insufficiente per ospitare l'immagine del processo). Si passa così dalla frammentazione interna alla frammentazione esterna.
- All'istante t_6 il processo P_2 termina e rilascia la sua partizione, che ha dimensione N_2 ed è adiacente in termine di indirizzi fisici alla partizione di dimensione D_6. È bene che il modulo del sistema operativo si accorga di questa adiacenza e fonda le partizioni.

Overhead. Verosimilmente partizioni variabili comportano un overhead maggiore, non tanto nella fase di allocazione (dove semplicemente verifichiamo una condizione, la presenza o meno di spazio) ma nella fase di rilascio (devo controllare l'adiacenza delle partizioni).

Limite. Non è possibile usare lo swapping, il trucco delle code visto prima con le partizioni fisse non può essere più usato.

Frammentazione interna ed esterna

- La frammentazione interna è stata eliminata
- La frammentazione esterna è inevitabile con un meccanismo basato su partizioni contigue. La segmentazione (spiegata più avanti) permette di attenuare il problema.

7.4.3 Riflessioni su come viene mantenuta la lista delle partizioni libere

Qualunque tecnica basata sulle partizioni richiede una struttura dati a disposizione del caricatore dove viene descritto lo stato di allocazione della memoria (dimensione delle partizioni e booleano che indica se le partizioni sono libere o meno). La struttura dati è in evoluzione (aggiunta di nuove partizioni ed eventuali fusioni), dunque dobbiamo gestire l'ordine delle partizioni:

- ***best-fit***, ordine crescente in base alla dimensione della partizione (best-fit perché si sceglie la partizione più piccola tra quelle di dimensione sufficiente).
    - Scorro un certo numero di elementi, sicuramente non le scorro tutte (quindi la complessità non è pari alla dimensione del vettore).
    - Quando alloco un nuovo spazio mi si crea una partizione più piccola, questa finisce all'inizio del vettore per la tecnica best-fit. Il costo è abbastanza contenuto, la ricerca si esaurisce nei primi elementi.
    - Nella fase di rilascio abbiamo una partizione nuova che devo sistemare nella lista. La posizione dipende dallo spazio che si è appena liberato. Anche questa operazione ha costo contenuto, il problema sta nella fusione: devo scorrere la lista di tutte le partizioni libere (per cercare gli indirizzi contigui alla partizione appena creata).
- ***first-fit***, ordine crescente in base agli indirizzi fisici (scelgo tra le partizioni disponibili quella con indirizzo più basso).
    - Ho una lista ordinata per indirizzi, se voglio individuare la partizione più piccola devo scorrere tutto il vettore. Ma non è un problema: la cosa più conveniente è fermarsi alla partizione più grande, in generale alla prima partizione libera.
    - Diventa semplice fare fusioni, perché l'ordinamento si basa sugli indirizzi.

7.5 Tecniche di gestione della memoria sotto la rilocazione dinamica

Ricordiamo: in presenza di rilocazione dinamica il programma genera indirizzi virtuali, che vanno tradotti eseguendo ogni volta una funzione di rilocazione. In presenza di rilocazione dinamica possiamo fare compattamento, fare swap liberamente e individuare nuovi indirizzi base.

7.5.1 Segmentazione

Rilocazione dinamica -> Allocazione contigua -> Spazio virtuale segmentato -> Caricamento totale

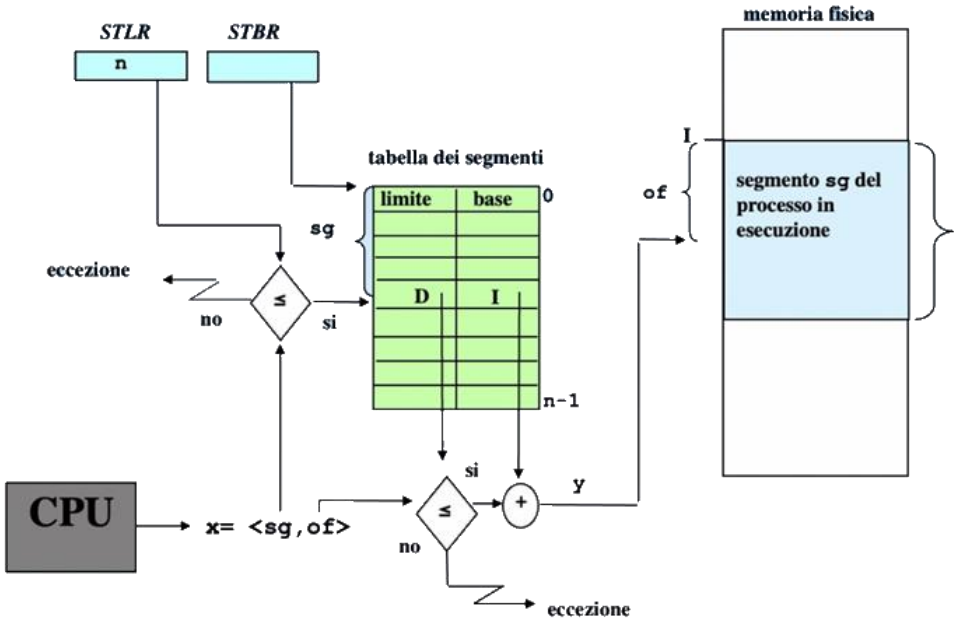
Quando parliamo di rilocazione dinamica e allocazione continua parliamo di segmentazione. Nel caso in cui i segmenti siano caricati subito, prima dell'esecuzione, si parla di caricamento unico. Nel caso in cui il caricamento venga rimandato si parla di segmentazione su domanda.

- Gli indirizzi virtuali sono a due dimensioni: $x = < sg, of >$
  (sg sta per segmento, of sta per offset/spiazzamento all'interno di quel segmento)
- Per avere una memoria segmentato è necessario un compilatore e un *linker*, che a partire dal codice sorgente sia in grado di generare indirizzi virtuali nella forma indicata. Ricordiamo la MMU, che memorizza per ogni segmento indirizzo base e indirizzo limite.
- Abbiamo già visto un'idea di divisione in tre segmenti: *code*, *data* e *stack*. Il nostro interesse è superare il limite di un numero fisso di segmenti, dunque superare l'implementazione completamente hardware della MMU.
    - La strategia è di avere un segmento specifico per ogni tipologia di dato, in modo tale da garantire indipendenza e gestire in modo più agile i permessi. Altra cosa è la possibilità di condividere codice tra più processi (basta porre gli stessi indirizzi fisici nella MMU, quindi basta una sola allocazione).
    - Ciò che andiamo a fare è creare una tabella dei segmenti al momento della creazione del processo (tempo di vita pari alla durata del processo, non piccolissima ma neanche grandissima). Ogni elemento è un descrittore che contiene l'indirizzo base e la dimensione limite (con cui costruiamo l'indirizzo limite). Il limite lo possiamo conoscere a priori grazie al contenuto del file eseguibile.
    - La tabella deve stare sempre in memoria principale: è inevitabile, ogni volta che leggiamo un indirizzo virtuale dobbiamo accedere a questa tabella. Appartiene allo spazio virtuale del sistema operativo E NON del processo: in memoria fisica abbiamo partizioni dedicate ai segmenti del sistema operativo.
    - Nel momento in cui eseguiamo il processo dobbiamo essere in grado di raggiungere questa tabella. A tal proposito è necessario introdurre due registri macchina:
        - *Segment Table Base Register* (STBR), contiene l'indirizzo fisico della tabella dei segmenti del processo in esecuzione

- *Segment Table Length Register* (STLR), contiene il numero di segmenti definiti per il processo.

Ovviamente questi dati finiranno nell'array *contesto* del descrittore di processo.

7.5.1.1 Traduzione degli indirizzi segmentati



- La CPU genera indirizzi strutturati (a due dimensioni, con sg e of).

- **Controllo del segmento**.
  Il primo controllo di protezione si fa sul segmento: poiché il numero di segmento dipende dal processo dobbiamo essere certi che l'indice di segmento generato runtime sia corretto. Facciamo questo controllo utilizzando il registro STLR e con un comparatore: se l'indice di segmento non è minore o uguale rispetto al valore di STLR si lancia un'eccezione.
- **Utilizzo della tabella dei segmenti**.
  Superato questo primo controllo possiamo utilizzare la tabella dei segmenti, che possiamo vedere come un'implementazione della tabella di rilocazione (i campi contenuti nel descrittore mi permettono di ottenere l'indirizzo fisico a partire dall'indirizzo virtuale). Col registro STBR recuperiamo l'indirizzo della base della tabella dei segmenti. Dalla base indicata da STBR ci spostiamo di sg posizioni.

- **Controllo dell'offset**.
  Dopo l'operazione possiamo accedere al descrittore del segmento. Il valore limite viene portato in un comparatore assieme all'offset: se l'offset è maggiore del limite si solleva un'eccezione (stiamo andando oltre la dimensione del segmento). Come gestisco questa eccezione? O abortisco il processo, o nel caso di un'allocazione dinamica possiamo lanciare una richiesta di aumento di dimensione del segmento.

- **Tutto è a posto**: prendo l'indirizzo base e l'offset, li uso per generare l'indirizzo fisico con cui accedere alla memoria fisica.

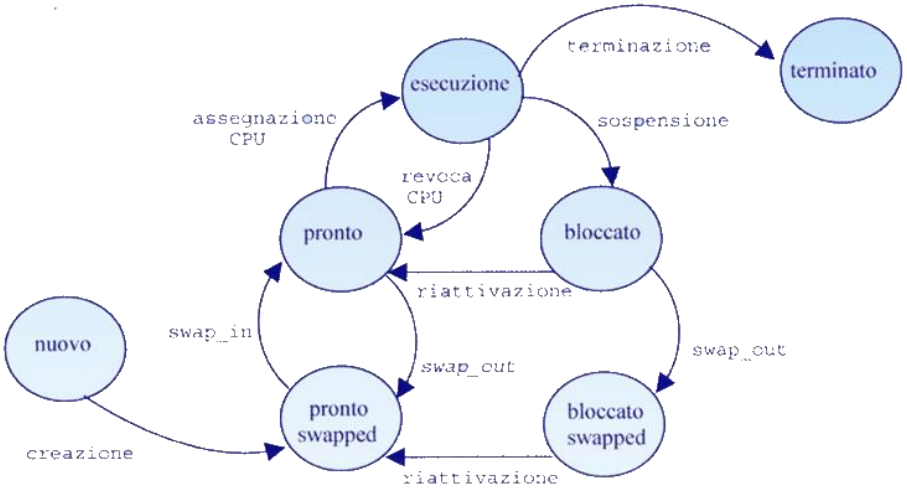
7.5.1.2 Stati swapped di un processo

In presenza del meccanismo di swap dobbiamo aggiornare il grafo degli stati possibili di un processo.

- Abbiamo i tre soliti stati: *pronto*, *esecuzione*, *bloccato*.
- Pronto e bloccato vengono sdoppiati creando *pronto swapped* e *bloccato swapped*.
- Al momento della creazione del processo si creano tutte le strutture dati. In questa versione del grafo non si va subito in *pronto*, ma in *pronto swapped*: questo perché non è scontato che il contenuto del processo venga allocato in memoria principale. Si passa da *pronto swapped* a *pronto* con l'operazione di `swap_in`.
- Per decisione del sistema operativo un processo può passare da pronto a pronto swapped attraverso un'operazione di `swap_out`.
- Si prevede la revoca: il processo passa da *esecuzione* a *pronto*.
- Per liberare spazio è possibile spostare i processi bloccati in memoria secondaria: con lo `swap_out` un processo può passare da *bloccato* a *bloccato swapped*. Questo `swap_out` è quello più ricorrente: la scelta

privilegiata per levare spazio è applicare l'operazione sui processi bloccati, cioè su processi che per un po' non faranno nulla (in contrasto coi processi pronti, che potrebbero essere riattivati con l'algoritmo di scheduling.

- Per riattivare lo stato si aspetta che questo venga riattivato, passando dallo stato *bloccato swapped* allo stato *pronto swapped*.



7.5.1.3 Contenuto del descrittore di segmento (con bit per la segmentazione su domanda)

Analizziamo infine il contenuto del descrittore di segmento.

base	limite	controllo				
Indirizzo della partizione o della *swap area*	Dimensione della partizione	R	W	U	M	P

- Abbiamo l'indirizzo base e la dimensione della partizione (calcoliamo l'indirizzo limite a partire da questo dato).
- Oltre ai dati già noti abbiamo dei bit di controllo: tra questi indichiamo i bit R e W, con cui indichiamo se è possibile leggere e/o scrivere all'interno del segmento (per esempio potrei porre il bit W uguale a 0 per quanto riguarda il segmento contenente codice, abbiamo già detto che è lesivo modificare runtime il codice del programma).
- Attenzione: l'accesso ha un costo, non solo controlli ma accesso a memoria fisica. Il fatto è che per ogni accesso a memoria fisica devo fare accessi in più per ottenere l'indirizzo fisico. Alleggeriamo la cosa delegando alla MMU il ruolo di cache: diventa TLB.

- **Segmentazione a domanda**

  Consideriamo la segmentazione a domanda, cioè la tecnica di gestione con rilocazione dinamica, allocazione contigua, spazio segmentato e caricamento a domanda. Nella segmentazione possiamo considerare una delle seguenti varianti:
  - nessun processo inizialmente è caricato in memoria principale;
  - caricamento di qualche segmento al momento della creazione dei processi (più efficiente).

  Abbiamo aggiunto tre bit:
  - Bit P mi segnala se il segmento è caricato in memoria o no (quindi se la riga della tabella rappresenta effettivamente traduzioni da indirizzi virtuali a indirizzi fisici).
    - Quando $P = 1$ il significato del campo base è quello già detto.
    - Quando $P = 0$ il campo base è usato per memorizzare l'indirizzo della swap area del segmento.
  - Bit U mi segnala se c'è stato un accesso al segmento (lettura e/o scrittura, si genera un indirizzo virtuale appartenente a un certo segmento). Quando un segmento viene caricato in memoria principale sostituiamo il contenuto del campo base e poniamo $U = 0$
  - Bit M (*modify*) ci serve per ottimizzare il meccanismo di swapping, ci segnala se c'è stato almeno un accesso in scrittura. Inizialmente $M = 0$

  Lo schema che abbiamo già visto non cambia, ma si ha un ulteriore test di controllo: il controllo del bit di presenza P. Se $P = 0$ si solleva un'eccezione di tipo *segment fault*. Per avere un'idea sul significato dei bit si veda l'algoritmo di rimpiazzamento second-chance.

Eliminiamo alla radice il problema della frammentazione esterna introducendo la paginazione dalla prossima pagina.

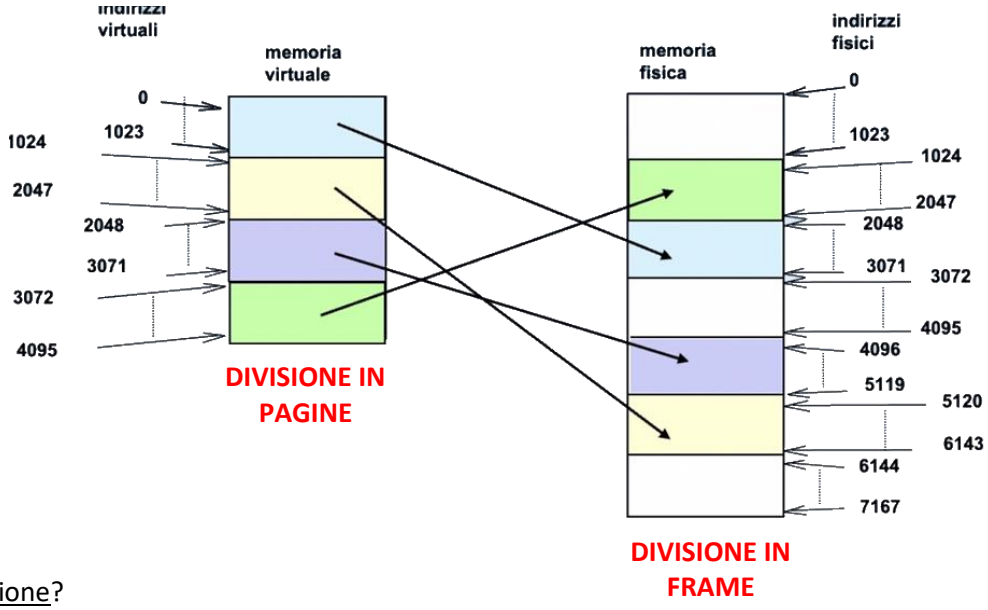
7.5.2 Paginazione

Rilocazione dinamica -> Allocazione non contigua

La soluzione consiste nell'eliminare la contiguità nell'allocazione dello spazio fisico (ovviamente la contiguità viene mantenuta nello spazio di indirizzamento virtuale).

- Si suddivide lo spazio virtuale in blocchi di indirizzi virtuali detti *pagine* (o pagine virtuali).
- Si suddivide lo spazio fisico in blocchi di indirizzi fisici detti *frame*.

Non ci poniamo più il problema di individuare una partizione libera di dimensione sufficiente, ma cercheremo il numero di pagine necessarie per il nostro processo, e queste non devono essere necessariamente contigue.



E la frammentazione?

- La frammentazione esterna è eliminata perché abbiamo modificato il problema.
- La frammentazione interna rispetto ai frame scelti rimane, ma di fatto non è un problema (vista la dimensione di *pagina e frame*). La soluzione dovrebbe essere ridurre la dimensione delle pagine fino a farle coincidere col singolo byte, ma questa cosa implica un aumento della dimensione della tabella delle pagine: diventa grande quanto lo spazio virtuale, moltiplicato per la dimensione del descrittore di pagina (un aumento a due dimensioni, sia nel numero di righe che nel numero di colonne).

Condivisione di una pagina.

Condividere una pagina è più scomodo rispetto al condividere un segmento: molto più semplice dire che più processi condividono un segmento, poiché questo rappresenta una unità semantica del programma.

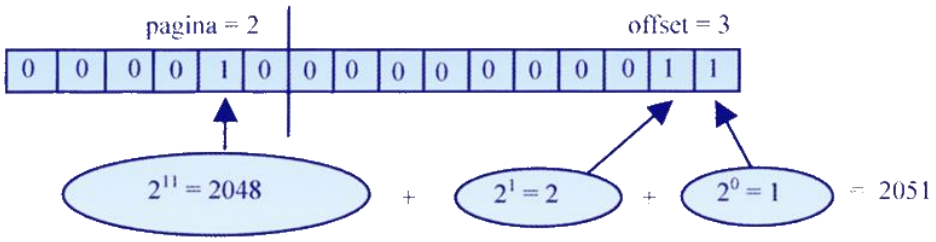
Perdita di significato degli stati swapped.

In presenza di segmentazione a domanda lo stato *swapped* perde di significato: teoricamente il processo è sempre *swapped*, tanto è che nei casi estremi il descrittore entra in coda pronti pur non avendo nulla allocato. Lo stesso discorso vale per la memoria paginata: lo stato esiste in linea di principio, è reato con paginazione tutta insieme, vedremo che continua ad esistere anche con paginazione a domanda.

7.5.2.1 Traduzione di indirizzi virtuali in indirizzi fisici, numero di pagina e tabella delle pagine

Siamo in rilocazione dinamica, anche qua dobbiamo chiederci come avverrà la traduzione. Per ogni processo avremo una tabella delle pagine dove stabiliamo le corrispondenze tra pagine e frame: implementiamo così la funzione di rilocazione.

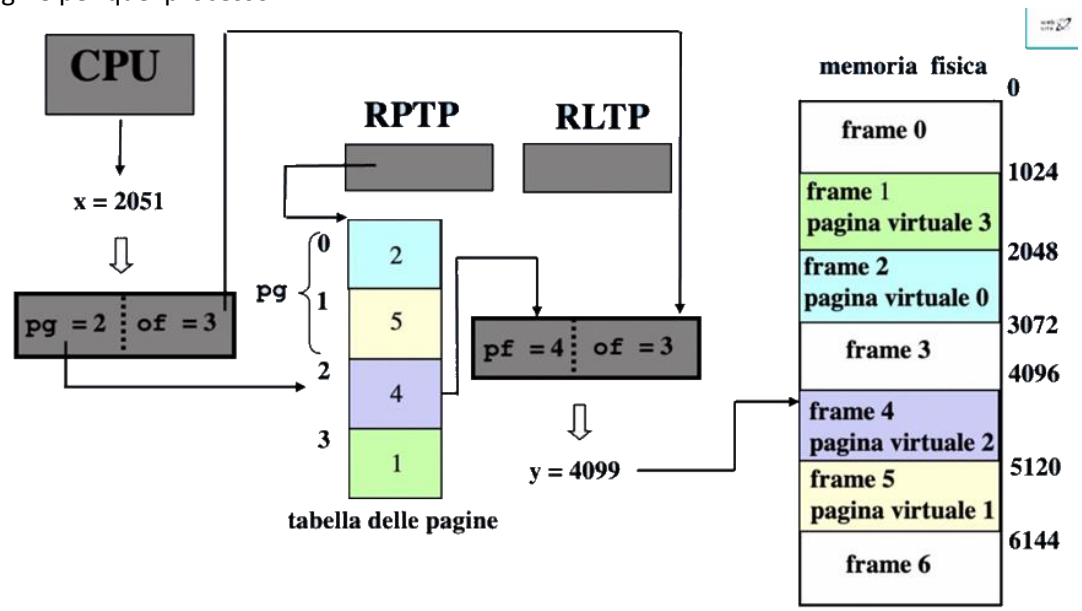
Come nella segmentazione ci immaginiamo un indirizzo virtuale diviso in due parti: numero di pagina e offset.



Supponiamo che gli indirizzi siano su 16 bit e che ogni pagina sia di un 1Kb. Sappiamo che per indirizzare 1Kb sono necessari 10 bit. I primi dieci bit meno significativi sono dedicati all'offset della pagina, i rimanenti identificano la pagina in cui ci troviamo.

Se aumentiamo la dimensione della memoria senza alterare il numero di bit per pagina si ha un aumento del numero di bit dedicati all'identificativo della pagina. La strutturazione dell'indirizzo non richiede cambiamenti di programma (in contrasto con la segmentazione).

Tabella delle pagine. Al momento della creazione del processo allochiamo il descrittore del processo e creiamo la tabella delle pagine per quel processo.



- **Quanto è grande quella tabella?**
  Se abbiamo k bit per rappresentare l'identificativo di una pagina di 1Kb allora avremo una tabella delle pagine con 2^k descrittori di pagina. Attenzione, ogni descrittore ha una certa dimensione (sicuramente superiore al singolo byte).
- **Dove si trova la tabella delle pagine?**
  Ovviamente in memoria principale, dobbiamo prevedere spazio nello spazio virtuale del sistema operativo.
- **Cosa scriviamo nella tabella?**
  Dobbiamo codificare la funzione di rilocazione, per questo riportiamo all'interno del descrittore l'indice di frame. Attenzione: il descrittore riporta l'indice del frame e non l'indirizzo del frame, questo è un bene perché ci permette di risparmiare bit. Quanto risparmiamo? I bit normalmente usati per rappresentare l'offset.
- **Come raggiungiamo la tabella e come otteniamo l'indirizzo fisico?**
  Utilizziamo il registro RPTP (Registro Puntatore Tabella Pagina), che contiene l'indice del primo frame dove è contenuta la tabella delle pagine. Nella lettura utilizziamo questo numero di frame, ci spostiamo per raggiungere il descrittore relativo al numero di pagina indicato in input, leggiamo il numero di frame e lo concateniamo con l'offset dell'indirizzo virtuale. Per i controlli di protezioni si consideri anche il registro RLTP (Registro Lunghezza Tabella Pagine): se $pg > RLTP$ allora lanciamo un'eccezione, il processo sta tentando di accedere ad una pagina che non fa parte del suo spazio (basta questa condizione perché lo spazio virtuale è sempre contiguo).
- **Come rappresentiamo lo stato?**
  Per rappresentare lo stato di allocazione dobbiamo usare i due registri. I valori di questi vengono determinati al momento della creazione del processo e posti nell'apposito array del descrittore di processo. Si calcola il numero di pagine per lo spazio virtuale del processo, si crea e si inizializza la tabella delle pagine, si carica la tabella in memoria fisica. La tabella delle pagine si riempie con gli indici di frame, questi li abbiamo solo dopo aver allocato tutto lo spazio virtuale del processo.
- **Ulteriori accessi**
  Come con i segmenti è necessario fare accessi ulteriori per leggere la tabella delle pagine. La cosa aumenta i tempi, tempi che andiamo ad abbattere con una cache. La MMU in architetture con paginazione funge da TLB.
- **Quale struttura dati usa il gestore della memoria in presenza di paginazione?**
  Abbiamo bisogno di una struttura dati che concatena in sequenza i frame liberi attualmente presenti in memoria. Tipicamente la frame table è organizzata come una coda circolare.

7.5.2.2 Descrittore di pagina

Il descrittore di pagina assume la seguente forma (tenendo conto anche della paginazione su domanda)

Campo pagina fisica	controllo				
Indice della pagina fisica se $P = 1$ Indirizzo su disco se $P = 0$	R	W	U	M	P

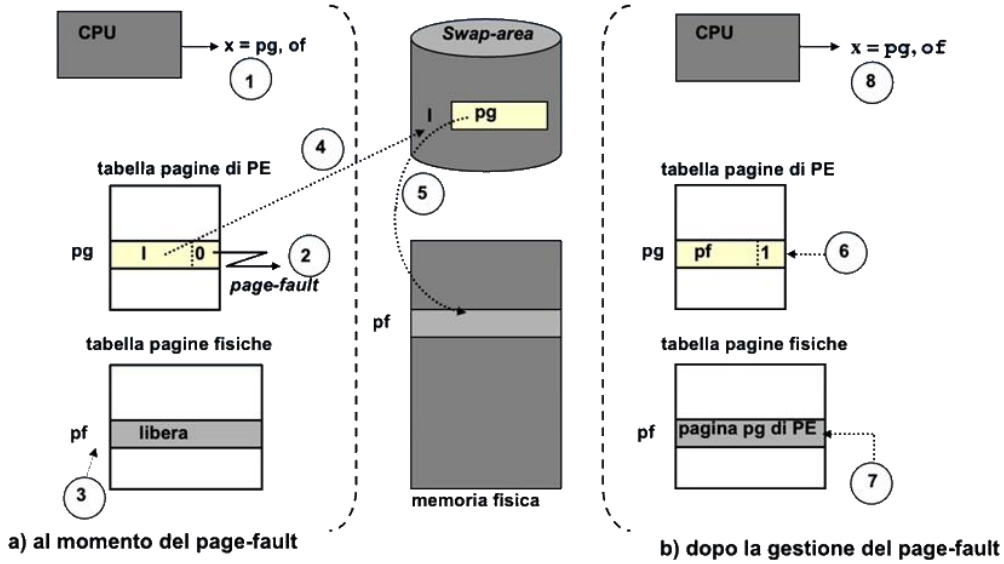
- Come già anticipato abbiamo il solo indice della pagina fisica dove è allocata la pagina logica. Nella paginazione a domanda il sistema non necessariamente alloca tutto lo spazio al momento della creazione del processo, si allocano le pagine ogni volta che viene generato un riferimento a quella pagina.
- Il bit P di presenza indica se la pagina è caricata o no in memoria fisica. In base al valore di P indichiamo l'indice della pagina fisica o l'indirizzo (su disco) alla *swap area*.
- Abbiamo i bit R e W per indicare i diritti di accesso alle pagine (se si può leggere e/o scrivere) e gestire i controlli.
- Il bit di uso U e il bit di modifica M vengono usati per gestire la paginazione a domanda (algoritmi di rimpiazzamento). Indico, rispettivamente, se ho fatto un accesso e se ho fatto un accesso in scrittura. Anche qua, per avere un'idea più chiara, si consiglia di vedere l'algoritmo di rimpiazzamento second-chance.

7.5.3 Gestione del page fault (paginazione su domanda)

L'eccezione di *page fault* viene lanciata nel momento in cui, visitando un descrittore nella tabella delle pagine, si trova il bit P uguale a 0. Avere il bit con questo valore significa che il contenuto della pagina non è stato posto in memoria fisica (e per questo si indica nel campo della pagina fisico l'indirizzo alla *swap-area* del disco). L'eccezione è uno dei pochi casi (se non l'unico) in cui si cerca di rimediare all'errore attraverso un'apposita gestione.

7.5.3.1 Eventi nel caso di frame liberi

Il seguente schema ci aiuta a capire la sequenza cronologica degli eventi importanti previsti per la gestione di un page fault. Siamo nel caso in cui si hanno ancora pagina libere quando si manifesta il page fault.



La CPU genera indirizzi virtuali strutturati in numero di pagina ed offset, abbiamo un gestore della memoria che fa affidamento su una tabella delle pagine fisiche. Per ogni pagina fisica vengono riportate delle informazioni utili, necessarie per gestire il page fault.

- Quando un frame non è libero non mi basta un bit: devo indicare quale pagina e quale processo occupa quel frame (serve un indice di pagina grande quanto pg e un indice di processo grande quanto il pid).
- In presenza di un *page fault* il sistema deve caricare quella pagina dal disco alla memoria principale. Per caricare quella pagina dobbiamo trovare frame liberi (quindi consultare la tabella dei frame) e poi fare trasferimento I/O.

Vediamo gli eventi in ordine cronologico. Supponiamo di avere un processo PE:

- **Esecuzione dell'istruzione e individuazione del page fault**
  1. La CPU esegue un programma e genera un indirizzo virtuale pg, of.
  2. Nel descrittore di pagina (posto nella tabella delle pagine del processo PE) troviamo $P = 0$, si genera il *page fault*.
     a. **Che fine fa il processo che lo ha generato?**
        Il cambio di contesto è operazione costosa, che in questo caso non è necessario fare per eseguire

il gestore. Ci limitiamo a fare un cambio di privilegio (in sostanza un cambio di soggetto, ripensando a UNIX) con salto alla prima istruzione del gestore.

b. **Cosa fa il gestore?**
Cerca un frame libero nella tabella dei frame liberi. Supponiamo ci sia almeno un frame libero…

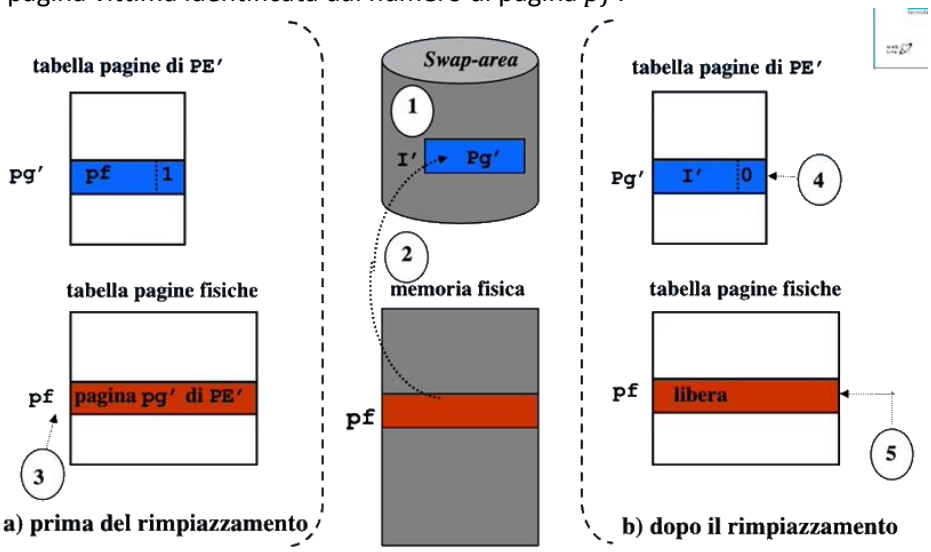
- **Individuazione del frame libero**
  3. Esiste un frame libero, dunque possiamo leggere pf (l'indice di memoria fisica) nel relativo descrittore della tabella delle pagine fisiche.
- **Trasferimento DMA da swap-area a frame**
  4. Il passaggio avviene attraverso trasferimento in DMA da disco (da swap area) a memoria principale. Tra descrittore di pagina e descrittore di pagina fisica (frame) abbiamo tutte le informazioni necessarie per fare questo trasferimento:
     a. l'indirizzo sorgente di trasferimento (l'indirizzo della swap-area, letto nel relativo descrittore della tabella delle pagine del processo PE, dove $P = 0$);
     b. l'indirizzo destinatario (il numero di frame pf, letto nella tabella delle pagine fisiche[10]);
     c. la dimensione in byte del trasferimento (la dimensione della pagina, determinata a priori e fissa).

  A questo punto il processo non ha più bisogno della CPU. Il processo che ha generato il page fault viene sospeso: durante il trasferimento viene lanciato lo scheduler ponendo così in esecuzione un altro processo.
  5. Si conclude il trasferimento dalla swap-area (disco) alla memoria fisica.
- **Aggiornamento delle informazioni delle tabelle e nuova esecuzione dell'istruzione**
  6. Aggiornamento della tabella delle pagine del processo PE (numero di frame pf e settiamo P).
  7. Aggiornamento della tabella delle pagine fisiche (dove indichiamo l'associazione tra numero di frame pf e numero di pagina pg).
  8. Riattivazione del processo PE e nuova esecuzione dell'istruzione che aveva generato il page fault.

È possibile che si generi nuovamente un page fault? Significa che ho sbagliato le mie previsioni in termini di rimpiazzamento, rimpiazzando una pagina che mi sarebbe stata utile poco dopo.

7.5.3.2 Eventi nel caso di frame non liberi

Vediamo cosa succede quando, al momento del page fault, la tabella delle pagine fisiche non mi da nulla (memoria piena). L'idea è avere qualche schema di rimpiazzamento delle pagine, cioè scegliere una pagina vittima da spostare nella swap-area. Riprendendo la scaletta di prima vediamo che i primi due step sono uguali, e al terzo step non troviamo frame disponibili. Consideriamo un processo PE' ed eseguiamo l'algoritmo di rimpiazzamento: l'output dell'algoritmo è una pagina vittima identificata dal numero di pagina pf.



- **Individuazione dell'indirizzo di swap-area (constatazione)**
  1. La prima cosa è individuare un indirizzo di swap-area in cui fare swap-out della pagina da liberare. Supponiamo che l'indirizzo di swap sia I'.
- **Trasferimento in DMA da frame a swap-area**
  2. Inizializzo il trasferimento in DMA. Come prima recuperiamo le informazioni necessarie al trasferimento:
     a. l'indirizzo sorgente è il pf (la pagina vittima restituita dall'algoritmo di rimpiazzamento);

[10] Quando scegliamo la dimensione della pagina (quindi del frame) può essere sensato porla uguale all'organizzazione a blocchi tipica del disco. Il disco può essere visto come una periferica I/O organizzata a blocchi (ci ritorneremo successivamente).

b. l'indirizzo destinatario è l'indirizzo di swap I';
c. La dimensione del trasferimento è quella del frame.

Il processo PE' si sospende durante il trasferimento.

- **Aggiornamento delle informazioni delle tabelle**
  3. Aggiorno la tabella delle pagine del processo PE', metto $P = 0$ (il bit di presenza) e pongo I' per ricordarmi dove è stata posta la pagina.
  4. Aggiorno la tabella delle pagine fisiche: all'indice pf scrivo libero (non c'è più il legame tra numero di pagina e numero di frame).

Dopo questi step si torna allo step tre della gestione del page fault: adesso abbiamo pagine libere.

7.5.3.3 Algoritmi di rimpiazzamento per lo swap-out

L'algoritmo di rimpiazzamento indica quali sono i processi vittima su cui fare *swap-out*, liberando pagine fisiche in memoria principale: garantisce la massima flessibilità, permette uno spazio virtuale che va oltre la dimensione della memoria fisica.

- Il costo emerge quando facciamo cattivi rimpiazzamenti (per esempio la situazione già detta prima).
- La località dei riferimenti permette di definire il *Working-Set* di ogni processo, l'insieme delle pagine su cui il processo sta lavorando e che in virtù di ciò devono essere caricate in memoria. Se il sistema ha caricamento unico il working-set coincide con lo spazio virtuale, se il caricamento è a domanda la cosa è più articolata. In realtà il processo utilizzerà solo un sottoinsieme delle pagine a sua disposizione, dunque il Working-set rappresenta l'insieme delle pagine utilizzate dal processo. Affinchè le pagine siano utilizzate è necessario in primis che vengano caricate.
- Se all'inizio non abbiamo pagine caricate allora partiremo subito con un page fault: ne avremo un bel po', il *Working-Set* inizia a formarsi col caricamento delle varie pagine. A un certo punto la formazione del Working-Set rallenta, e rimane stabile per un po' di tempo.

Algoritmo ottimo (che non esiste). L'algoritmo di rimpiazzamento ottimo indica solo le pagine che non verranno più riferite da un sistema. Comporta un numero di page fault pari alla dimensione del Working-Set (se io ho dieci pagine genero dieci page fault, e dopo basta!).

Algoritmo ideale (quello che cerchiamo di fare). L'algoritmo di rimpiazzamento ideale indica le pagine che saranno riferite più tardi nel tempo. In caso di previsioni errate il sistema va in *trashing*: dedica la maggior parte dei cicli di CPU alla gestione del *page fault*, questo perché fa le scelte sbagliate nei processi vittima.

Come si valutano le prestazioni di un algoritmo di rimpiazzamento? Analizzo il modo in cui evolve il Working-Set di un determinato processo contando il numero di page fault. Per fare questa cosa utilizziamo la Page Reference List, che indica in sequenza quale pagine virtuali il processo ha generato. La cosa può dipendere anche da una cattiva programmazione: una cosa strutturata riduce senza ombra di dubbio il numero di page fault.

Due strategie possibili nell'applicazione dell'algoritmo di rimpiazzamento

- Scelgo la vittima tra le pagine del processo che ha generato il page fault (*rimpiazzamento locale*)
- Scelgo la vittima tra le pagine di tutti i processi (*rimpiazzamento globale*).

7.5.3.3.1 Il più semplice degli algoritmi di rimpiazzamento: algoritmo FIFO

L'algoritmo di rimpiazzamento più semplice è l'algoritmo FIFO, che lavora sulla tabella delle pagine fisiche. Supponiamo di organizzare questa tabella come una coda circolare, con due puntatori agganciamo la testa e la coda, questi scorrono a seconda dell'uso. Con le code circolari implemento la politica FIFO. La coda, quando si riempie, avrà testa e coda coincidenti: a quel punto entra in scena l'algoritmo di rimpiazzamento, levo la testa e quindi il processo più vecchio.

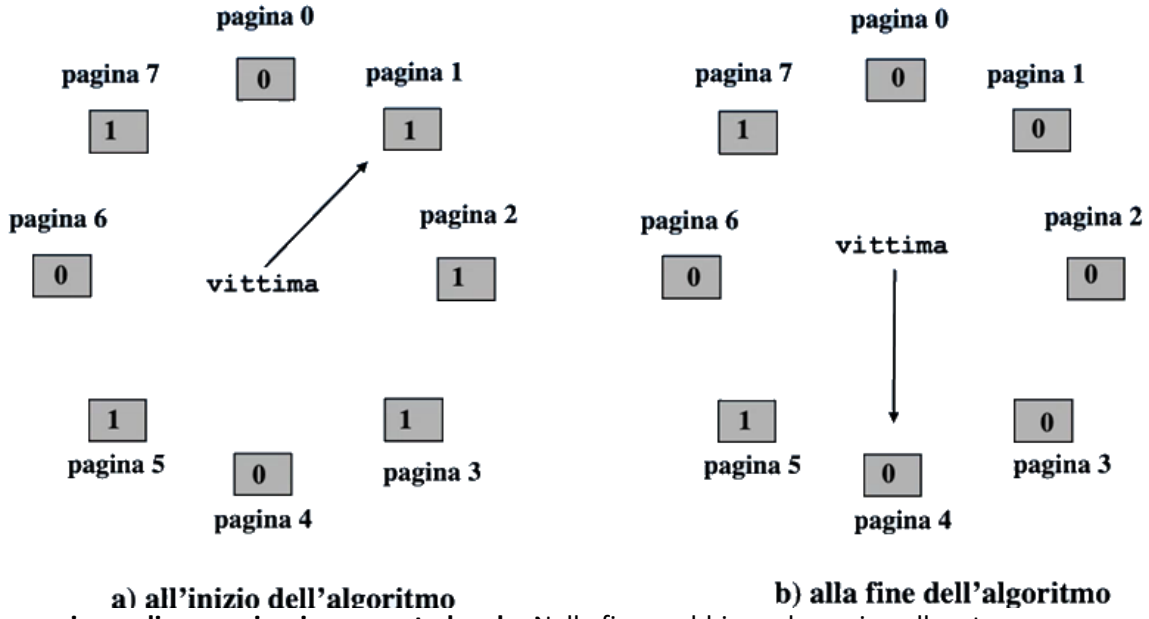
- La complessità dell'algoritmo è praticamente nulla, overhead veramente minimo.
- Il problema è che la vittima non è detto sia quella giusta per il principio di località dei riferimenti (nei processi possono permanere pagine costantemente).

Quale strategia è più conveniente? Quella di rimpiazzamento globale, applicare l'altra aumenta i rischi di sbagliare (agiamo su un numero di pagine minori). Il rimpiazzamento locale comporterebbe anche la creazione di una coda per ogni processo, mentre col rimpiazzamento globale abbiamo un'unica coda.

7.5.3.3.2 Algoritmo *Least Recently Used* (LRU)

Altra strategia potrebbe essere guardare da quanto tempo una pagina viene riferita, e scegliere come vittima quella non riferita da più tempo. Facciamo questo ponendo in ogni descrittore di pagina un *timestamp*: attenzione ai bit necessari, ma soprattutto alla granularità. 32 bit dovrebbero essere sufficienti se rimaniamo ai secondi, ma l'aumento di spazio non è banale. In più ogni volta dovrei generare il timestamp e copiarlo nel descrittore. Morale della favola: overhead temporale e overhead spaziale.

7.5.3.3.3 Algoritmo second-cache (o *clock algorithm*)



- **Supponiamo di avere rimpiazzamento locale**. Nella figura abbiamo le pagine allocate a un processo. Per ogni processo abbiamo un bit di uso. Organizziamo una coda circolare e creiamo un puntatore *vittima*.
- A un certo istante il puntatore vittima punta alla pagina 1. Il processo fa page-fault, facciamo rimpiazzamento prendendo la pagina puntata dal puntatore vittima.
- La vittima deve essere confermata: ciò avviene solo se il bit di uso è uguale a 0, non prendo quelle bit di uso uguali ad 1 perché sono state usate e quindi, nella logica di chi ha realizzato l'algoritmo, è più probabile che vengano richiamate in causa. Le pagine saltate vedono il loro bit di uso posto uguale a 0, e rimarrà tale salvo nuovi riferimenti alle stesse.
- Nell'esempio qua sopra dopo pagina 1 si sceglie pagina 4.

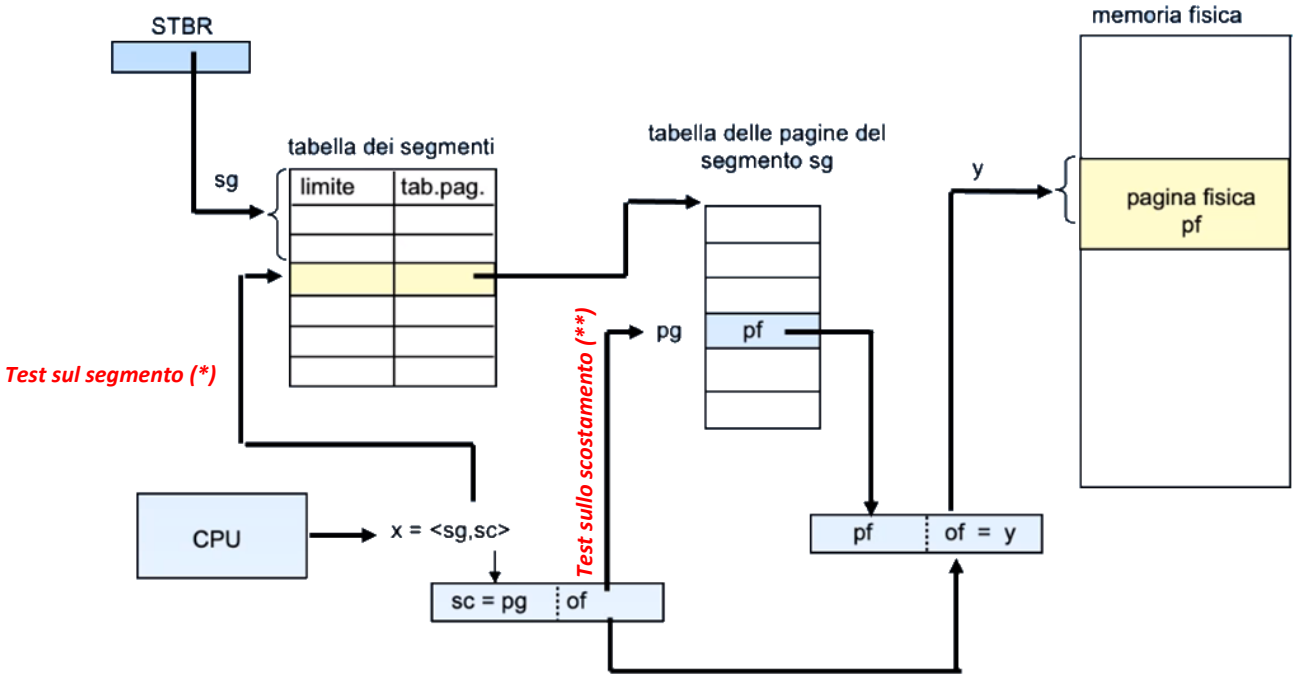
E se io avessi tutti i bit d'uso uguali a 1? Otterrei un comportamento simile all'algoritmo FIFO: si prende la pagina più vecchia.

Vantaggi. Non dobbiamo scrivere/gestire *timestamp*, dobbiamo scorrere una struttura dati che è piccola con rimpiazzamento locale e più grande in presenza di rimpiazzamento globale.

Ottimizzazione. Possiamo fare una modifica all'algoritmo considerando anche il bit di modifica. Quando la vittima ha il bit d'uso uguale a 0 andiamo a guardare anche il bit di modifica: favorisco nella scelta della modifica chi ha questo bit uguale a 0. Il fatto è che se questo bit è uguale a zero allora non ci sono state modifiche della pagina durante la sua permanenza in memoria centrale, ergo il contenuto non è cambiato rispetto alla copia in swap-area: se decido di rimpiazzare questa pagina non ho bisogno di fare una copia del processo in swap-area con l'operazione di swap-out, visto che il contenuto non è cambiato.

7.5.4 Segmentazione paginata

Rilocazione dinamica -> Allocazione non contigua -> Spazio virtuale segmentato -> Caricamento a domanda



La segmentazione paginata unisce le due tecniche precedentemente descritte: segmentazione e paginazione. Lo schema consiste nella composizione a cascata dei due schemi già visti.

- La CPU genera un indirizzo virtuale strutturato in due campi $< sg, sc >$ (dove sg è l'identificativo del segmento ed sc è l'offset/scostamento)
- Ogni processo ha una sua tabella dei segmenti. **Il segmento è paginato**, dunque non è più opportuno parlare di partizione: abbiamo un segmento costituito da più pagine
- Ogni processo avrà un numero di tabelle di pagine pari al numero di segmenti in cui viene diviso lo spazio virtuale.
- Tutte queste tabelle **DEVONO** risiedere in memoria principale (immaginatevi andare ogni volta sulla memoria secondaria, overhead alto). Possibile un caricamento su domanda, dove le tabelle rimangono in swap-area fino a quando non si ha la prima richiesta. Consideriamo tra i controlli quello del bit di presenza P.
    - Nel caso di bit $P = 0$ nella tabella dei segmenti abbiamo che la tabella delle pagine relativa al segmento non è posta in memoria (*segment fault*). Dobbiamo recuperare in swap-area la tabella delle pagine del segmento, o crearla nel caso di primo riferimento. Sappiamo la dimensione del segmento, dividiamo per la dimensione della pagina ottenendo così il numero di descrittori che comporrà la tabella delle pagine.
    - Nel caso di bit $P = 0$ nella tabella delle pagine il significato è quello immaginato. In quel caso segue la gestione di *page fault* vista nelle pagine precedenti.
- Il registro STBR contiene, per ogni processo, l'indirizzo di memoria dove si trova la tabella dei segmenti. Per raggiungere il descrittore di segmento che ci interessa sommiamo ad STBR il prodotto tra sg e la dimensione in byte del descrittore.
- **Test sul segmento (*, esistenza del segmento all'interno del processo)**
  Il primo test consiste nel controllare che il segmento sia effettivamente esistente (sg non deve essere maggiore del numero dei segmenti che compongono lo spazio di indirizzamento virtuale).
- **Test sullo scostamento (**, esistenza della pagina all'interno del segmento)**
  Il secondo test riguarda lo scostamento. Esso si scompone in $< pg, of >$. Supponiamo che sc sia 20 bit, e che la pagina di sistema sia di $1Kb$. I dieci bit meno significativi costituiscono l'offset, i rimanenti costituiscono l'indice di pagina virtuale. Dobbiamo verificare che il numero di pagina non superi il limite (esattamente quanto già detto quando abbiamo introdotto la paginazione). Ricordiamo che esistono due modi per gestire la cosa:
    - abortire il processo, o
    - espandere uno dei segmenti (possibile in sistemi con allocazione dinamica della memoria).

- **Traduzione di sc**
  Superati i test precedenti possiamo andare in memoria, ma dobbiamo tradurre sc visto che il segmento è paginato.
  - La tabella dei segmenti è formata da descrittori di segmenti che contengono i soliti dati, ma al posto della base si ha l'indirizzo della relativa tabella delle pagine.
  - All'interno del descrittore di pagina individuiamo il numero identificativo del frame associato alla pagina: questo, concatenato con l'offset, restituisce l'indirizzo fisico.

  Per dare un'idea, otteniamo:

$$< sg, < pg, of >> \longrightarrow < pf, of >$$

Bella tecnica, ma overhead non banale: necessari più accessi in memoria per raggiungere sia la tabella dei segmenti che la tabella delle pagine. Consideriamo anche che con la cache aumenta l'overhead al cambio di contesto. La preferenza va verso i sistemi a thread, dove la cache rimane valida nel caso di thread appartenenti a un particolare processo.

Ricordarsi che la paginazione è trasparente al programma, mentre la segmentazione non lo è!

7.5.5 Problemi della rilocazione dinamica e soluzioni proposte (in primis paginazione a più livelli)

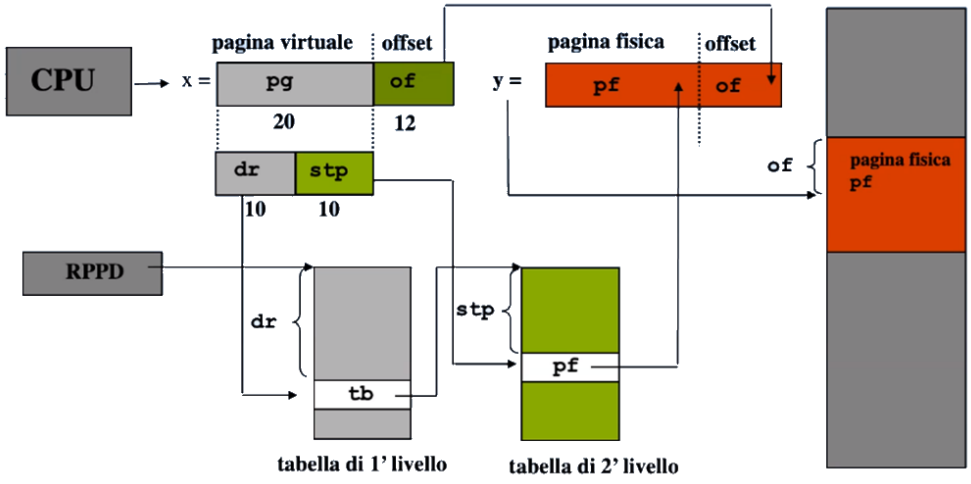
Emergono due questioni abbastanza importanti nella rilocazione dinamica.

- Spazi condivisi, associare aree di memoria a segmenti/frame di più processi implica condividere righe di codice contenenti gli stessi indirizzi virtuali. Logicamente la traduzione di questi indirizzi virtuali e indirizzi fisici dovrà coincidere.
- Funzione di rilocazione e routine di sistema, in caso di invocazione di una routine di sistema è lecito chiedersi a quale spazio di indirizzamento virtuale appartengano questi indirizzi.

Inizialmente si è riservato uno spazio di indirizzamento al solo sistema operativo, ma la soluzione appesantisce: sia per il cambio di funzione di rilocazione sia per la necessità, per un processo, di generare indirizzi relativi a spazi di indirizzamento diversi.

Soluzione adottata.

- **La prima soluzione adottata** è l'espansione (già annunciata) dello spazio di indirizzamento virtuale a un punto tale da poter contenere sia gli indirizzi del sistema operativo che gli indirizzi relativi al singolo processo. La cosa permette di evitare il cambio di funzione di rilocazione.
- **Necessaria una seconda modifica**, visto che a causa della precedente le tabelle delle pagine assumono dimensioni enormi (quelle dei segmenti non arriveranno mai a dimensioni tali). Risolviamo con la paginazione a due livelli:
  - Abbiamo una CPU che genera un indirizzo virtuale x costituito da pagina virtuale offset.
  - Per ogni spazio possiamo avere fino 2^{20} pagine virtuali, dunque la dimensione della tabella sarà $2^{20} \cdot 4$ byte (supponendo che ogni descrittore di pagina sia grande 4 byte).
  - L'idea sta nell'organizzare la tabella in sotto-tabelle. Abbiamo un indice dr relativo alla tabella di primo livello e un indice stp relativo a una tabella di secondo livello.
  - Nella tabella di primo livello abbiamo il puntatore alla pagina che contiene la sotto-tabella. All'indirizzo base sommiamo stp per ottenere il descrittore della sotto-tabella che ci interessa. A quel punto individuiamo l'identificativo del frame, che concatenato all'offset dell'indirizzo virtuale iniziale mi permette di ottenere l'indirizzo fisico.



7.6 Alcuni problemi del libro

Primo problema. *In un sistema che alloca la memoria mediante partizioni, in un certo istante sono disponibili le seguenti partizioni libere, per ciascuna delle quali sono di seguito indicate le caratteristiche.*

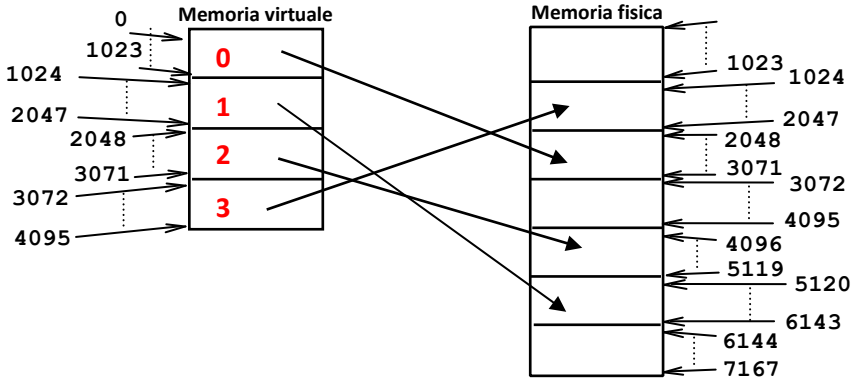
Partizione	Indirizzo base	Dimensione
1	$I_1 = 16384$	$D_1 = 512$
2	$I_2 = 65536$	$D_2 = 4096$
3	$I_3 = 25088$	$D_3 = 16384$
4	$I_4 = 1024$	$D_4 = 8192$
5	$I_5 = 4172$	$D_5 = 3072$

Quale partizione verrebbe utilizzata per allocare in memoria un segmento di dimensioni pari a 2560 byte utilizzando lo schema best-fit? E quale, se viceversa, fosse utilizzato lo schema first-fit? Per ciascuno dei due schemi, quale porzione della partizione utilizzata resterebbe disponibile?

- **Schema best-fit**. I processi sono disposti in coda nell'ordine: 1, 5, 2, 4, 3. Nella logica best-fit dobbiamo prendere il processo con la dimensione più piccola possibile in grado di soddisfare il fabbisogno del processo. La partizione 5 è la prima che soddisfa la condizione, poiché $2560 \geq I_5 = 4172$.
  - Dopo tale allocazione rimarrebbe ancora disponibile una partizione caratterizzata dai seguenti parametri: (I:44032-D:512) corrispondente alla parte porzione non utilizzata della partizione originaria.
- **Schema first-fit**. I processi sono disposti in coda nell'ordine: 4, 1, 3, 5, 2. Nella logica first-fit prendiamo la prima partizione disponibile in grado di soddisfare il fabbisogno del processo, indipendentemente dalla sua posizione. La partizione 4 è la prima che soddisfa la condizione, poiché $2560 \geq I_4 = 8192$
  - Dopo tale allocazione rimarrebbe ancora disponibile una partizione caratterizzata dai seguenti parametri: (I:3584-D:5632) corrispondente alla parte porzione non utilizzata della partizione originaria.

Secondo e terzo problema.

- *Utilizzando le tabelle indicate in figura 4.23*



specificare quali sono gli indirizzi fisici corrispondenti ai seguenti indirizzi virtuali:

$$2054, 980, 3126, 4098, 3060$$

- *In un sistema paginato la sequenza delle pagine logiche a cui un processo fa riferimento durante la sua esecuzione viene indicata come page-reference string (stringa dei parametri di pagina). Con riferimento al precedente problema e supponendo che quelli indicati siano, nell'ordine, tutti e soli gli indirizzi virtuali generati durante la sua esecuzione da un semplicissimo processo, indicare la corrispondente page-reference string*.

La dimensione delle pagine è $2^{10} = 1024$ byte. Calcoliamo le seguenti divisioni ottenendo quoziente (numero di pagina) e resto (gap tra base della partizione e indirizzo fisico).

Indirizzo virtuale	Divisione	Quoziente intero (Numero di pagina)	Resto	Indirizzo fisico
2054	$2054/1024$	***2***	$2054 - 1024 \cdot 2 = 6$	$4096 + 6 = 4102$
980	$980/1024$	***0***	$980 - 0 = 980$	$2048 + 980 = 3028$
3126	$3126/1024$	***3***	$3126 - 1024 \cdot 3 = 54$	$1024 + 54 = 1078$
4098	$4098/1024$	*4*	Errore, il numero di pagina non è valido	
3060	$3060/1024$	***2***	$3060 - 1024 \cdot 2 = 1012$	$4096 + 1012 = 5108$

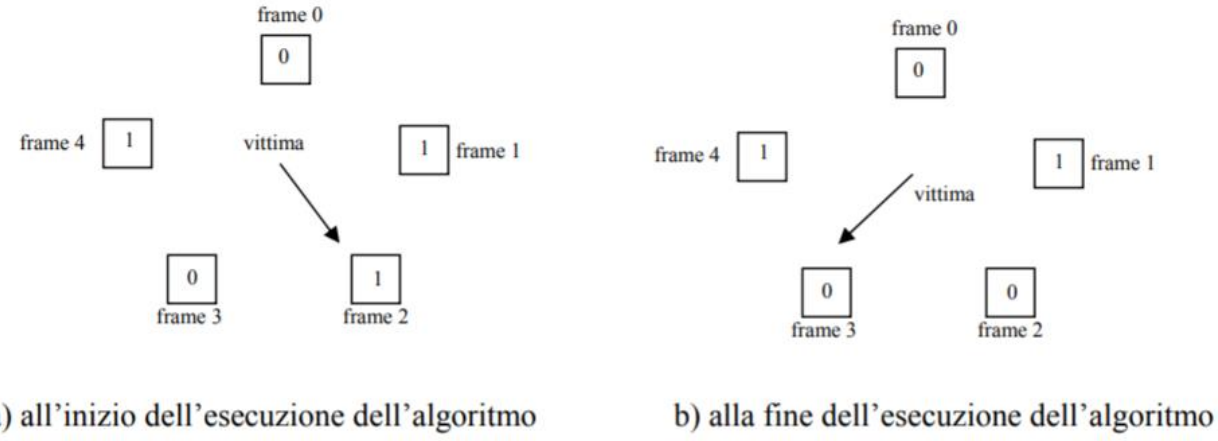
Sesto problema. *In un sistema paginato la memoria fisica è composta da 5 frame ognuna delle quali contiene una pagina logica. Nella tabella seguente sono indicati, per ciascuna frame, l'istante in cui la pagina logica in essa contenuta è stata caricata e l'istante in cui è stato fatto l'ultimo riferimento a una informazione della corrispondente pagina logica (tali istanti sono espressi in termini di timer di sistema). Inoltre, per ogni frame viene indicato il valore attuale del bit di uso U:*

Frame	*Istante di caricamento*	*Istante di ultimo riferimento*	*Bit di uso U*
0	*256*	*382*	*0*
1	*340*	*350*	*1*
2	*268*	*370*	*1*
3	*250*	*390*	*0*
4	*290*	*388*	*1*

Indicare quale frame verrà rimpiazzata al prossimo page fault utilizzando l'algoritmo di rimpiazzamento:

- *FIFO*
- *LRU*
- *Second-chance*

- In FIFO il prossimo frame rimpiazzato è il numero 3, caricato prima rispetto a tutti gli altri (250).
- In LRU, invece, il prossimo frame rimpiazzato è il numero 1, non utilizzato da più tempo (350).
- Per quanto riguarda second-chance ci accorgiamo dalla tabella che l'ultimo frame rimpiazzato è il numero 1 (istante di caricamento uguale a 340). L'algoritmo inizia a verificare i bit di uso associati alle varie frame di memoria iniziando dal frame successivo a quello che per ultima è stata rimpiazzata. Poiché l'ultimo frame che è stato rimpiazzato è quello di indice 1, come risulta dai dati presenti nella seconda colonna della tabella, l'algoritmo comincia a scandire i frame a partire da quella di indice 2 azzerando ogni bit di uso che trova al valore 1 e fermandosi al primo frame che trova col bit di uso a zero. Quindi, come indicato nella parte b) della figura, la vittima da rimpiazzare è il frame di indice 3.



a) all'inizio dell'esecuzione dell'algoritmo

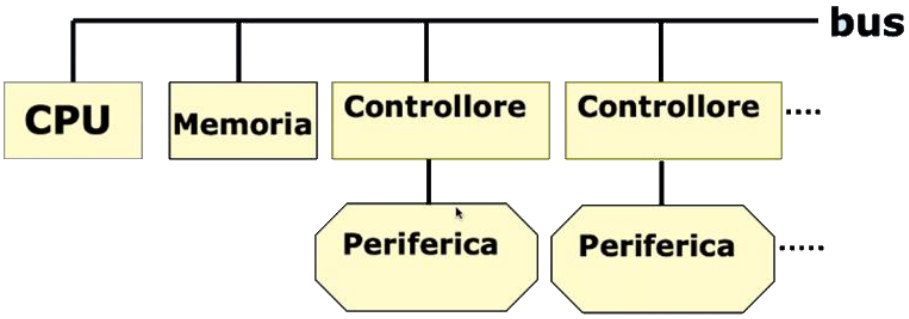
b) alla fine dell'esecuzione dell'algoritmo

8 Gestione delle periferiche

Abbiamo già detto che le periferiche sono fondamentali, permettono al calcolatore di comunicare con l'esterno (si pensi allo schermo, o allo stampante). La questione è abbastanza particolare: dobbiamo trasferire dati dalla memoria principale a un supporto esterno, o viceversa: questo significa che abbiamo bisogno di particolari istruzioni che ci permettano di comunicare con le periferiche.

8.1 Compiti del sottosistema di I/O

La complessità e le funzioni cambiano a seconda del sistema operativo. Riprendiamo la solita architettura di Von Neumann con processore, memoria principale e memorie secondarie. In questo capitolo andiamo ad aggiungere le periferiche, ciascuna accompagnata da un **controllore** (che equivalgono alle *interfacce* viste nel corso di Reti logiche).



8.1.1 Nascondere i dettagli hardware dei controllori dei dispositivi

Le periferiche sono caratterizzate da una fortissima eterogeneità: non solo le diverse tecnologie adottate, ma anche le diverse velocità con cui i dispositivi si muovono a seconda della loro natura (disaccoppiamento naturale tra azioni dei processi interni e azioni dei processi esterni). La cosa ci porta a organizzare il tutto attraverso livelli di astrazione.

Dispositivo	Velocità di trasferimento
Tastiera	10 bytes/sec
Mouse	100 bytes/sec
Modem	10 Kbytes/sec
Linea ISDN	16 Kbites/sec
Stampante laser	100 Kbytes/sec
Scanner	400 Kbytes/sec
Porta USB	1.5 Mbytes/sec
Disco IDE	5 Mbytes/sec
CD-ROM	6 Mbytes/sec
Fast Ethernet	12.5 Mbytes/sec
Monitor XGA	60 Mbytes/sec
Ethernet Gigabit	125 Mbytes/sec

Ogni periferica, ricordiamo, è specifica per una particolare funzione di trasferimento di informazione da calcolatore a mondo esterno (e viceversa). Spesso le periferiche sono indicate con trasduttore, nome che deriva dalla cultura elettronica dove un fenomeno del mondo esterno viene convertito in una tensione. La tensione prelevata dal trasduttore viene convertita in numeri attraverso un convertitore A/D (quello visto a Reti logiche). Il segnale analogico viene campionato scegliendo una frequenza (il segnale analogico è continuo e non trattabile). Altra cosa da indicare è il numero di bit con cui rappresentiamo il risultato (si pensi a quanto già visto sul convertitore A/D).

Il controllore interagisce con la periferica attraverso un bus con varie linee (per esempio di controllo), che adotta un particolare protocollo. Il controllore non vuole vedere come è fatto quel bus: ci pensa il controllore, **che assieme agli altri controllori uniforma la visione delle periferiche** (nasconde i dettagli, inclusa la velocità del dispositivo – si osservi dalla tabella come possano esserci differenze rilevanti).

Con il bit di controllo possiamo definire un'operazione di lettura e una di scrittura, operazioni esercitate su elementi di memoria indirizzabili. Nella visione tradizionale del calcolatore si distingue lo spazio di indirizzamento di memoria dallo spazio di indirizzamento dedicato a tutte le periferiche[11], che ci dà visibilità sull'insieme dei registri delle interfacce, implementati dal controllore. Nella nostra visione abbiamo:

- **registro di controllo**, dove la CPU scrive per lanciare comandi alla periferica;

[11] Oggi la cosa è meno marcata, noi continuiamo a ragionare come in principio per comodità.

- **registro di stato**, utilizzato dal controllore per rappresentare l'esito di un comando precedentemente richiesto dalla CPU attraverso il registro di controllo;
- **registro buffer**, utilizzato per trasferire i dati (può essere registro in sola lettura, in sola lettura o entrambi, abbiamo rispettivamente periferiche input, periferiche output e periferiche bidirezionali).

A questo punto diventa automatico definire il modo con cui avviene l'interazione tra processore e periferica (il processore usa i registri nel controllore per lanciare comandi, ribadiamo).

Distinzione tra processo interno e processo esterno. Noi chiamiamo processo l'esecuzione di un programma da parte della CPU, chiameremo processo anche l'insieme delle azioni compiute da un controllore per la sua periferica. Distinguiamo il **processo interno**, cioè il processo come visto fino ad ora, dal **processo esterno**, letteralmente una rappresentazione del dispositivo all'interno del nostro calcolatore).

8.1.1.1 Sincronizzazione tra processo e processo esterno: disaccoppiamento temporale

Abbiamo introdotto il concetto di processo esterno per distinguere l'attività del controllore di periferica da quella del processore, che possono lavorare letteralmente in parallelo. Parliamo di sincronizzazione a causa della differenza di velocità: è diversa la velocità tra periferiche, è diversa la velocità tra periferica e processore. Il processore è molto più veloce di qualunque periferica: se non interveniamo la lentezza delle periferiche annulla i vantaggi della velocità del processore.

- Consideriamo come esempio una tastiera che produce un byte alla volta con una velocità inferiore a quella del processo (quella delle dita): ho una bassissima velocità a fronte di un processore decisamente più veloce.
- Il disaccoppiamento temporale si ottiene imponendo la sospensione del processo che invoca il trasferimento: abbiamo un sistema multiprogrammato, non possiamo tenere la CPU in *busy wait*. Fare questo risolve i problemi della lentezza delle periferiche, visto che il processore si mette a fare altro durante l'attesa.
- Il processo esterno è colui che determina la sospensione del processo e la successiva riattivazione.

8.1.1.2 Disaccoppiamento spaziale con introduzione di buffer intermedio (bufferizzazione)

Abbiamo detto che le periferiche sono eterogenee in velocità di trasferimento, ma lo sono anche rispetto al cosiddetto blocco fisico: esso consiste nell'unità di trasferimento della periferica (si va da singoli byte a centinaia di byte). L'eterogeneità è dovuta anche al fatto che l'utente potrebbe richiedere, attraverso le apposite primitive, trasferimenti di dimensione decisamente minore a quella del blocco fisico.

- Poniamo un buffer tra tastiera e spazio dei dati del processo, che permette di fare entrambi i disaccoppiamenti. Il buffer è inizialmente vuoto, viene riempito dalla tastiera secondo la velocità tipica dell'utente, solo quando questo buffer sarà pieno la tastiera si occuperà di sincronizzarsi col processo interno avvertendolo che è pronto un certo insieme di dati (che possono essere prelevati).
- Attenzione ai dispositivi a blocchi, che richiedono trasferimento di grandi porzioni di byte: la cosa richiede tempo e buffer adeguatamente dimensionati.

Si osservi che la bufferizzazione contribuisce anche al disaccoppiamento temporale: avere un buffer intermedio rende più flessibile il passaggio dei contenuti all'utente! Possiamo limitare, ad esempio, il numero di trasferimenti all'utente facendoli solo quando strettamente necessario.

8.1.2 Definire lo spazio dei nomi con cui identificare i dispositivi

I dispositivi possono essere rappresentati attraverso nomi simbolici. Sempre i dispositivi vengono rappresentati in UNIX attraverso file speciali (la cosa è vantaggiosa, possiamo sfruttare il file system anche per i dispositivi).

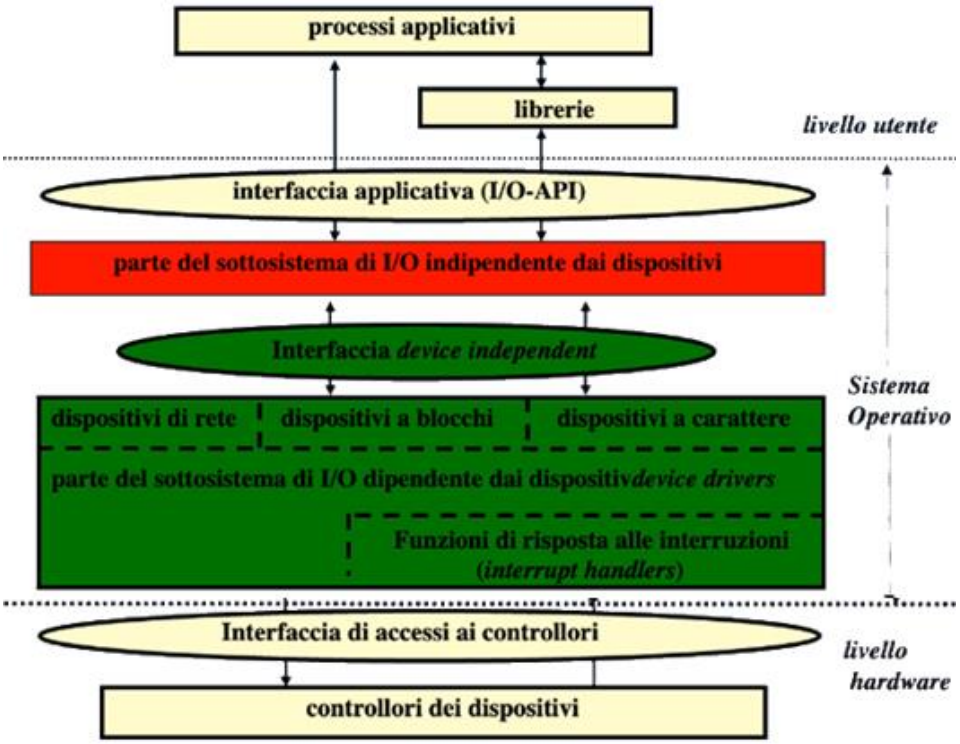
Si ha quindi un livello simbolico: il sottosistema di I/O deve convertire il nome simbolico in un identificatore univoco (tipicamente un intero) che rappresenta il dispositivo all'interno del sistema operativo.

8.1.3 Gestione dei malfunzionamenti

La gestione dei malfunzionamenti dipende fortemente dal tipo di periferica: anche in questo caso abbiamo una situazione di forte eterogeneità. Un malfunzionamento, in generale, deve poter essere sollevato dal livello in cui si genera fino al livello inferiore in cui deve essere gestito (la cosa è spiegata meglio più avanti).

8.2 Introduzione alla struttura logica del sottosistema di I/O

Livello utente e interfaccia I/O-API.



Il livello più alto è il livello utente, dove abbiamo processi applicativi (programmi da eseguire scritti in un qualunque linguaggio) che fanno uso di primitive note al programmatore per l'accesso al sottosistema di I/O.

- **Interfaccia I/O-API**. Queste primitive sono raggiungibili in modo diretto attraverso l'interfaccia applicativa *I/O-API*. `read` e `write`, ad esempio, sono primitive definite in questa interfaccia. La cosa viene fatta a un livello tale da rendere i dispositivi indistinguibili (si propone una sintassi valida per tutti i dispositivi, si richiama un qualcosa del sottosistema di I/O indipendente dai dispositivi). Gli unici elementi richiesti sono un puntatore a un buffer, e il numero di byte da trasferire.
- **Librerie**. I linguaggi mettono a disposizione librerie che permettono di fare operazioni un po' più complesse su I/O. A loro volta evocano le primitive definite dall'interfaccia applicative (livello di astrazione ancora più elevata, cosa che può essere molto comoda per il programmatore).

Livello Sistema operativo.

Nello strato del sistema operativo abbiamo…

- **La parte del sottosistema di I/O indipendente dai dispositivi**, contiene il codice implementativo delle funzioni definite dall'interfaccia applicativa I/O-API. Nella stessa parte troveremo il codice che esegue funzioni comuni ai dispositivi, prima fra tutte il *naming* (traduzione da nome simbolico a nome univoco scelto dal sistema).
- **Interfaccia *device independent***. La parte che riguarda la sincronizzazione col processo esterno sono implementate a loro volta da funzioni interne non visibili al programmatore, definite da un'interfaccia interna al sistema operativo. Diciamo *independent* perché è definita a favore del sottosistema di I/O indipendente dai dispositivi.
- **Area *device dependent***. Sotto l'interfaccia verde abbiamo quella parte del sottosistema di I/O dipendente dai dispositivi, costituita a partire dai *device driver*.
    - Le parti tratteggiate ricordano che nell'organizzare questa parte potrebbe tornare utile distinguere dispositivi di rete (vari tipi di rete, da Ethernet a Wi-fi e Bluetooth), dispositivi a blocchi (principalmente le memorie di massa) e i dispositivi a carattere (dispositivi di natura più semplice, direttamente interfacciati come l'essere umano – si pensi a tastiere, schermi e stampanti).
    - Il device driver si divide tipicamente in due componenti (dal punto di vista del codice):
        - la parte che inizializza il dispositivo;
        - la parte contenente le funzioni di risposta lanciate dai dispositivi attraverso le interruzioni.

Livello hardware.

Il livello più basso è il solito livello hardware: abbiamo un'ulteriore interfaccia per l'accesso ai controllori. Per quel che ci riguarda l'interfaccia è banalmente costituita da quei tre registri posti nello spazio di I/O.

8.3 Funzioni del livello indipendente dai dispositivi

Fanno parte di questo livello tutte quelle funzioni che sono indipendenti dai particolari dispositivi. Queste funzioni hanno lo scopo di:

- definire l'interfaccia applicativa I/O-API;
- interfacciarsi con la componente *device dependent* del sottosistema di I/O.

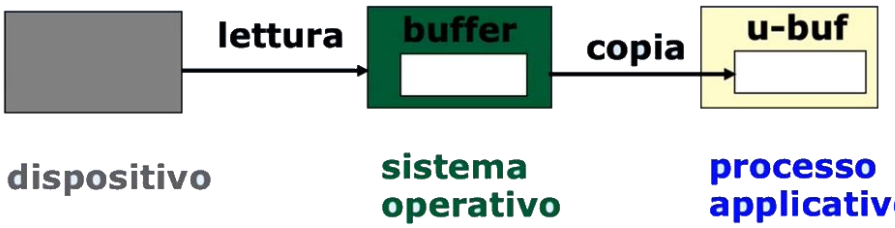
In questo livello indipendente individuiamo l'implementazione di diverse funzioni.

8.3.1 Naming

Abbiamo già rammentato il naming, esso consiste nella traduzione da nome simbolico a identificatore univoco di una periferica. Tipicamente l'identificatore univoco è un numero, oppure un puntatore (come in UNIX). Il nome simbolico segue uno schema che può essere quello del file system (cosa fondamentale, poter usare il file system significa utilizzare sulle periferiche tutti i meccanismi di protezione e controllo degli accessi tipici del file system).

8.3.2 Buffering

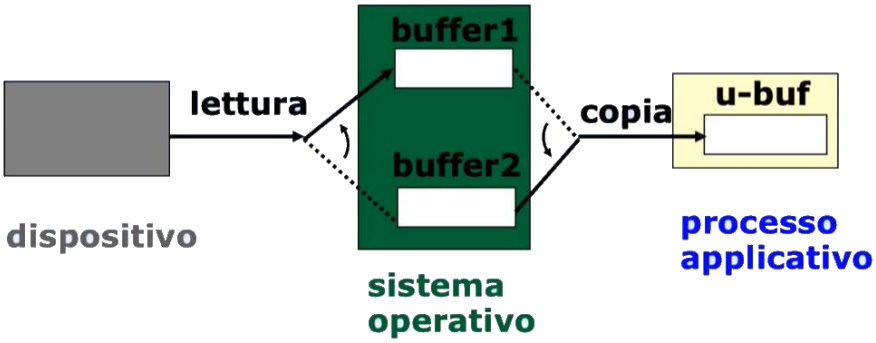
Ci aspettiamo delle primitive `read` e `write` che conterranno come parametro di ingresso un puntatore a buffer allocato nello spazio virtuale del processo applicativo.



Il sottosistema di I/O si occupa del trasferimento di un certo numero di byte dal dispositivo al buffer (e viceversa). Per risolvere i problemi precedentemente descritti facciamo disaccoppiamento temporale e spaziale attraverso l'introduzione di un buffer intermedio, memorizzato nell'area di memoria riservata al sistema operativo: nel caso della read non avremo una copia diretta del contenuto letto nel buffer *u-buf* indicato dall'utente attraverso l'apposito parametro della primitiva, ma un passaggio intermedio.

- I due buffer in generale hanno dimensione diversa: il primo risponde alle esigenze del dispositivo e il secondo a quelle della diapositiva.
- Il livello indipendente dai dispositivi deve collegare *u-buf* (indicato dall'utente quando ha invocato la primitiva) con il buffer dedicato a quella periferica, allocato nello spazio del sistema operativo. Il contenuto del buffer di sistema viene copiato in *u-buf* quando ritenuto opportuno. Il processo applicativo viene risvegliato quando *u-buf* è pieno, oppure quando il numero di byte richiesto è stato trasferito.
- L'introduzione del buffering permette di risolvere una questione importante relativa allo swapping. Supponiamo di avere un buffer memorizzato nell'area riservata al processo, che ha invocato la primitiva: cosa succede se dopo la sospensione del processo (dovuta a operazione di I/O) il processo viene sottoposto a swap-out e swap-in? Il processo potrebbe essere ricollocato a indirizzi fisici diversi. La rilocazione dinamica permette di alleviare il problema (basterebbe sostituire l'indirizzo fisico associato al corrispondente indirizzo virtuale), ma non possiamo darla per scontato in qualunque sistema operativo.
- Ulteriori vantaggi:
    - posso richiedere un numero di byte inferiore alla dimensione del blocco in caso di lettura, copio nel buffer di sistema tutto il blocco e pongo in u-buf solo i byte richiesti dai parametri della primitiva;
    - i buffer di sistema possono essere usati come memorie cache, se più processi richiedono byte diversi relativi a uno stesso blocco fisico allora la lettura del blocco fisico avverrà una volta soltanto.

Ottimizzazione con buffer in parallelo. Possibili varianti sull'implementazione, che tendono ad aumentare il parallelismo. Potrei introdurre un doppio buffer: mentre leggo uno si fa trasferimento nell'altro, e viceversa.



8.4 Gestione di malfunzionamenti / eventi anomali

Durante un'operazione di I/O dobbiamo gestire l'eventuale manifestarsi di eventi anomali.

- Non parliamo per forza di malfunzionamento: possono manifestarsi eventi anomali che necessitano un intervento affinchè le operazioni possano avere successo.
- La gestione di questi eventi non è competenza di una particolare componente del sottosistema di I/O: il luogo in cui verrà gestito un certo evento dipende dalla natura dell'evento stesso. La regola fondamentale è **gestire gli eventi il più vicino possibile a dove si sono manifestati, propagandoli ai livelli superiori solo se non è stato possibile gestire l'evento in modo adeguato**. Il fatto che un errore si propaghi fino al livello più alto implica che l'errore non è transitorio, ergo non è risolvibile oppure è necessario l'intervento umano per risolverlo.
- **Esempi**.
    - Errori rilevati a livello di controllore sono errori in generale transitori, causati da cose come polvere sulla testina o disturbo elettromagnetico: non conviene propagare l'errore a livelli superiori, basta ripetere l'operazione. I livelli superiori non si accorgeranno della ripetizione dell'operazione (se la cosa è transitoria il tentativo successivo avrà successo).
    - Un esempio di evento propagato fino al livello più alto è quello della mancanza di carta nel vassoio: la cosa potrà essere risolta solo con intervento umano (si arriva a livelli superiori e si segnala all'utente con popup che manca la carta).

8.5 Gestione delle richieste dei processi ai dispositivi

Nella sottosistema I/O *device dependent* abbiamo la distinzione tra dispositivi di rete, dispositivi a blocchi e dispositivi a caratteri: le specificità nell'accesso ai dispositivi sono definite qua.

- **Creazione delle richieste e gestione delle code**.
  Nel sottosistema device-independent vengono create le richieste di accesso ai dispositivi, e poste in coda.
- **Gestione delle singole richieste**.
  Il modo in cui vengono gestite le code dipende dal dispositivo, quindi la gestione è in mano al sottosistema *device dependent*. Al sistema possono arrivare più richieste da parte di processi concorrenti: non è conveniente decidere lo scheduling brutalmente, dobbiamo tenere conto dei singoli dispositivi.
- **Ottimizzazione: *spooling***.
  Alternativa all'allocazione dinamica di un dispositivo è il cosiddetto *spooling*: nel richiedere l'uso di un dispositivo, già occupato, non si opera direttamente sul dispositivo ma si crea un suo equivalente virtuale e privato del processo. L'esempio si ha con la stampante: quando la stampante è spenta si crea un *file di spool* e la richiesta effettiva al dispositivo non viene lanciata subito, verrà fatto al momento dell'accensione della stampante.

8.6 Visione funzionale del dispositivo (dal punto di vista del sottosistema indipendente)

Cominciamo a parlare della parte verde dello schema, in particolare la visione funzionale rispetto ai registri del dispositivo. L'interfaccia standard di una funzione che rappresenta il device driver è

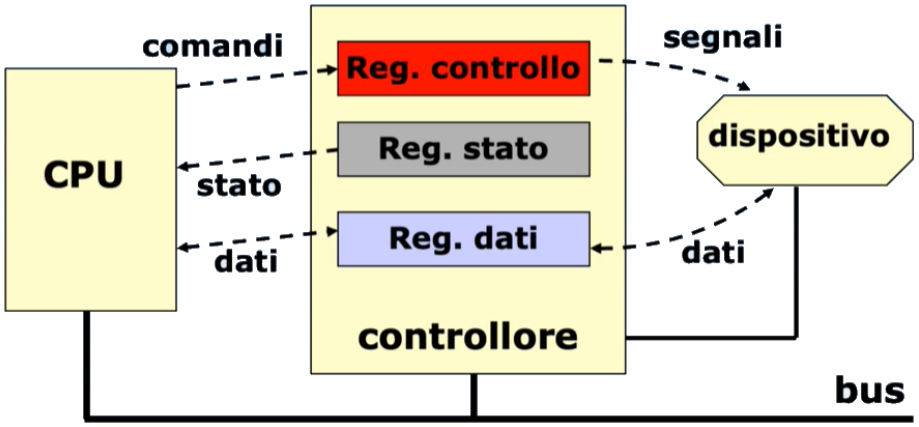
```
N = read(disp, buffer, nbytes)
```

- `disp` è il nome unico del dispositivo (**NON** il nome simbolico, il nome simbolico è indicato dall'utente a livello applicativo)
- `buffer` è il buffer di sistema (**NON** l'u-buf indicato dall'utente a livello applicativo).
- Si chiede il trasferimento di `nbytes` byte.

La funzione restituirà `N`, cioè il numero di byte effettivamente letti (ho raggiunto l'*end of file* oppure qualcosa è andato storto, solitamente errori grossi sono segnalati restituendo -1).

8.6.1 Schema semplificato di un dispositivo nell'esecuzione della funzione

Supponiamo che tutti i controllori, indipendentemente dal dispositivo, presentino sempre la stessa interfaccia funzionale.



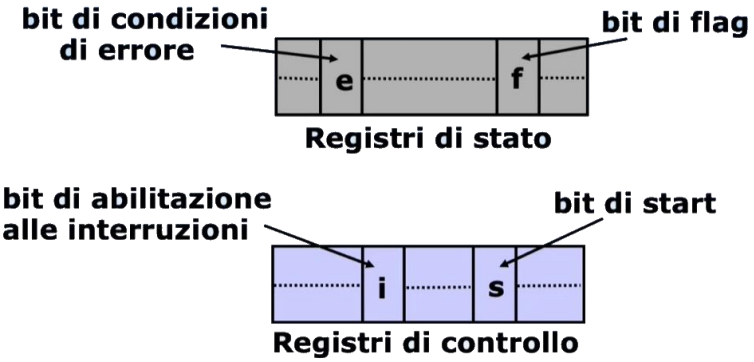
- La CPU scrive i comandi sul registro di controllo, la modifica dello stato del registro controllo è l'evento che "triggera" l'attività del controllore (da inizio al processo esterno).
- Il contenuto posto nel registro controllo viene convertito in un segnale da trasmettere al dispositivo. Il dispositivo, che ha i suoi tempi, a un certo punto restituisce dati.
- I dati restituiti vengono posti nel registro dati e successivamente letti dalla CPU.

Il controllore deve verificare che tutto sia avvenuto in modo corretto, e intervenire in caso di errore. In presenza di dati si altera il registro di stato (è lì che si indica se tutto è andato bene o no).

8.6.1.1 Soluzioni per controllare lo stato delle periferiche e bit posti nei registri di stato e di controllo

Prima soluzione per capire se ci sono novità è fare *pooling*, si controlla periodicamente il registro di stato e appena lo si vede modificato si agisce. La cosa è vietata nei sistemi multiprogrammati, dunque ricorriamo al secondo modo: le interruzioni!

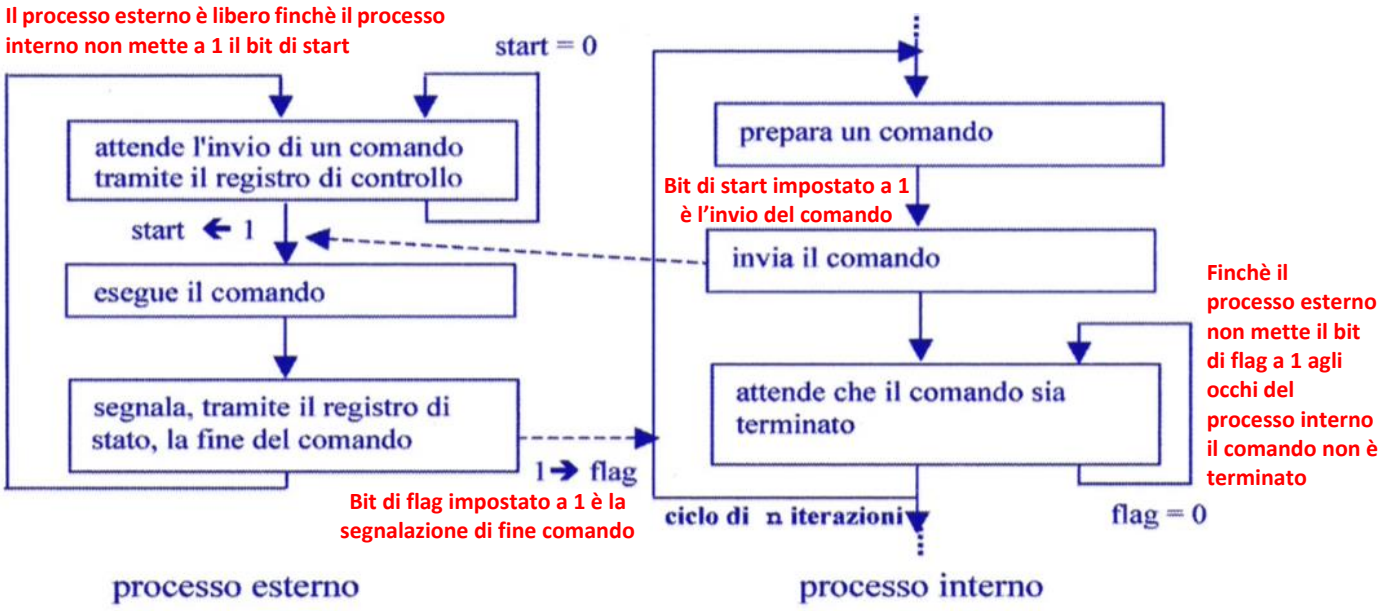
- La CPU recupera i dati dal registro dei dati, ponendoli prima nel buffer di sistema indicato nella `read` chiamata a livello *device independent*.
- Nel registro di stato abbiamo:
  - un bit che mi indica la presenza di errori,
  - un bit il cui significato dipende dal tipo di periferica (bit di flag).
- Nel registro di controllo abbiamo:
  - il bit di abilitazione alle interruzioni (necessario abilitarle in un sistema multiprogrammato),
  - il bit di start con cui diamo il via all'operazione.



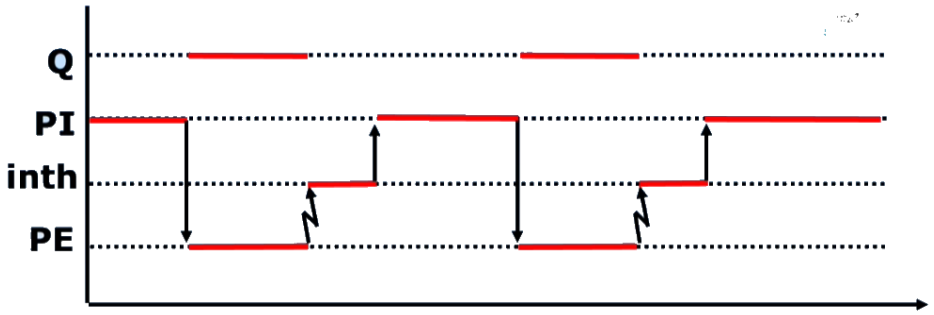
8.6.1.2 Schemi con azioni di processi esterni e processi applicativi

A sinistra abbiamo lo schema del processo esterno, a destra quello del processo applicativo

Possiamo unire gli schemi precedenti nel seguente modo:



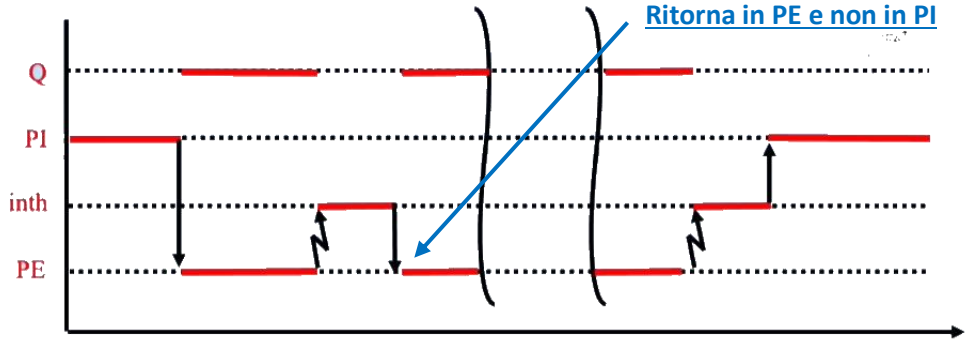
8.7 Diagramma temporale della gestione a interruzioni



Abbiamo un processo applicativo `PI` che attiva il dispositivo, un processo esterno `PE`, una routine `inth` e un altro processo applicativo `Q`.

- **Attivazione del dispositivo e quindi del processo esterno**.
  Il processo applicativo a un certo punto attiva il dispositivo (scrive nel registro di controllo nella periferica), quindi si attiva il processo esterno e il processo applicativo perde il controllo della periferica (si sospende, lo *scheduler* manda in esecuzione un altro processo applicativo).
- **Lancio dell'interruzione da parte del processo esterno**.
  A un certo punto il processo esterno finisce col trasferimento (si osservi che lavora in parallelo all'altro processo applicativo, Q), invia un'interruzione che viene recepita dalla CPU ed esegue *inth*. La routine di sistema verifica l'assenza di errori sul registro di stato e riattiva il processo interno.

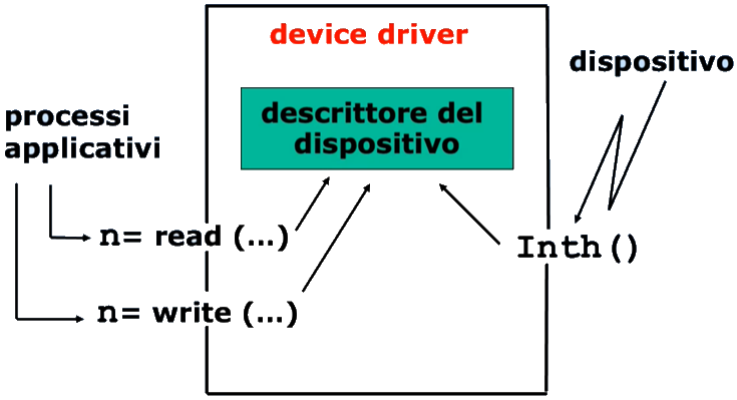
Dobbiamo considerare l'ipotesi in cui noi vogliamo trasferire tanti byte: il trasferimento si ripete dieci volte, ma non possiamo sospendere il processo applicativo dieci volte (aumenta in modo inutile l'overhead). Gestiamola nel seguente modo:



Diamo a *inth* il compito di contare il numero di byte da trasferire, riattiviamo il processo interno PI solo al termine della decima operazione.

8.8 Astrazione di un dispositivo

8.8.1 Descrittore del dispositivo



La prima cosa che dobbiamo fare quando abbiamo a che fare con una risorsa astratta è definire una struttura dati che la descriva, cioè un *descrittore di dispositivo*. Rappresentiamo non solo il dispositivo, ma anche le azioni che coinvolgono quel dispositivo: le primitive `read` e `write`, la routine `inth`…
Al suo interno troviamo….

- **Indirizzi dei tre registri** (nello spazio di I/O) che costituiscono la visione funzionale del dispositivo: registro di controllo, registro di stato, registro dei dati. Sono costanti, anche se presenza di PCI vengono determinati nella fase di bootstrap sulla base di range predefiniti (BIOS).
- Per la sincronizzazione tra processo e processo esterno abbiamo:
    - **Semaforo `Dato_disponibile`** inizializzato a 0 (all'inizio non abbiamo alcun dato disponibile).
    - **Contatore dei dati da trasferire**. Dal diagramma visto prima vediamo che il processo applicativo inizializza il caricamento specificando il numero di byte da trasferire: da quel momento si sospende e si riattiva solo al completamento del trasferimento. Vogliamo anche fare in modo che non si ritorni subito nel processo applicativo nel caso in cui ci siano più byte da trasferire. Il valore iniziale viene recuperato dai parametri di ingresso della primitiva (ricordarsi che la primitiva read appena vista viene invocata dal sistema operativo).
    - **Puntatore al buffer** in memoria (buffer di sistema, **ATTENZIONE**)
- **Esito del trasferimento**: dobbiamo tenere conto di errori/eccezioni non mascherabili a livello basso, che vengono sollevate a livello del device driver (che memorizza nel descrittore, permettendo una decisione).

8.8.2 Struttura (pseudo) del device driver

8.8.2.1 Struttura della read invocata dal livello device-dependent

Siamo a livello della parte dipendente dai dispositivi, nella read precedentemente definita.

```
int read(int disp, char* pbuf, int cont) {
    // Inizializzazioni
    descr[disp].cont = cont;
    descr[disp].punt = pbuf;

    // da questo momento in poi bit_start = 1 (e ciò
    triggera il processo esterno, il controllore inizia il suo lavoro)
    <Attivazione del dispositivo>

    // Sospendiamo il processo (quale processo? Il processo applicativo, che ha invocato una
       primitiva ad alto livello, questa eseguito un po' di codice e infine ha invocato la
       nostra read, supponiamo che il pid sia il solito poiché ci siamo limitati a un solo
       cambio di privilegio e non di contesto)
    descr[disp].dato_disponibile.wait();

    // Arriviamo qua dopo che è avvenuto il trasferimento di cont byte, se è andato
      male avremo qualcosa scritto nel campo esito del descrittore
    if(descr[disp].esito == <codice errore>) return -1; // Semplificato all'osso
    return cont-descr[disp].cont; // Restituisco il numero di byte effettivamente letti
}
```

- Inizializzo la struttura dati (pensnado in particolare ai dati passanti in ingresso).
- Attivo il dispositivo.
- Sospendo il processo a seguito dell'attivazione del dispositivo.
- Il codice seguente sarà eseguito dopo il risveglio del processo: verifico l'esito, restituisco -1 in caso di esito negativo, restituisco il numero di byte effettivamente letti altrimenti.

8.8.2.2 Routine inth eseguita a seguito del lancio di una interruzione

Funzione che risponde all'interruzione esterna lanciata dal dispositivo

```
void inth() {
    char b; // Buffer di appoggio

    <legge registro di stato fornito dal controllore>
    if(bit_errore == 0) {
        b = <lettura registro dati del controllore>;
        // sistemiamo ciò che è stato letto nel buffer di sistema
        *descr[disp].punt = b;
        descr[disp].punt++;
        descr[disp].cont--;
        if(descr[disp].cont != 0)
            <riattivazione del dispositivo>
        else {
            descr[disp].esito = <terminazione corretta>;
            <disattivazione dispositivo, se prevista>;
            descr[disp].dato_disponibile.signal(); // risveglio il processo
        }
    }
    else {
        // Zona del codice molto dipendente dall'implementazione
        <routine gestione dell'errore> // non andiamo in profondità

        if(<errore non è recuperabile>) descr[disp].esito = <codice errore>;
        descr[disp].dato_disponibile.signal(); // risveglio il processo
    }
    return;
}
```

- Per prima cosa verifico che il trasferimento sia andato a buon fine controllando il registro di stato: se il trasferimento non è andato a buon fine si lancia la routine di gestione. Nel caso in cui la routine non sia in grado di rimediare all'errore si segnala un codice di errore nell'esito e si risveglia il processo invocante.
- Se non si hanno errori gestiamo il passaggio in buffer di quanto posto nel registro dati, verifichiamo se il contatore è sceso a zero e nel caso poniamo l'esito, disattiviamo il dispositivo e risvegliamo il processo.

8.9 Analisi (con schema) del flusso di controllo durante un trasferimento

Analizziamo il flusso di controllo (i vari strati già introdotti), dal momento in cui il processo interno chiede l'accesso a una periferica I/O.

Interfaccia applicativa

Processo P_i

…
…
…

Nel momento in cui il processo P_i vuole accedere allo spazio di I/O provvederà a istanziare un buffer (u-buf) nello spazio virtuale del processo applicativo.

```
int n;
int ubufsize;
char ubuf[ubufsize];
```

A un certo punto abbiamo l'invocazione della primitiva, che appartiene all'interfaccia di I/O (API) messa a disposizione dal sistema operativo al "sistema di programmazione". Sia `IN` il nome simbolico del dispositivo:

```
n = read(IN, ubuf, ubufsize);
```

System call

A seguito della chiamata si ha la *system call*, con cui avviene cambio di privilegio da utente a kernel e disattivazione delle interruzioni (salvataggio in pila e aggiornamento opportuno del registro dei flag).

UNIX. In UNIX il soggetto è una terna dei valori *pid-uid-gid*. Quello che facciamo è cambiare il soggetto, alterando solo uid-gid.

…
…
…

Sistema operativo *device-independent*

Siamo passati da lato applicativo a lato sistema operativo *device-independent*, attraversando l'interfaccia applicativa. Eseguiamo la primitiva read invocata dal processo: al suo interno definiamo il buffer di sistema, controlliamo i permessi, facciamo naming e invochiamo il device driver (quindi della read che abbiamo definito poco indietro)

```
int read(device dp, char* punt, int cont) {
  int n, D;

  // N dimensione ideale per ottimizzare il
     trasferimento con la periferica (dimensione
     del blocco logico periferica)
  char buffer[N];

  < controllo accessi >;

  < naming per la traduzione dell'identificatore
   simbolico nell'identificatore univoco dp <-> D >;

  n = _read(D, buffer, cont);

  <trasferimento dati da buffer a ubuf>;

  return <codice di errore, in generale numero di byte letti>;
}
```

Interfaccia *device-independent*

Sistema operativo *device-dependent*

Abbiamo attraversato l'interfaccia *device-independent* (interna, non visibile al programmatore ma necessaria per strutturare il codice del gestore I/O all'interno del sistema operativo) ponendoci nella parte *device-dependent*:

```
int _read(int disp, char* pbuf, int cont) {
    <inizializza il descrittore del dispositivo>;
    <attiva il dispositivo>;
    <sospende il processo>;
    return byte_letti;
}
```

L'attivazione del dispositivo avviene lanciando istruzioni dell'interfaccia hardware. L'attivazione del dispositivo da inizio all'attività del processore esterno (controllore della periferica). Il device driver, dopo aver attivato il dispositivo, sospende il processo (e viene posto in coda, nel semaforo dedicato alla periferica).

A un certo punto il dispositivo lancia un'interruzione esterna, che gestiamo attraverso l'interrupt handler *inth*

Interruzione esterna

```
void inth() {
    <trasferimento dati in buffer>;
    <riattivazione del processo>;
}
```

Il lancio dell'interruzione esterna con esecuzione di inth può ripetersi diverse volte, al più cont volte.

Interfaccia Hardware

HARDWARE

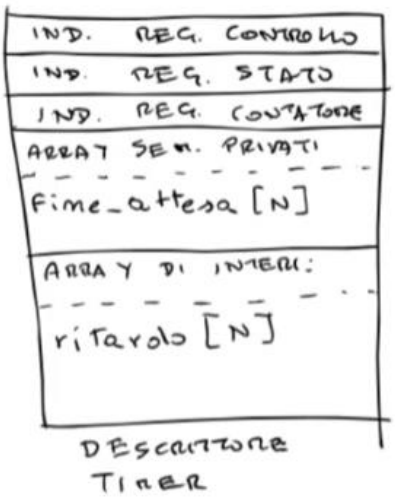
Ogni volta che eseguiamo return torniamo al livello superiore. Attenzione: non abbiamo descritto tutto, dovrebbe essere ovvio che quando sospendiamo il processo il descrittore dello stesso va nella coda del semaforo dedicato alla periferica. A quel punto la CPU è libera e viene assegnata a un nuovo processo con lo scheduler. Il processo esterno si trova sotto l'interfaccia hardware e viene attivato dalla `_read`.

8.10 Esempi di periferiche

8.10.1 Timer

Il timer è un orologio che oscilla secondo una certa frequenza, programmabile per lanciare interruzioni esterne. Queste interruzioni sono utilizzate dai processi per generare eventi che possono avere natura periodica. Agli eventi si associano comportamenti dipendenti dalle applicazioni.

Esempio con Round Robin. Un esempio col sistema operativo è lo *scheduling*: si pensi a Round-Robin. Il timer fa sì che n processi possano sospendersi su un semaforo ed essere riattivati un quanto di tempo che costituisce il parametro della primitiva invocata.



8.10.1.1 Descrittore del timer

La prima cosa che ci serve è un descrittore per il timer. Il descrittore presenta le strutture dati necessarie per la primitiva `delay`, che implementeremo poco più avanti.

- Per prima cosa abbiamo gli indirizzi ai registri che costituiscono la visione funzionale: registro di controllo, registro di stato e registro contatore (registro dati).
- Un array per i semafori privati `fine_attesa[N]`
- Array di interi `ritardo[N]` per ricordarsi il tempo per cui ciascun processo ha chiesto di essere sospeso

Si suppone per comodità che N sia il numero dei processi.

8.10.1.2 Esempio di codice del device driver: implementazione della `delay`

8.10.1.2.1 Codice della primitiva

Supponiamo esista una primitiva `delay` che il processo può invocare per sospendersi per un determinato periodo.

```
void delay(int n) {
    int proc;
    proc = <indice proc in esecuzione>;
    descr.ritardo[proc] = n;
    descr.fine_attesa[proc].wait();
}
```

- Recupero in qualche modo (per esempio con una primitiva) l'identificativo del processo in esecuzione.
- Abbiamo detto che abbiamo nel timer array di dimensioni pari al numero di processi.
- Uso come indice l'identificativo del processo e aggiorno i dati necessari: pongo il ritardo uguale a quello posto in ingresso e sospendo il processo col relativo semaforo. La funzione finisce subito dopo la `wait`: non abbiamo bisogno di riattivare il dispositivo (**il timer in realtà è sempre attivo**).

8.10.1.2.2 Routine *inth*

Il timer, come tutte le periferiche, lancerà delle interruzioni esterne. Supponiamo che n faccia riferimento al numero di periodi durante i quali vuole rimanere sospeso. Come è fatto il gestore dell'interruzione esterna `inth`?

```
void inth() {
    for(int i = 0; i < N; i++) {
        if(descr.ritardo[i] != 0) {
            descr.ritardo[i]--;
            if(descr.ritardo[i] == 0)
                descr.fine_attesa[i].signal();
        }
    }
}
```

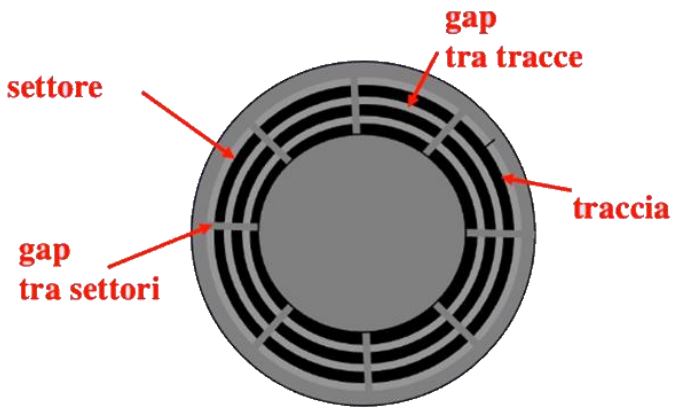
- Scorro con un array tutti i possibili processi sospesi.
- Capisco se un processo è sospeso solo se ha un ritardo diverso da zero nel relativo elemento dell'array.
- Si decrementa il ritardo.
- Se il decremento porta ad avere un ritardo uguale a zero allora il tempo indicato dal processo è scaduto e quindi possiamo risvegliare il processo con la `signal`.

Attenzione, abbiamo problemi di scalabilità: se N è abbastanza grande potrebbe emergere un ritardo significativo dovuto proprio al ciclo.

Interruzioni mascherate. Altro dettaglio è che l'interruzione esterna potrebbe essere servita non subito, potrebbe arrivare quando le interruzioni sono mascherate (durante l'esecuzione di una primitiva di sistema).

8.10.2 Dischi magnetici

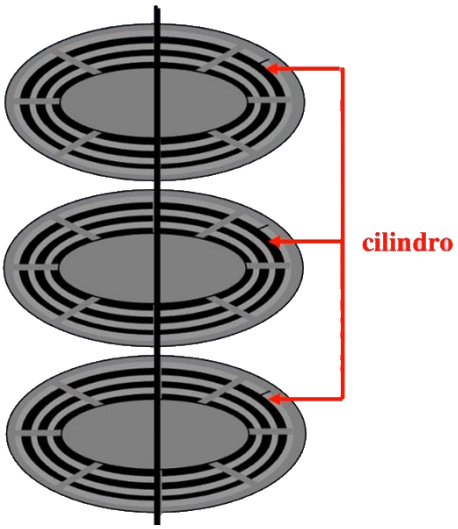
I dischi magnetici sono un tipo di periferica ancora utilizzata, anche se progressivamente superati. Abbiamo una risorsa fisica, un disco, a cui dobbiamo garantire l'accesso in mutua esclusione. È compito del device driver attivare il dispositivo, cioè inviare alla periferica un comando da eseguire: nel caso del disco si invia un comando più complesso, che indica un particolare settore del disco (o un insieme di settori del disco) del file a cui si vuole accedere. Lo scheduling dipende molto dalla natura meccanica del disco.



In foto abbiamo un disco singolo, che può avere una o due facce magnetizzate secondo un particolare pattern: le tracce (che disegniamo con cerchi concentrici).

- Le tracce sono separate dal *gap delle tracce*, materiale amagnetico che permette di distinguere le tracce magnetiche.
- Le tracce sono divise a loro volta in settori, che corrisponde all'unità minima utilizzata nel trasferimento: i file sono memorizzati in un certo numero di settori, ogni volta che dobbiamo leggere o scrivere sul file andremo a fare lettura e scrittura di un settore alla volta.
- I settori sono separati attraverso i *gap tra settori*, riconoscibili da una testina. La testina è un dispositivo capace di leggere o scrivere su materiale ferromagnetico. La testina può essere fissa o mobile: nel primo caso avremo una testina dedicata ad ogni traccia, nel secondo la testina si muove per posizionarsi sulla traccia interessata.
- Di un settore ci interessa conoscere la capacità: la particolarità è che tutti i settori hanno la stessa capacità al di la della loro dimensione fisica. I settori appartenenti a tracce più esterne sono più rarefatti, mentre quelli più vicini al centro del disco sono più densi.

In realtà andremo a costruire un cilindro insieme a più dischi (**disk pack**): ciascun disco può avere una o due facce magnetiche, con tracce e settori. L'organizzazione dei dischi è simmetrica.



Dal punto di vista logico possiamo immaginarci il disco organizzato come un array di settori.
I settori sono indicizzati utilizzando una terna di valori: faccia, traccia e settore. Supponiamo che il processo esegua la solita read: i parametri permetteranno al sistema operativo che esegue la system call di calcolare a quale settore fa riferimento la read stessa. Dobbiamo conoscere la dimensione in byte del file, e calcolare quanti settori mi servono per allocare il file, conoscendo la dimensione del settore del disco (non è un parametro che posso inventarmi,

dipende dal disco). Dopo aver fatto ciò devo scegliere tra tutti i settori liberi quelli in cui allocare il file (il modo in cui scegliamo i settori dipende dall'algoritmo adottato).

8.10.2.1 Tempo medio di trasferimento: seek time, rotational latency e tempo di trasferimento del settore

Supponiamo di avere un disco con testina mobile. Definiamo il tempo medio di trasferimento del settore TF

$$TF = TA + TT$$

dove TA è il tempo di attesa e TT il tempo di trasferimento del settore. Quando il disco viene formattato si registrano informazioni in grado di rendere consistente l'indicizzazione dei settori.

- Il TA consiste nella somma tra *seek time* (ST) e *rotational latency* (RL)

  $$TA = ST + RL$$

  Il primo è il tempo necessario alla testina per traslare e trovare la traccia, il secondo è il tempo necessario affinchè la testina raggiunga il settore interessato nella traccia.
- **Si fa riferimento a valori medi**: non sappiamo in quale testina si posizionerà all'inizio.
- TT non è un valore medio, ma secco! Anche questo si recupera dai datasheet.

8.10.2.2 Esercizio di esempio

Abbiamo i seguenti dati:

- Numero tracce (numero tracce sul disco): 13614
- Numero facce (numero facce per cilindro): 8
- Numero settori per faccia: 320
- Numero byte per settore: 512

Capacità del disco. Per prima cosa calcoliamo la capacità del disco

$$13614 \cdot 320 \cdot 512$$

Otteniamo una cosa dell'ordine di 20GB.

Calcolo del tempo medio di trasferimento. Abbiamo i seguenti dati:

- Seek time minimo (tempo di seek da una traccia a una adiacente): 0.6msec
- Seek time medio: 5.2 msec
- RL: 6msec (giro completo, mediamente 3msec – metà giro)
- TT: 19 microsec

Abbiamo un file di 320KB, supponiamo di avere allocazione su settori contigui. Ogni settore contiene 512 byte, per ottenere il numero di settori faccio la seguente divisione:

$$\frac{320\ Kb}{512\ byte} = \frac{320 \cdot 2^{10}}{512} = 640 \text{ settori necessari}$$

Calcoliamo il tempo di trasferimento medio del singolo settore:

$$TF = 5.2 + 3 + 6 + 0.6 + 3 + 6 = 23.8 \text{ msec}$$

- Prendo il *seek time medio* 5.2, non sappiamo dove si trova la nostra testina.
- Considero la *rotational latency*: so che per fare un giro mi servono 6 msec, in media ce ne metterò 3. Adesso ci troviamo sul primo settore dove è memorizzato il file.
- Il file occupa due tracce intere: prima di fare il trasferimento devo far passare due tracce intere sotto la testina. Dopo 6 msec ho letto metà file (320 settori).
- Devo spostare la testina per leggere la seconda traccia. Ho di nuovo un seek time, ma visto che l'allocazione è contigua non prendo il seek time medio, ma quello minimo (0.6msec, passo da una traccia a quella adiacente).
- Aspetto una *rotational latency media* (3) per vedere il primo settore della traccia e poi 6 msec per leggere l'intera seconda traccia.

Adesso supponiamo di non avere allocazione su settori non contigui. Avremo:

$$TF = (5.2 + 3 + 0.019) \cdot 640 = 5260.16\ msec$$

- Si riparte da capo. Per prima cosa abbiamo il *seek time medio* (5.2 msec), dobbiamo porci nella traccia richiesta. Si aspetta una rotational latency media (3 msec) affinchè la testina si collochi sul primo settore richiesto.
- Trasferisco un solo settore alla volta, ergo considero ogni volta TT (0.019 msec). Dobbiamo trasferire 640 settori (abbiamo detto che il file è costituito da due tracce intere), dunque il risultato della somma appena descritta va moltiplicato per il numero di settori.

8.10.2.3 Algoritmi di scheduling per la gestione delle richieste di trasferimento

8.10.2.3.1 Perché è necessario un algoritmo di scheduling?

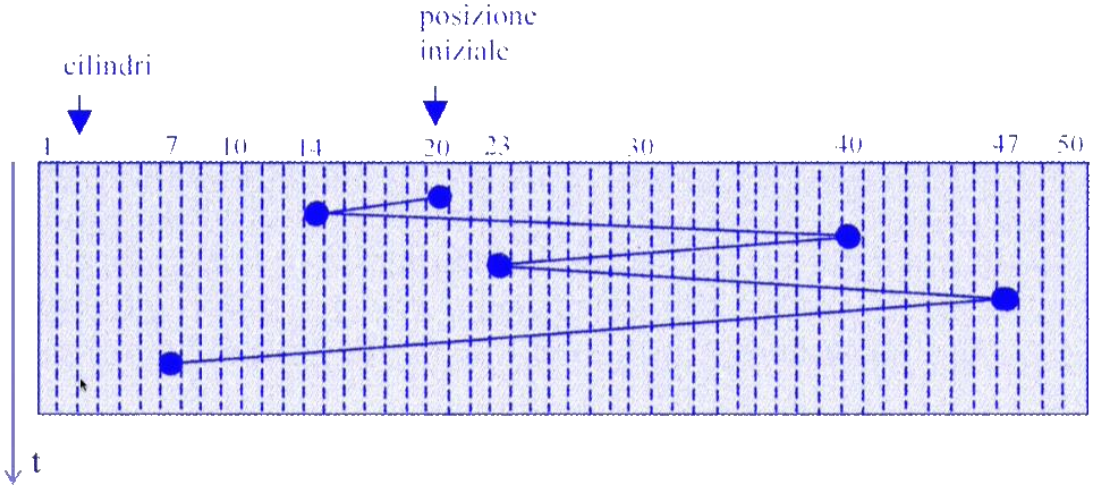
Se osserviamo i dati dell'esercizio precedente ci accorgiamo di come il seek time sia il contributo più rilevante al tempo di trasferimento medio. Abbiamo già detto che il seek time è il tempo necessario affinchè la testina si collochi nella traccia desiderata.

Le richieste di accesso al disco, provenienti dai processi applicativi, sono formulate in termini di accesso a un particolare settore (e quindi a una particolare traccia). L'algoritmo di scheduling stabilisce l'ordine con cui il disco gestirà le richieste di trasferimento: quello che fa è disporre le richieste di accesso in un ordine tale da minimizzare il movimento della testina, e quindi del *seek time*.

Nell'introduzione dei seguenti algoritmi consideriamo una posizione iniziale comune della testina (traccia 20). La metrica che adottiamo per valutare l'efficienza degli algoritmi è il numero di salti fatti.

8.10.2.3.2 Algoritmo *First-Count-First-Served*

Consideriamo il seguente grafico, che descrive il movimento della testina. Abbiamo richieste di accesso alle tracce 14, 40, 23, 47 e 7. Banalamente FCFS pone gli accessi alle tracce secondo l'ordine di presentazione delle richieste.



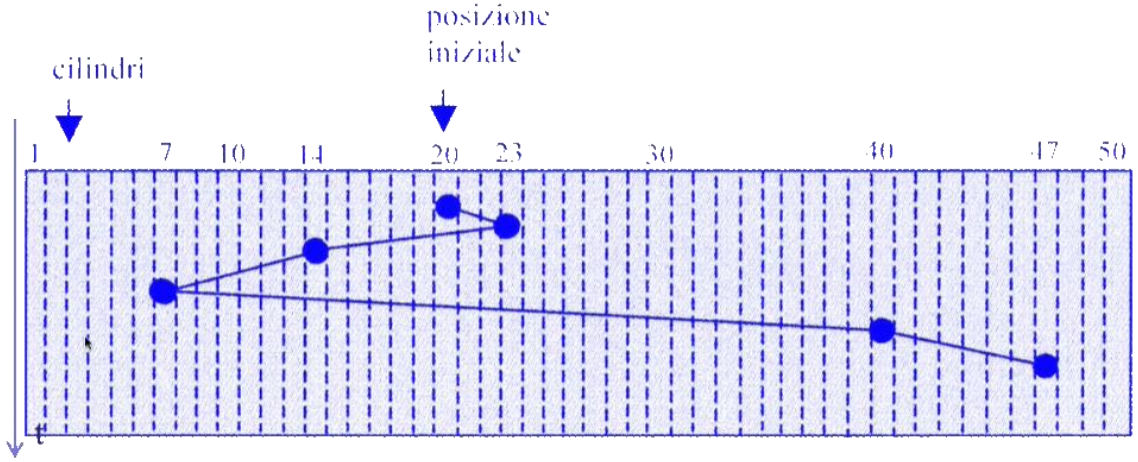
Otteniamo che la testina si è mossa sopra 93 tracce.

$$(20-14)+(40-14)+(40-23)+(47-23)+(47-7)=93$$

Dobbiamo fare di meglio: il grafico è abbastanza chiaro nel rappresentare i continui movimenti della testina.

8.10.2.3.3 Algoritmo *shortest-seek-time-first*

Abbiamo richieste di accesso alle tracce 14, 40, 23, 47 e 7. L'algoritmo SSTF ordina le richieste in base al seek time necessario per raggiungere la testina. Nel determinare l'ordine facciamo differenza (ci interessa il modulo) tra numero identificativo della traccia dove si trova la testina e gli identificativi delle altre tracce coinvolte in richieste: ci si sposta verso la traccia più vicina (modulo minimo).



Otteniamo che la testina si è mossa sopra 59 tracce.

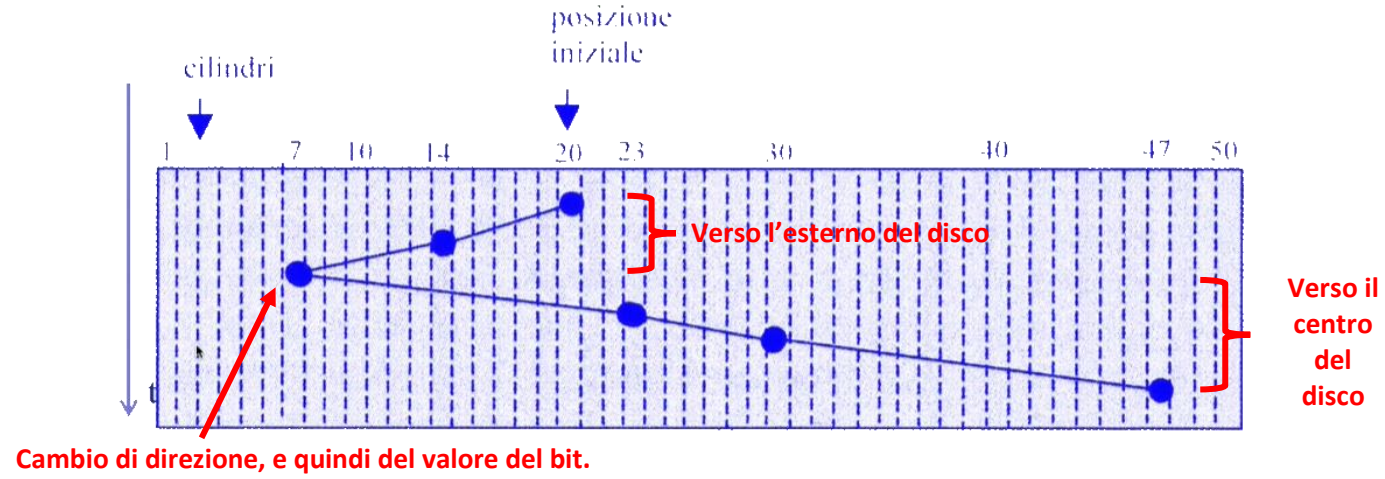
$$(23-20)+(23-14)+(14-7)+(40-7)+(47-40)=59$$

Il numero di movimenti della testina si è ridotto, tuttavia l'overhead non è più trascurabile:

- ogni volta dobbiamo scorrere tutta la coda delle richieste per individuare quella più vicina
- c'è il rischio di *starvation* (se tutte le volte entrano in scena richieste con *seek time* minore….).

8.10.2.3.4 Algoritmo *SCAN* (Algoritmo dell'ascensore)

L'algoritmo di SCAN mira a risolvere i due problemi precedenti. Dato un bit si indica la direzione assunta dalla testina (verso il bordo – numeri identificativi decrescenti - o verso il centro del disco – numeri identificativi crescenti). Abbiamo come prima le richieste di accesso alle tracce 14, 40, 23, 47 e 7. Quello che facciamo è gestire prima le richieste verso una direzione e poi quelle verso l'altra direzione. Al cambio di direzione si altera il bit precedentemente citato.



SCAN è detto anche *algoritmo dell'ascensore* per il comportamento simile agli ascensori (non è una cosa molto presente in Italia):

- l'ascensore sale e scende;
- l'utente ha due tasti per chiamare l'ascensore, uno che preme se vuole scendere e l'altro che preme se vuole salire;
- se l'ascensore sale questo gestisce solo le richieste di coloro che vogliono salire, quindi si ferma ai piani dove qualcuno ha chiamato l'ascensore col tasto per salire;
- se l'ascensore scende questo gestisce solo le richieste di coloro che vogliono scendere, quindi si ferma ai piani dove qualcuno ha chiamato l'ascensore col tasto per scendere.

Otteniamo che la testina si è mossa sopra 53 tracce.

$$(20-14)+(14-7)+(23-7)+(30-23)+(47-30)=53$$

Abbiamo un ulteriore miglioramento rispetto a SSTF.

8.11 Alcuni problemi del libro

Primo problema. *Abbiamo visto mediante pseudocodice la struttura della funzione di lettura (`read`) e della funzione di risposta alle interruzioni (`inth`) ipotizzando che la gestione del dispositivo sia sincrona (cioè che il processo applicativo si sospenda quando invoca una funzione di input in attesa che il trasferimento sia completato. Volendo implementare una gestione asincrona possiamo ipotizzare tre funzioni: la funzione `start_input`, invocata per attivare il trasferimento ma che non blocca il processo che quindi può continuare in parallelo con il trasferimento, la funzione `wait_input`, che il processo può invocare per bloccarsi se, quando invocata, il trasferimento non è completato, e la funzione `inth` di risposta alle interruzioni. Utilizzando lo stesso pseudocodice, descrivere le tre funzioni*.

L'intestazione di `start_input` è la stessa della `read`, mentre la `wait_input` (nuova funzione) richiede in ingresso l'identificativo del dispositivo. L'intestazione di `inth` è sempre la stessa, ma soprattutto l'implementazione della routine non cambia! La differenza sostanziale sta nelle due funzioni. La parte relativa alla gestione dell'esito dell'operazione di I/O è posta in `wait_input` e non in `start_input` (che è l'equivalente della `read`), inoltre la `start_input` non invoca la primitiva wait sul semaforo della periferica.

```
void start_input(int disp, char* pbuf, int cont) {
    // Inizializzazioni
    descr[disp].cont = cont;
    descr[disp].punt = pbuf;

    // da questo momento in poi bit_start = 1 (e ciò
    triggera il processo esterno, il controllore inizia il suo lavoro)
    <Attivazione del dispositivo>
}
```

```
int wait_input(int disp) {
    // Sospendiamo il processo (quale processo? Il processo applicativo, che ha invocato una
       primitiva ad alto livello, questa eseguito un po' di codice e infine ha invocato la
       nostra read, supponiamo che il pid sia il solito poiché ci siamo limitati a un solo
       cambio di privilegio e non di contesto)
    descr[disp].dato_disponibile.wait();

    // Arriviamo qua dopo che è avvenuto il trasferimento di cont byte, se è andato
      male avremo qualcosa scritto nel campo esito del descrittore
    if(descr[disp].esito == <codice errore>) return -1; // Semplificato all'osso
    return cont-descr[disp].cont; // Restituisco il numero di byte effettivamente letti

}

void inth() {
    char b; // Buffer di appoggio

    <legge registro di stato fornito dal controllore>
    if(bit_errore == 0) {
        b = <lettura registro dati del controllore>;
        // sistemiamo ciò che è stato letto nel buffer di sistema
        *descr[disp].punt = b;
        descr[disp].punt++;
        descr[disp].cont--;
        if(descr[disp].cont != 0)
            <riattivazione del dispositivo>
        else {
            descr[disp].esito = <terminazione corretta>;
            <disattivazione dispositivo, se prevista>;
            descr[disp].dato_disponibile.signal(); // risveglio il processo
        }
    }
    else {
        // Zona del codice molto dipendente dall'implementazione
        <routine gestione dell'errore> // non andiamo in profondità

        if(<errore non è recuperabile>) {
           descr[disp].esito = <codice errore>;
           descr[disp].dato_disponibile.signal(); // risveglio il processo
        }
    }
    return;
}
```

Secondo problema. *Con riferimento ai dati nella seguente tabella*

Parametro	***Valore***
Numero cilindri (numero tracce per ogni faccia)	*13614*
Tracce per cilindro	*8*
Settori per traccia	*320*
Byte per settore	*512*
Capacità	*18.3GB*
Tempo minimo di seek (tra cilindri adiacenti)	*0.6 msec*
Tempo medio di seek	*5.2 msec*
Tempo di rotazione	*6 msec*
Tempo di trasferimento di un settore	*19 microsec*

Supponiamo che si desideri leggere un file costituito da 960 record allocati in modo contiguo (i primi 160 record negli ultimi 160 settori della traccia numero 3, i successivi 320 record nella traccia numero 4, gli ulteriori 320 record nella traccia numero 5 e gli ultimi 160 record nei primi 160 settori della traccia numero 6. Valutare il tempo necessario per leggere il file in memoria.

Il tempo richiesto per leggere i primi 160 record allocati negli ultimi 160 settori della traccia N. 3 è pari al tempo di *seek*, più la *rotational latency,* più il tempo richiesto per il trasferimento dei 160 settori pari a 3 millisecondi in quanto questi settori corrispondono alla metà dei settori presenti sulla traccia e quindi necessitano di un tempo di trasferimento pari alla metà del tempo di rotazione del disco:

5.2 + 3 + 3 = 11.2 millisecondi

Per leggere i successivi 320 settori della traccia N. 4 è necessario il tempo di *seek*, che però adesso si riduce al minimo per il passaggio da una traccia alla successiva, più la *rotational latency,* più il tempo necessario per il trasferimento dei 320 settori, che ora corrisponde all'intero tempo di rotazione del disco:

0.6 + 3 + 6 = 9.6 millisecondi

Lo stesso tempo è necessario per il trasferimento dei 320 settori presenti sulla traccia successiva (la N. 5):

0.6 + 3 + 6 = 9.6 millisecondi

Infine, per trasferire gli ultimi 160 record presenti sulla traccia N. 6, è necessario il tempo minimo di *seek* e la *rotational latency* come nei due casi precedenti, più il tempo di trasferimento dei record, che adesso corrisponde, come per i primi 160 record, alla metà del tempo di rotazione del disco:

0.6 + 3 + 3 = 6.6 millisecondi

Concludendo, il tempo necessario per leggere l'intero file è pari a:

11.2 + 9.6 + 9.6 + 6.6 = 37 millisecondi

Terzo problema. *Con riferimento ai dati relativi al problema precedente, valutare di nuovo il tempo necessario per leggere il file nell'ipotesi che i record fossero allocati su disco in modo casuale*.

Se i 960 record sono allocati su disco in modo casuale, il tempo necessario per trasferire ciascun settore corrisponde al tempo di *seek* (corrispondente per tutti allo stesso valore relativo al tempo medio di *seek*) più la *rotational latency,* più il tempo richiesto per il trasferimento di un singolo settore che, come indicato nella tabella 5.2 riportata nel testo, è di 0.019 millisecondi. Quindi il tempo necessario per trasferire un solo record è di:

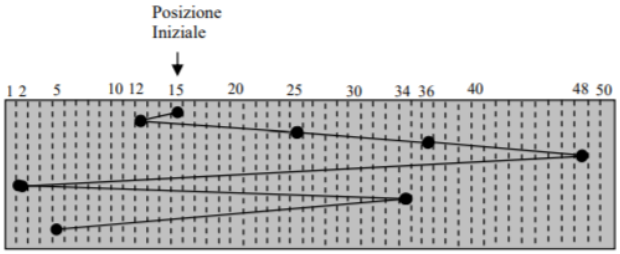
5.2 + 3 + 0.019 = 8.419 millisecondi

Per cui il tempo complessivo per trasferire l'intero file è di:

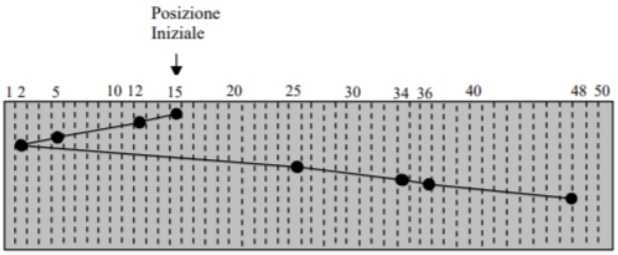
960*8.419 =8082.24 millisecondi

e cioè oltre 8 secondi.

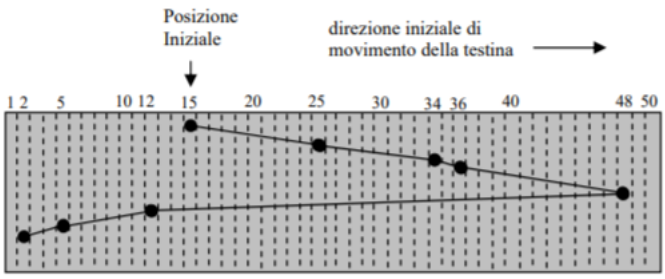
Quarto problema. *Supponiamo che mentre un processo sta operando sulla traccia numero 15 del disco, siano arrivate al gestore del disco richieste da parte di altri processi, per operare su altre tracce. In particolare, supponiamo che le richieste siano, nell'ordine di arrivo, quelle relative alle tracce: 12, 25, 36, 48, 2, 34, 5. Indicare il numero totale di cilindri su cui è necessario spostare la testina di lettura/scrittura per ciascuno dei seguenti algoritmi di scheduling delle richieste del disco: FCFS, SSTF, SCAN*.



$$\text{Tracce FCFS} = (15-12) + (25-12) + (36-25) + (48-36) + (48-2) + (34-2) + (34-5) = 146$$



$$\text{Tracce SSTF} = (15-12) + (12-5) + (5-2) + (25-2) + (34-25) + (36-34) + (48-36) = 59$$



$$\text{Tracce SCAN} = (25-15) + (34-25) + (36-34) + (48-36) + (48-12) + (12-5) + (5-2) = 79$$

9 Gestione della memoria secondaria (file system)

9.1 File system in generale

Fino ad ora abbiamo visto come si gestiscono le periferiche, i problemi di sincronizzazione tra processi interni ed esterni, l'accesso a risorse fisiche e logiche. Abbiamo anche detto che la memoria di massa (o disco) può essere considerata una periferica a tutti gli effetti. In un sistema operativo *general-purpose* una memoria di massa è irrinunciabile, e si basa su un'organizzazione logica in file. Come interagiamo con questi file?

9.1.1 Definizione e scopo del file system

Il file system consiste nell'insieme di meccanismi, politiche e tecniche di allocazione utilizzate in un sistema operativo per risolvere il problema dell'archiviazione delle informazioni in memoria secondaria: essa è persistente e ha grandi capacità, ma ha tempo di accesso maggiore rispetto a quello in memoria primaria. Lo svantaggio della lentezza è superato da tutti gli altri vantaggi, irrinunciabili in un calcolatore: un calcolatore è tale non solo se ha la CPU, la memoria principale e un certo numero di periferiche, ma deve avere anche una memoria persistente in grado di mantenere programmi e dati all'interno del sistema (altrimenti uno dovrebbe caricare sistema operativo e programmi ogni volta che avvia il calcolatore).

E la memoria ROM? La memoria ROM è il primo esempio che abbiamo visto (a Reti logiche) di memoria persistente. È mappata in un intervallo dello spazio di memoria del sistema, ma non ha le caratteristiche necessarie: è in sola lettura (oppure i tempi di scrittura sono più elevati rispetto a quelli di un disco) e la capacità è insufficiente (ospita al più ciò che serve per fare il bootstrap all'avvio).

9.1.2 Introduzione all'organizzazione del file system

Rappresentiamo la struttura del file system attraverso un'architettura a più strati. Tra i vari strati abbiamo interfacce, cioè insieme di funzioni implementate a livello inferiore ed utilizzate al livello superiore. Anche qua la stratificazione garantisce la modularità del sistema, quindi l'aggiornamento degli strati mantenendo le interfacce invariate.

- **Struttura logica**
  La struttura logica ha a che fare con l'organizzazione del file system: primitive che permettono l'interazione col file system stesso (creazione di file, navigazione del file system, linking per l'attribuzione di nomi diversi allo stesso file…), astrazioni tipiche del file system (file, directory, partizione).
- **Accesso**
  Protezione: come si definisce e come viene controllata runtime (verificare se l'azione richiesta può essere eseguita sul file indicato). Modalità di accesso al file: sequenziale, diretto e a indice. Strutture dati che rappresentano file e directory. A questo livello il file è visto come un array di record. Qual è la differenza tra record e blocco?
  - Il blocco è il blocco fisico, mentre
  - il record (detto anche record logico) è il blocco logico ed è molto più piccolo del blocco fisico.

  Quando eseguo una primitiva devo capire che tipo di accesso viene eseguito e agire di conseguenza. Definiamo la dimensione del record e i diritti che vogliamo esercitare sul file (si pensi a quelli definiti su UNIX).
- **Organizzazione fisica**
  Nel livello organizzazione fisica definiamo le tecniche di allocazione di un file all'interno del dispositivo virtuale. Due vincoli:
  - il modo in cui il file è rappresentato in questo livello, e
  - il modo in cui viene rappresentato il dispositivo virtuale (già detto prima, array di blocchi fisici).

  In questo livello si perde la distinzione tra file e directory.
- **Dispositivo virtuale**
  Il livello più basso ci rende indipendenti dallo specifico disco fisico: si parla di dispositivo virtuale. Non lo approfondiremo: **l'unica cosa da tenere a mente è che il dispositivo virtuale è niente meno che un vettore lineare di blocchi fisici** (insiemi di byte che costituiscono l'unità atomica di trasferimento di informazioni col dispositivo virtuale). Ogni blocco fisico, come in un qualunque array, risulta identificato da un indice.

9.1.3 Struttura logica del file system

Partiamo dal livello più alto, introducendo le astrazioni caratterizzanti il file system. In questo livello definiamo l'organizzazione del file system, in particolare le primitive utilizzabili dall'utente per interagire col file system.

9.1.3.1 Astrazione: il file

Il nome *file system* deriva dalla principale astrazione attuata dallo stesso: il file. Definiamo il file come unità logica di memorizzazione persistente. Con unità logica intendiamo che il contenuto di un file può essere interpretato in funzione della specifica applicazione che utilizzerà quei dati (il formato), in modo da dare a quei dati unitarietà dal punto di vista semantico: si pensi a Word con docx o Powerpoint con pptx. Il file può contenere anche programmi: programmi sorgente e programmi eseguibili.

Il file è un insieme di informazioni rappresentante attraverso un vettore di record logici (in UNIX il record logico ha dimensione di un byte). Un file ha bisogno di:

- **un nome** (ritorna la questione del naming), dato liberamente dal creatore, segue una sintassi che dipende dall'organizzazione del file system (sintassi che si modifica se introduciamo, ad esempio, un'organizzazione gerarchica con le directories – si concatenano gli elementi con lo slash);
- **un'estensione**, separata solitamente dal nome attraverso un punto, permette di capire quale interpretazione dare al contenuto del file;
- **un indirizzo**, puntatore alla memoria secondaria;
- **ulteriori attributi**, si pensi alla dimensione[12] (espressa in byte o in blocchi), il proprietario (utente o gruppo nel caso di UNIX, come visto a laboratorio), i bit di protezione (anch'essi già visti a laboratorio), la data e ora di creazione e/o modifica del file.

Definiremo anche il proprietario e i diritti di accesso al file.

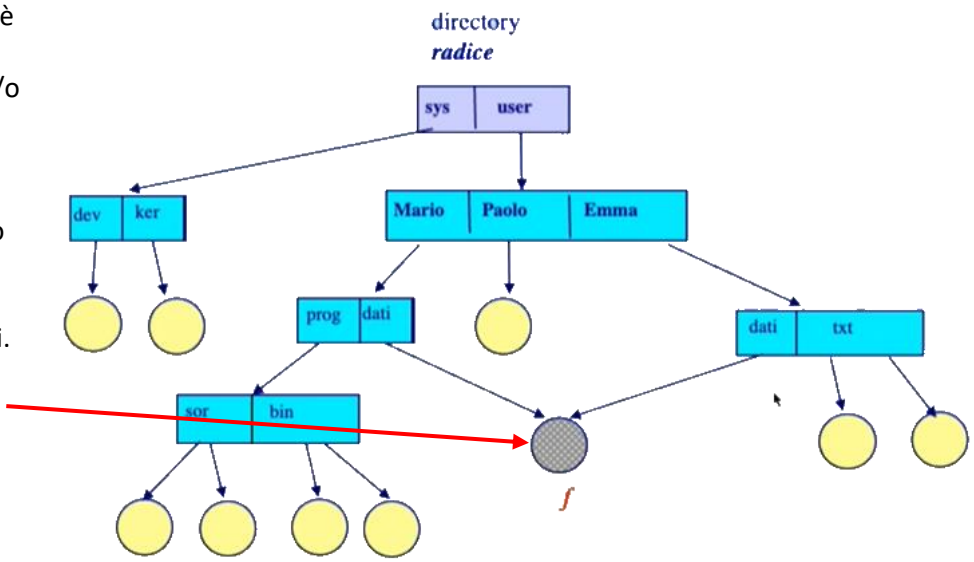
9.1.3.2 Astrazione: directory

La directory non è altro che un insieme di file e altre directory: *un file un po' particolare, il cui contenuto consiste in un elenco di file e/o directory dove esiste una relazione di contenimento* (cit.). Directory contenute in altre directory permettono di costruire una gerarchia padre-figlio, caratterizzata da una radice rappresentata dal file system. Nel caso di UNIX abbiamo la root, rappresentata dallo slash che è parte della sintassi: con lo slash introduciamo il pathname: attraverso una particolare sintassi descriviamo il percorso per raggiungere il file partendo dalla radice.

Si osservi che possiamo avere file aventi lo stesso nome ma path diverso.

9.1.3.3 Organizzazione di file e directory come grafo diretto e aciclico.

L'organizzazione di file e directory è di tipo gerarchico, visto che le directory sono contenitori di file e/o altre directory con rapporti di relazioni padre-figlio. Il file system può essere visto come un albero, anche se in realtà in UNIX abbiamo un grado diretto e aciclico. Questa cosa è dovuta al linking, con cui associamo a un file percorsi diversi.



Nell'esempio abbiamo, per il file f:

- `/user/Mario/dati/`
- `/user/Emma/dati/`

9.1.3.4 Astrazione: partizione

La partizione consiste in un disco logico mappato su un disco fisico (è un insieme di file), che coincide con tutto il dispositivo o una parte dello stesso. Al momento dell'installazione del sistema operativo UNIX si definiscono le partizioni, dischi logici mappati sullo stesso disco fisico. Solitamente abbiamo sempre due partizioni indipendenti: una usata come area di swap e un'altra che consiste nelle partizioni rimanenti, unificate a livello di file system (una radice unica).

[12] La dimensione incide pesantemente sull'efficienza (più è grande il file maggiore è il numero di settori che dovrò coinvolgere nelle mie operazioni sul disco)

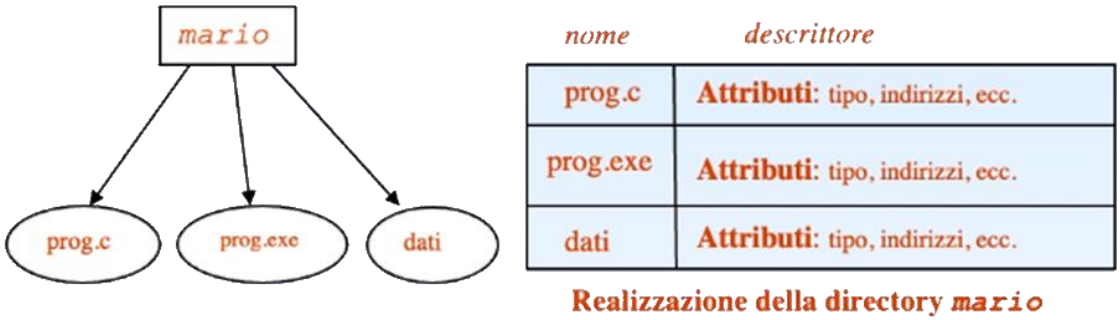
9.1.4 Livello di accesso del file system

In questo livello definiamo il modo in cui vengono rappresentanti file e directory nel sistema operativo (il file è visto come un array di blocchi logici - o record), le tipologie di accesso ai files e i controlli runtime di protezione.

9.1.4.1 Rappresentazione del file attraverso un descrittore e delle directory mediante tabelle

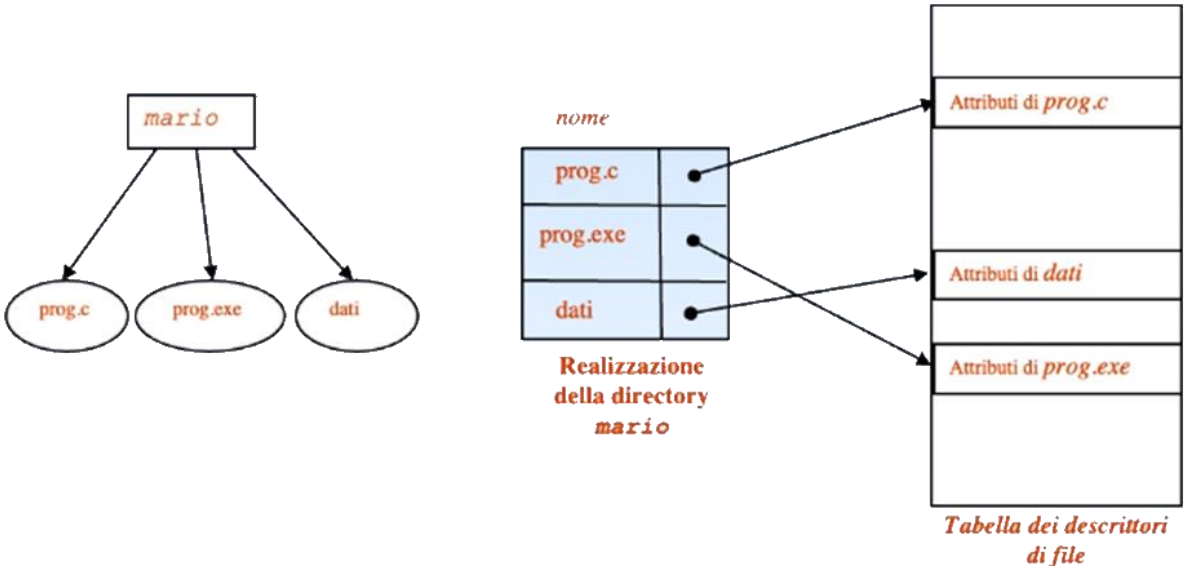
Ogni file è rappresentato da un descrittore di file, che memorizza gli attributi. Chiaramente questi descrittori devono persistere, quindi sono memorizzati in memoria secondaria. Per quanto riguarda le tabelle abbiamo due varianti possibili.

- **Windows (Approccio distribuito)**
  La directory è rappresentata attraverso un file che contiene una struttura dati di tipo tabellare. Abbiamo, per ogni file, due campi: uno contiene il nome simbolico del file e l'altro il descrittore del file.



  Nell'esempio in foto la directory `mario` contiene `prog.c`, `prog.exe` e `dati`. L'approccio è distribuito: la cosa potrebbe non essere efficiente nel caso di ricerche tra i vari descrittori. La cosa viene affrontata introducendo un meccanismo di caching.

- **Unix (Approccio centralizzato)**
  La directory è rappresentata attraverso un file dove ogni file viene rappresentato con due campi: un nome simbolico e un puntatore a un elemento della tabella *iList*, dove l'elemento puntato è il descrittore del file (detto iNode). La ricerca è più efficiente, poiché tutti i descrittori sono concentrati in un'area di memoria, tuttavia andiamo ad introdurre un livello di lettura in più.



  Supponiamo di voler accedere a un file a partire da un programma: per prima cosa utilizziamo la primitiva open (operazione preliminare con cui individuiamo strutture dati e descrittori coinvolti nell'accesso al file, e verifichiamo che ci siano i permessi per svolgere l'operazione), l'interfaccia ci permette di indicare il nome simbolico. Come uso il nome simbolico? Dobbiamo raggiungere il descrittore, per calcolare l'indice del descrittore (tabella) **devo per forza attraversare il grafo**, partendo dalla radice.

Se teniamo conto dell'organizzazione vista prima osserviamo che la tabella si compone di più parti: il puntatore non è per forza a descrittore di file, ma anche a descrittore di directory.

9.1.4.2 Operazioni di accesso ai record logici del file

Le operazioni di accesso sono:

- **Lettura** di informazioni contenute nel file (lettura di record logici, indipendentemente dal tipo del contenuto del file la dimensione del record logico è la stessa, dipende dal file system – nel caso di UNIX il record logico ha dimensione 1 byte[13]);
- **Scrittura**, immissione di nuove informazioni all'interno del file (si distingua la scrittura vera e propria, dove il contenuto del file prima dell'accesso viene perso, dell'*append*, dove i record logici vanno ad aggiungersi a quelli già esistenti).

Ogni operazione richiede un accesso al descrittore del file che, come già detto, è memorizzato sul disco. Fare ogni volta lettura dei descrittori dal disco è cosa piuttosto onerosa. Per tali motivi andiamo a creare una tabella dei file aperti, che contiene:

- copie dei descrittori dei file;
- I puntatori ai prossimi record logici da leggere/scrivere (si pensi all'accesso sequenziale);
- Informazioni sul processo che accede al file.

Questa tabella è posta in memoria centrale, riducendo così i tempi necessari per le letture. In alcuni sistemi operativi si fa anche un'operazione di *memory mapping*, cioè si copia tutto o parte del file in memoria centrale (lavorando lì fino all'operazione di chiusura). L'efficienza ci porta a distinguere due operazioni:

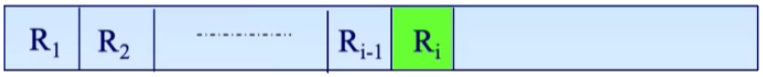
- **apertura**, introduzione di un nuovo elemento nella tabella dei file aperti ed eventuale *memory mapping* del file;
- **chiusura**, salvataggio dell'eventuale file memory mapped in memoria secondaria ed eliminazione dell'elemento corrispondente dalla tabella dei file aperti.

9.1.4.3 Metodologie di accesso

L'accesso a file può avvenire secondo varie modalità (che abbiamo già anticipato). Si osservi che le metodologie sono indipendenti dal dispositivo utilizzato e dalla tecnica di allocazione dei blocchi in memoria secondaria.

9.1.4.3.1 Accesso sequenziale

Il file è una sequenza di record logici $\{R_1, R_2, \dots, R_N\}$. Vogliamo accedere al record logico R_i: per fare ciò dobbiamo scorrere i precedenti $i - 1$ record.



L'approccio pare obsoleto e ha ragioni storiche (quando le memorie erano nastri magnetici l'accesso avveniva a livello fisico, per raggiungere una posizione devo per forza scorrerlo), ma è stato mantenuto poiché in alcuni casi ha un costo di accesso minore rispetto all'accesso diretto. Durante una sessione di accesso a un file il sistema registra la posizione del prossimo elemento della sequenza da leggere attraverso un ***puntatore a file*** (detto I/O pointer). Le primitive utilizzabili sono:

- `readnext(f, &V)`
  Lettura del prossimo record logico della sequenza.
  - Assegno il valore del prossimo record logico del file f alla variabile V
  - Posiziono il puntatore al file f sull'elemento successivo a quello letto.
- `writenext(f, V)`
  Scrittura del prossimo record logico della sequenza.
  - Scrive nel prossimo record logico del file f il valore della variabile V.
  - Posiziona il puntatore al file f sull'elemento successivo a quello scritto.

9.1.4.3.2 Accesso diretto

Il file è un insieme non ordinato di record logici $\{R_1, R_2, \dots, R_N\}$. Accediamo al record logico R_i in modo diretto dopo aver calcolato il relativo indice i. Le primitive utilizzabili sono:

- `readd(f, i, &V)`
  Lettura del record logico R_i dal file f, il valore letto viene assegnato alla variabile V
- `writed(f, i, V)`
  Scrittura del valore della variabile V nel record logico R_i del file f

I parametri sono gli stessi delle primitive dell'accesso sequenziale, ma si ha anche l'indice i. L'indice è parametro esplicito della primitiva che esegue l'operazione su file e non viene mantenuto dal sistema operativo (se ne occupa il programma). Per avere un'idea si guardi la differenza di codice tra accesso sequenziale e accesso diretto:

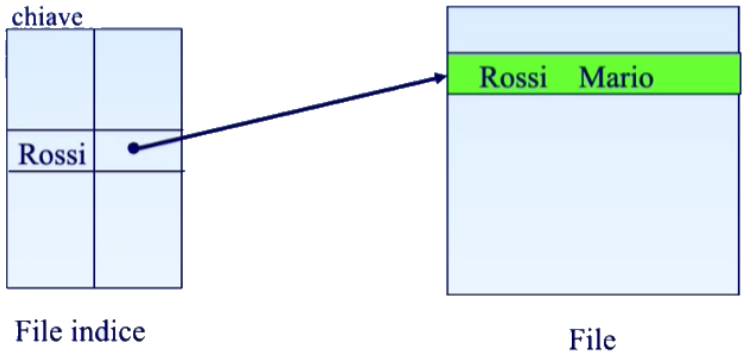
[13] Array di byte

```
Accesso sequenziale
Prima leggo i record logici precedenti e poi il record che ci interessa
for(i = 1; i < k; i++)
    readnext(f, &V);
readnext(f, &V);

Accesso diretto
Indico direttamente il record logico a cui voglio accedere
readd(f, k, &V);
```

9.1.4.3.3 Accesso a indice

Ogni record logico è strutturato in almeno due campi, uno dei quali contiene la chiave che identifica il record stesso. A ogni file si associa anche una struttura tabellare detta indice (una sorta di tabella hash) dove si stabiliscono tutte le corrispondenze tra chiavi e record (si pongono nella tabella i vari riferimenti ai record).



Chiaramente conviene avere il file indice in memoria primaria. Le primitive sono:

- `readk(f, key, &V)`
  Lettura del record logico con chiave uguale a *key* dal file f, il valore letto viene assegnato alla variabile V.
  Nell'esempio qua sopra facciamo una lettura scrivendo `readk(f, "Rossi", &V)`.
- `writek(f, key, V)`
  Scrittura del valore della variabile V nel record logico del file f avente chiave *key*.

9.1.5 Livello dell'organizzazione fisica del file system

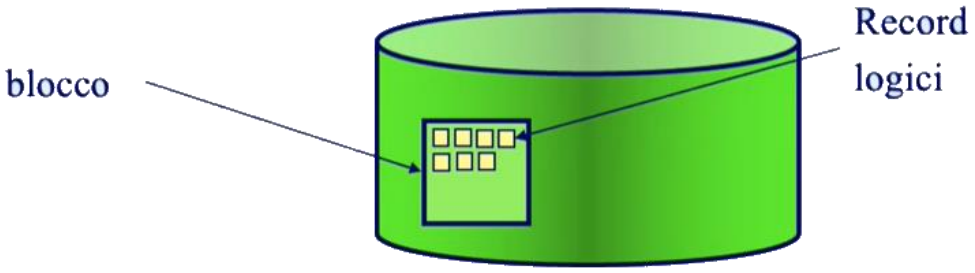
Nel livello organizzazione fisica definiamo tutte le regole principali che permettono di allocare un file nel dispositivo virtuale. In questo livello si perde la distinzione tra file e directory.

9.1.5.1 Premessa introduttiva

Il descrittore del file esiste ed abita nella memoria secondaria. Dobbiamo caricarlo in memoria principale per ottimizzare le operazioni di accesso (cosa che si fa di solito nell'operazione di apertura del file). Noi non abbiamo alcuna idea di come sia il disco (potrebbe essere addirittura un nastro), non abbiamo consapevolezza di come questo array di record logici sia implementato nel disco stesso. **Compito del file system è stabilire una corrispondenza tra record logici e blocchi (fisici)**: ricordiamoci che solitamente

Dimensione(blocco) >> Dimensione(Record logico)

- Ogni dispositivo di memoria secondaria è visto come un array di blocchi (detti anche record fisici).
- Il blocco corrisponde all'unità di trasferimento in tutte le operazioni di IN/OUT verso il dispositivo che implementa la memoria di massa. È chiaro che la dimensione del blocco va stabilita e non è sempre la stessa, dipende da come è fatto il disco.



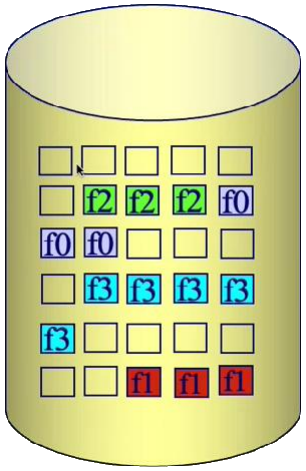
Se il record fisico ha dimensione maggiore rispetto ai record logici allora un blocco può contenere più record logici.

9.1.5.2 Metodi di allocazione dei file

Supponiamo che per comodità ogni blocco contenga un insieme di record logici contigui. Quali sono le tecniche più comuni per la scelta dei blocchi in cui allocare i record di un file?

9.1.5.2.1 Allocazione contigua

I blocchi che contengono i record dello stesso file sono contigui: il problema di allocazione contigua sarà trovare, nel vettore dei blocchi, un numero di blocchi contigui sufficiente per poter contenere tutto un file.



Ogni file è mappato su un insieme di blocchi fisicamente contigui. Immaginiamo il dispositivo virtuale attraverso il cilindro (dove il vettore è rappresentato per comodità come una matrice).

Cosa facciamo?

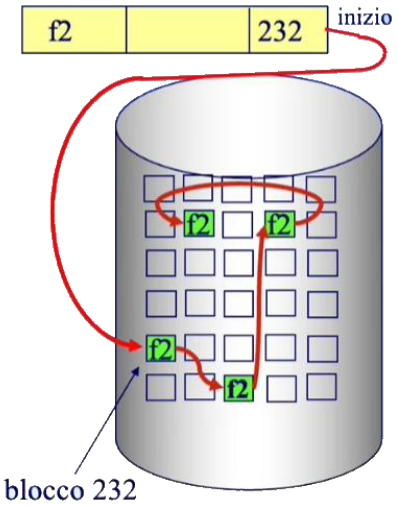
- Calcolo quanti blocchi mi servono.
- Ricerco nel disco una partizione di blocchi abbastanza grande per ospitare il mio file.

La tecnica è praticamente la fotocopia della tecnica a partizione variabili vista per la memoria centrale.

9.1.5.2.2 Allocazione a lista concatenata

L'organizzazione a lista è abbastanza equivalente alla paginazione: i blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata. Ciò permette di superare la contiguità dei blocchi che implementano un file.

- L'algoritmo di allocazione cambia: da un lato non devo più scorrere una lista di partizioni libere (al più ho una lista di blocchi liberi, che devo mantenere ordinata), dall'altro dovrò riservare spazio (in ogni blocco) per il puntatore al blocco successivo (l'indice del blocco per essere precisi).
- Nel descrittore troviamo l'indice del primo dei blocchi dove è allocato il file.



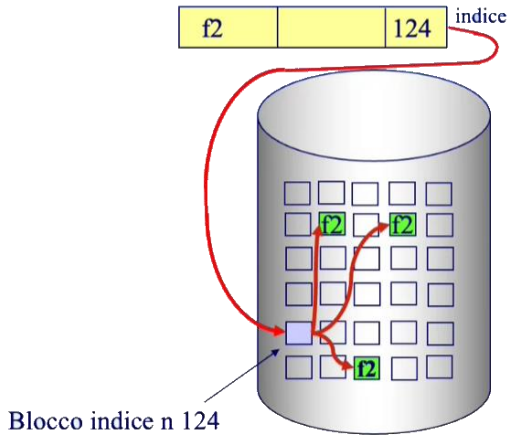
Esercizio estemporaneo: quanto spazio ci serve? Supponiamo di avere un disco di 1Tb (2^{40} byte), e blocchi di 4Kb (2^{12} byte per blocco). Per ottenere il numero di blocchi facciamo una divisione

$$\frac{2^{40}}{2^{12}} = 2^{28} \text{ blocchi}$$

Il numero 28 ci indica anche il numero di bit necessari per rappresentare l'indice del blocco. Segue che in ogni blocco dobbiamo riservare 28 bit per memorizzare l'indice del blocco successivo.

9.1.5.2.3 Allocazione a indice

Ogni file ha associato un blocco indice, il cui contenuto sarà una serie di puntatori a tutti i blocchi che mi servono per allocare il file.



Dire *tutti* è un po' rischioso. L'indice contiene un numero finito di indici a blocchi (il blocco ha una dimensione, non posso suppore un numero potenzialmente infinito di indici).

Esercizio estemporaneo.

Supponiamo di avere un disco da 1Tb, e blocchi grandi 1Kb (2^{10} byte per blocco). Gli indici dei blocchi saranno in numero

$$\frac{2^{40}}{2^{10}} = 2^{30} \text{ blocchi}$$

Mi servono 30 bit per rappresentare l'indice di ogni blocco. Quanti indici entrano nel blocco indice? Divido la dimensione del blocco per la dimensione dell'indice. Poniamo 32 visto che 30 non è multiplo di potenza di due (ci semplifichiamo la vita).

$$\frac{8 \cdot 2^{10}}{32} = \frac{2^3 \cdot 2^{10}}{32} = \frac{2^{13}}{32} = 2^9 \text{ indici di blocchi}$$

Posso avere al più 512 blocchi. Moltiplicando il numero per la dimensione dei blocchi ottengo la dimensione massima di un file. Deduciamo un problema di scalabilità.

9.1.5.3 Confronto

<table>
<tr><th>Allocazione contigua</th><th>Allocazione a lista concatenata</th><th>Allocazione a indice</th></tr>
<tr><th colspan="3">Frammentazione esterna</th></tr>
<tr><td>Presente[14]</td><td colspan="2">Assente</td></tr>
<tr><th>Vantaggi</th><th>Vantaggi</th><th>Vantaggi</th></tr>
<tr><td rowspan="2"><u>Costo di ricerca di un blocco basso</u>.
L’allocazione contigua semplifica drasticamente il calcolo dell’indirizzo del blocco fisico dove ho memorizzato l’i-esimo record.
$$\frac{B+i}{Nb} \Longrightarrow |B+i|_{Nb}$$
B è l’indice del primo blocco dove è memorizzato il file (e ciò è presente nel descrittore del file), Nb è il numero di record logici memorizzati all’interno del blocco fisico (costante ottenute da altre due costanti, dimensione del blocco fisico e dimensione del blocco logico). i dipende dal metodo di accesso: nel caso di accesso sequenziale abbiamo l’I/O pointer che gestisce; nell’accesso diretto siamo noi a indicare i, attraverso il secondo parametro della primitiva; nell’accesso a indice otteniamo i dalla tabella hash, dopo aver indicato la chiave.

<u>Indipendenza del costo dalla tipologia di accesso</u>.
Da questi discorsi deduciamo l’indipendenza a livello di costo dalla tipologia di accesso.</td><td><u>Minor costo di allocazione (assente frammentazione esterna)</u>.
La non contiguità riduce drasticamente il costo di allocazione: non devo cercare partizioni contigue, ma solo il numero di blocchi necessari (che non devono essere necessariamente contigue).

<u>Accesso sequenziale conveniente.</u>
L’accesso sequenziale è efficiente: le informazioni relative all’indice del blocco, calcolate e usate all’accesso precedente, possono essere utilizzate per l’accesso successivo.</td><td><u>Basso costo di accesso sequenziale e diretto</u>.
Leggo l’indice al primo accesso (nel blocco indice), faccio caching (in un colpo solo) in modo da poter recuperare al volo le informazioni degli altri indici. La velocità è maggiore. Il metodo è vantaggioso sia con accesso sequenziale che con accesso diretto.</td></tr>
<tr><td rowspan="2">Svantaggi
<u>Struttura dati per gestire le partizioni libere</u>.
Abbiamo la frammentaz. esterna. Maggior costo (conseguenza) per la ricerca di spazio libero: devo mantenere una struttura dati che mi indica le partizioni libere (con indirizzo e dimensione delle partizioni). Il dilemma è il solito: best-fit, first-fit, o worst-fit (ordinamento delle partizioni libere per dimensione decrescente).</td><td>Svantaggi
<u>Accesso diretto oneroso</u>
L’accesso diretto è oneroso: non possiamo calcolare in modo diretto il blocco fisico in cui si trova il record, logico, dobbiamo per forza compiere $\frac{i}{Nb}$ accessi prima di raggiungere il blocco fisico desiderato.

Possibilità di errore se un link viene danneggiato (perdo il concatenamento, perdo il contenuto del file). La cosa è molto seria: mi basta un solo link per perdere una porzione (anche grande) del file.

Risolvo con una lista doppiamente concatenata: mi serve ulteriore spazio nel blocco per costruire un’ulteriore lista concatenata, ma nel senso opposto (a quel punto per perdere porzioni significative del file devo avere per forza due guasti). Altra tecnica risolutiva, la migliore, è la FAT (File Allocation Table), una tabella di allocazione del file non più distribuita ma concentrata. Per ogni blocco si riporta l’informazione relativa al blocco successivo. Nell’ultimo blocco che implementa il file indichiamo -1. La FAT può essere anche copiata in memoria principale (lo facciamo quando apriamo il file)!!!</td><td>Svantaggi
<u>Non è scalabile</u>.
Problema visto nell’esercizio estemporaneo: all’aumentare della dimensione del file non abbiamo un blocco indice che aumenta di dimensione, il numero di indici contenuti in un blocco indice è finito.</td></tr>
</table>

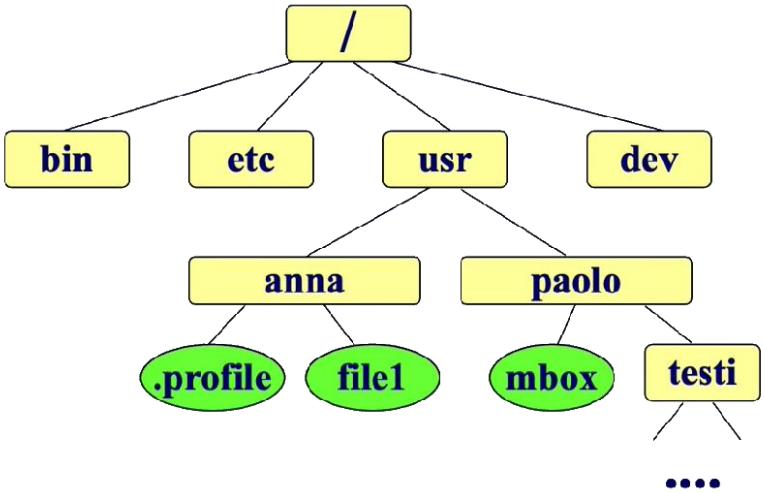
Abbiamo adottato come criterio di valutazione il numero di accessi che dobbiamo fare per individuare il blocco fisico. In molti casi si adottano ibridi: allocazione contigua in caso di file piccoli, allocazione a indice nel caso di file grandi.

[14] La soluzione è la compattazione, ma è un’operazione onerosa – molto di più rispetto a quella della memoria principale. Si pensi che nella compattazione della memoria centrale ricorriamo all’hard disk come supporto, ma noi adesso vogliamo compattare proprio l’hard disk!

9.2 File system in UNIX e Linux

9.2.1 Struttura logica

Il file system in UNIX è un grafo aciclico (grafo e non solo albero per il meccanismo di linking).



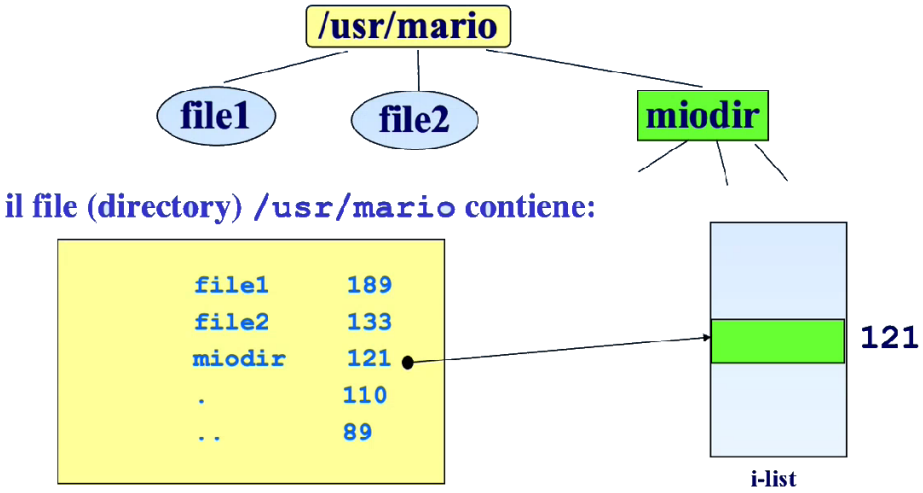
Una caratteristica di UNIX è l'omogeneità: ogni risorsa è vista come un file. Abbiamo tre categorie di file:

- file ordinari;
- directory;
- file speciali che descrivono i dispositivi.

Se io ho primitive che aprono, leggono/scrivono e chiudono un file ordinario può essere una buona idea usare le stesse primitive (stessa semantica e interfaccia) per aprire, leggere/scrivere e chiudere un file relativo a un dispositivo: l'unica differenza è che la lettura relativa al dispositivo può avere un *side effect*, in contrasto a letture di file ordinari.

9.2.1.1 Directory in UNIX

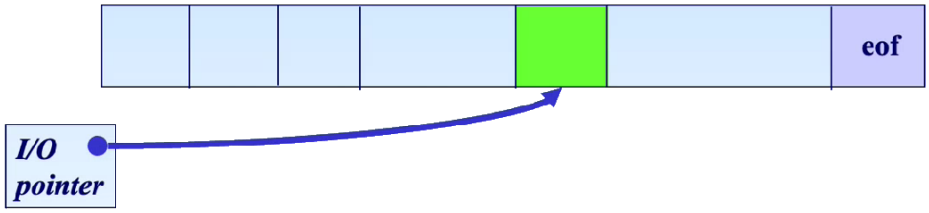
La directory in UNIX è una tabella dove ogni elemento è caratterizzato da due campi: il nome simbolico del file e l'iNumber, cioè l'identificativo che ci permette di raggiungere il relativo iNode all'interno della iList.



Quando lanciamo la primitiva open per aprire un file cosa fa il sistema? Deve percorrere vari descrittori per arrivare al descrittore del file desiderato.

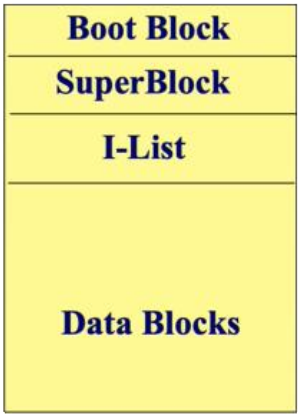
9.2.2 Livello dell'accesso

In UNIX l'accesso è sequenziale. Il file è una sequenza di record, in UNIX possiamo parlare di sequenza di byte visto che il record coincide col byte. Il sistema operativo mantiene un puntatore detto I/O pointer che indica la posizione corrente del record nel file aperto. Ogni accesso è subordinato all'operazione di apertura.



9.2.3 Organizzazione fisica

In UNIX si adotta un tipo di allocazione ibrida (allocazione ad indici a più livelli). I blocchi fisici hanno dimensione tra i 512-4096 byte. La superficie del disco virtuale è divisa in quattro partizioni:



- **Boot block**
  Informazioni fondamentali per il bootstrap (appendice sul disco delle informazioni o programmi che devono essere eseguiti al momento del bootstrap, inizializzazione del sistema)
- **SuperBlock**
  Informazioni sull'organizzazione del file system (in particolare i limiti delle quattro regioni che stiamo descrivendo, il puntatore alla lista dei blocchi liberi e il puntatore a una lista degli iNode liberi)
- **iList**
  Tabella contenente tutti gli iNode di file, directory e dispositivi. Ogni file ha associato uno o più nomi simbolici, uno **e un solo** descrittore detto iNode (raggiungibile a partire da un intero detto iNumber, che è l'indice dell'elemento posto nell'array iList[15]).
- **Data Blocks**
  Partizione contenente i blocchi utilizzati per allocare file.

9.2.3.1 Descrittore del file: iNode

iNode è il descrittore del file. Contiene un po' di cose:

- tipo di file (ordinario / directory / file speciale)
- proprietario (utente e gruppo, user-id e group-id)
- i 12 bit di protezione (i 9 bit dei permessi, SUID, SGID e STIcky bit)
- data di creazione e data di modifica
- dimensione
- numero di links
- vettore di indirizzamento (costituito da 13 a 15 indirizzi di blocchi, consente l'indirizzamento dei blocchi di dati sui quali è allocato il file)
- …

9.2.3.1.1 Vettore di indirizzamento

A cosa ci serve un vettore di indirizzamento che ha da 13 a 15 blocchi? Con indirizzamento in UNIX intendiamo come viene interpretato e utilizzato questo vettore.

Supponiamo che il blocco sia di 512 byte. Supponiamo che gli indirizzi dei blocchi siano su 32 bit (4 byte). Ogni blocco è quindi raggiungibile da un indice che sta su 4 byte: possiamo calcolare quanti indici/indirizzi di blocco possono essere contenuti in un blocco (nel blocco indice).

$$\frac{512}{4} = 128 \text{ indirizzi di blocco}$$

- I primi 10 indirizzi permettono di fare accesso diretto al blocco. Abbiamo 40 byte dedicati a contenere gli indirizzi di dieci blocchi diversi. Tutti i file che hanno dimensione inferiore a $10 \cdot 512$ byte hanno in sostanza un costo nullo di accesso (indirizzo già posto in memoria).
- L'undicesimo indirizzo che sta nel descrittore è l'indice di un blocco indice. Un blocco indice contiene fino a 128 indirizzi. Necessario un accesso per ottenere gli indirizzi dei 128 blocchi (*indirezione singola*). Otteniamo l'accesso a
  $$128 \cdot 512 \text{ byte} = 65536 \text{ byte} = 64KB$$
- Il dodicesimo indirizzo e il tredicesimo sono uguali ai precedenti, ma permettono di fare rispettivamente *indirezione doppia* e *indirezione tripla* (due e tre accessi). Otteniamo:
  $$128 \cdot 128 \cdot 512 \text{ byte} = 8MB$$
  $$128 \cdot 128 \cdot 128 \cdot 512 \text{ byte} = 1GB$$
  Questo "trucco" ha un'overhead spaziale (cit.), ma ci permette di superare il problema della scalabilità.

Complessivamente otteniamo come dimensione:

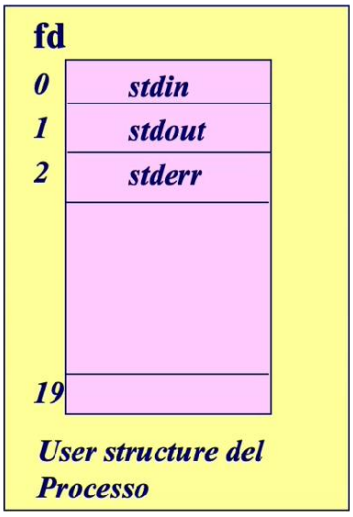
$$1GB + 8MB + 64KB + 5KB = 2^{30} + 2^{23} + 2^{16} + 5 \cdot 2^{10}$$

[15] Attenzione alla scalabilità, anche qua il numero massimo di file memorizzati dipende dalla dimensione della iList.

9.2.4 Strutture dati del Kernel per l'accesso a file: tabella dei file aperti del processo e del sistema

Ogni accesso è subordinato a un'operazione di apertura. La questione principale è gestire l'*iNode* di un file aperto: sono tanti, visto l'attività del sistema multiprogrammato. Dobbiamo considerare anche l'apertura di un file da parte di più processi: UNIX ne tiene conto e memorizza l'*iNode* una volta soltanto.

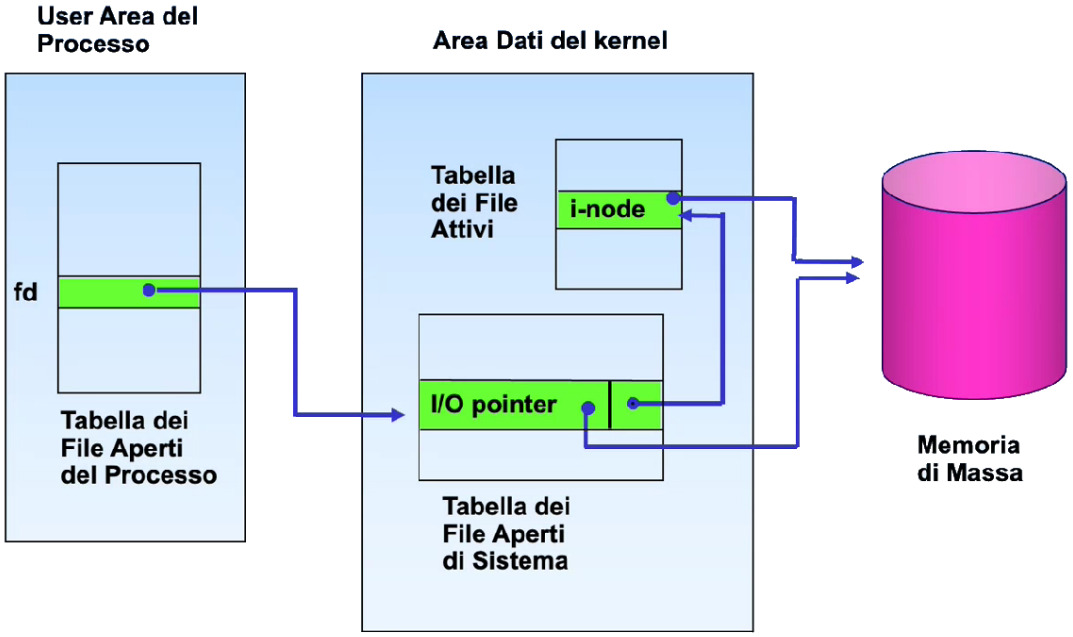
A ogni processo è associata una **tabella dei file aperti**, dove ogni elemento è un *file descriptor*. La tabella è raggiungibile dalla user structure di un processo.



Al momento dell'inizializzazione del processo la tabella contiene tre file DESCRIPTOR aperti "automaticamente": standard IN (0), standard OUT (1) e standard ERR (2). I rimanenti vengono usati in apertura: la primitiva open restituirà il *file descriptor*. Nell'area dati del kernel individuiamo anche altre due tabelle:

- la **tabella dei file attivi** (equivalente della cache, per ogni file aperto ho una copia del suo iNode);
- la **tabella dei file aperti di sistema**, ha un elemento per ogni operazione di apertura (con I/O pointer, posizione all'interno del file e puntatore all'iNode del file nella tabella dei file attivi). Possibile avere più elementi relativi allo stesso file.

Rendiamo chiare le idee attraverso la seguente figura:



<u>Cosa succede quando si lancia una open?</u>

- Si raggiunge attraverso più step il descrittore dell'iNode, che andiamo a copiare nella tabella dei file attivi (copia da memoria seconda a memoria centrale, se già presente non c'è bisogno di copiarlo nuovamente)
- Costruiamo il collegamento tra tabella dei file attivi e tabella dei file aperti di sistema: creo una nuova entry in questa tabella e ci copio l'indice dell'iNode posto nella tabella dei file attivi. L'I/O pointer sarà posto all'inizio o in fondo (dipende dall'eventuale presente in mode di O_APPEND).
- Costruisco il collegamento tra tabella dei file aperti del processo e tabella dei file aperti di sistema: creo l'elemento (dove poniamo anche il nome simbolico) e associamo. La funzione open restituirà un intero o un tipo che rappresenta il file descriptor (e le operazioni successive sul file verranno riferite con fd).

9.2.5 System Call per l'accesso a file

9.2.5.1 Primitiva `open` per l'apertura di un file

```
int open(char nomefile[], int flag[, int mode]);
```

La primitiva restituisce un intero e prende in ingresso:

- un vettore di caratteri contenente il nome simbolico del file (incluso il path, relativo o assoluto);
- un flag con cui specifico la modalità di accesso (Esempi di valori: O_RDONLY, O_WRONLY…), si consideri che la primitiva verificherà che il soggetto (!!!!!) possa porre effettivamente il flag indicato;
- un parametro opzionale che utilizziamo solo quando l'opzione di apertura determina la creazione del file (O_CREAT), per indicare i bit di protezione.

Nel caso in cui il parametro non venga posto quando necessario si utilizzerà un valore di default detto *user mask*. Il valore restituito è l'identificativo numerico del file descriptor: nel caso in cui l'apertura del file non abbia successo (per esempio nel caso in cui non abbiamo i permessi indicati) si restituisce -1.

9.2.5.2 Primitiva `close` per la chiusura di file

```
int close(int fd);
```

L'interfaccia è molto più semplice. L'unico parametro di ingresso richiesto è l'identificativo del file descriptor. Memorizza in memoria persistente il file, quando questo ha avuto delle modifiche. Il valore restituito è un intero che:

- è uguale a zero nel caso in cui l'operazione di chiusura abbia successo;
- è uguale a un valore minore di zero in caso di insuccesso.

La cosa è fondamentale, soprattutto per rimuovere dalle strutture dati precedentemente introdotto ciò che a noi non serve più. Verifica se l'iNode posto nella tabella dei file attivi dovrà essere rimosso o meno (verifico il numero di istante aperte dello stesso file, abbiamo un contatore memorizzato nel descrittore).

9.2.5.3 Primitive per lettura e scrittura di file [Ripasso]

Dopo l'apertura di un file l'accesso a questo avviene esclusivamente attraverso il *file descriptor* fd. Le operazioni di accesso sono sempre sequenziali: l'I/O pointer inizialmente punta al primo record, per ogni operazione di lettura/scrittura il sistema operativo andrà a incrementare il puntatore (**IL PROGRAMMA NON GESTISCE IL POINTER**).

- L'operazione è atomica: è eseguita come se fosse un'unica istruzione Assembler.
- Le operazioni devono essere sincrone: il processo che esegue read o write deve essere sospeso in attesa del completamento dell'operazione. Facciamo ciò ricorrendo, ovviamente, alle primitive wait e signal.

9.2.5.3.1 Primitiva `read` per la lettura

```
int read(int fd, char *buf, int n);
```

I parametri richiesti sono:

- il file descriptor del file;
- l'area in cui trasferire i byte letti (puntatore all'area);
- il numero di caratteri da leggere

In caso di successo (lettura di almeno un byte richiesto) viene restituito un intero positivo che rappresenta il numero di caratteri effettivamente letti. Il valore è minore di n nel caso in cui si sia manifestato un errore da gestire a livello applicativo o quando il file contiene un numero di byte minore rispetto al numero indicato in n.

Dal punto di vista del programmatore i vincoli dovuti all'accesso sequenziale sono "cammuffati": non sono costretto a leggere un byte alla volta, non mi accorgo della limitazione grazie al livello di astrazione.

9.2.5.3.2 Primitiva `write` per la scrittura

```
int write(int fd, char *buf, int n);
```

I parametri richiesti sono:

- il file descriptor del file;
- l'area in cui trasferire da cui trasferire i byte scritti (puntatore all'area);
- il numero di caratteri da scrivere

In caso di successo (scrittura di almeno un byte richiesto) viene restituito un intero positivo che rappresenta il numero di caratteri effettivamente scritti.

9.2.6 AGGIUNTA [29/11/2022]: gestione delle pagine libere in Linux

Si invita alla lettura di questa sezione sul libro: mi è stato fatto notare che Avvenuti ha chiesto diverse volte questa sezione e che diversi hanno avuto difficoltà visto che la cosa non è trattata in questa dispensa. Mi scuso fortemente per la mancanza e ne approfitto per invitarmi nuovamente a segnalarmi eventuali mancanze.

8.4 La gestione della memoria nel sistema Unix

Le più recenti versioni di Unix (dalla versione 3 del BSD in avanti) forniscono l'astrazione di memoria virtuale (vedi Paragrafo 4.2.1), sia per consentire a ogni processo di indirizzare un'area di memoria di dimensioni superiori a quelle della memoria fisica, sia per svincolare il grado di multiprogrammazione dai limiti della memoria effettivamente disponibile.

Il sistema di gestione della memoria in Unix si basa su paginazione e segmentazione: in particolare viene adottato il modello a segmentazione paginata (vedi Paragrafo 4.3.4). Infatti, lo spazio di indirizzamento di ogni processo è segmentato. Esso, cioè, viene partizionato in tre segmenti distinti: codice (*code segment*), dati (*data segment*) e stack (*stack segment*). L'allocazione dei segmenti viene gestita con la tecnica della paginazione a domanda (vedi Capitolo 4) effettuata dal kernel, con il supporto del processo di sistema *pagedaemon* (individuato dal valore di pid 2), che periodicamente (ogni 250 ms) si occupa della sostituzione delle pagine.

Il collegamento tra pagine logiche e pagine fisiche viene mantenuto all'interno di una tabella delle pagine. Inoltre, il kernel mantiene una descrizione dello stato di allocazione della memoria all'interno della tabella delle pagine fisiche (la *core map*), nella quale ogni elemento rappresenta una pagina fisica, e contiene le informazioni relative a essa. Per esempio, ogni elemento della *core map*, oltre a contenere l'indirizzo della pagina fisica che rappresenta, indica se essa è libera oppure allocata; se allocata, registra inoltre le informazioni relative alla pagina logica in essa allocata e al processo a cui appartiene.

La sostituzione delle pagine si basa sull'algoritmo di seconda chance (vedi Paragrafo 4.3.3). Il processo *pagedaemon* esegue l'algoritmo di sostituzione delle pagine soltanto se il numero delle pagine fisiche libere è inferiore a un valore di soglia prefissato, rappresentato dalla costante di sistema *lotsfree*. Può darsi, tuttavia, che nonostante l'intervento del pagedaemon, la frequenza di paginazione sia troppo elevata, e il numero di pagine libere rimanga comunque inferiore a *lotsfree*: in questo caso interviene lo *swapper* (vedi Capitolo 4) che, in modo più drastico, provvede al trasferimento di uno o più processi dalla memoria centrale a quella secondaria. Per rilevare questa situazione il sistema definisce altre due costanti di sistema: *minfree* (che esprime il numero minimo di pagine fisiche libere necessarie per evitare lo swap-out dei processi) e *desfree* (che esprime il numero medio *desiderabile* di pagine fisiche libere). La relazione tra le tre costanti di sistema è la seguente:

$$lotsfree > desfree > minfree$$

Lo *swapper*, quindi, interviene se sono soddisfatte le seguenti condizioni:

- il numero di pagine fisiche libere è minore di *minfree*;
- numero medio di pagine fisiche libere nell'unità di tempo è minore di *desfree*.

In questo modo si riesce a contenere l'uso della CPU da parte del *pagedaemon*.

10 Protezione e sicurezza

10.1 Introduzione

Cosa intendiamo con protezione e sicurezza?

- La **protezione** riguarda l'insieme di attività che si preoccupano di garantire all'interno di un sistema di calcolo il controllo dell'accesso alle risorse logiche e fisica da parte degli utenti.
- La **sicurezza** garantisce l'autenticazione degli utenti impedendo accessi non autorizzati al sistema (quando indichiamo delle credenziali), tentativi dolosi di alterazione e distruzione dei dati.

Approfondiremo soprattutto la protezione.

10.2 Protezione: Modelli, politiche e meccanismi

Abbiamo tre livelli concettuali: modelli, politiche e meccanismi.

10.2.1 Modello di protezione

Il modello di protezione si basa sui tre ingredienti evidenziati in grasso: soggetti, oggetti e diritti di accesso.

- Un modello di protezione definisce **soggetti** e **oggetti**.
- I soggetti (parte attiva) chiedono di accedere agli oggetti (parte passiva), o di poter controllare altri soggetti.
- Il processo di per sé non coincide col soggetto (lo abbiamo già detto): lo stesso processo può, durante l'esecuzione, alterare il livello di privilegio. Possiamo immaginarlo come una coppia

  $$< Processo, Dominio\ di\ protezione >$$

  Il dominio di protezione è l'ambiente di protezione nel quale il soggetto sta eseguendo, in un certo senso l'insieme dei **diritti di accesso** posseduti dal processo. Si osservi che:
  - ogni soggetto ha un unico dominio di protezione;
  - un processo può cambiare dominio durante la sua esecuzione.

10.2.1.1 Dominio di protezione sull'oggetto

Nel caso dell'oggetto è opportuno definire l'insieme dei diritti di accesso (un dominio)

$$< \text{Nome oggetto}, \text{Dominio di protezione} >$$

L'insieme è associato all'oggetto e non al soggetto: consiste in una lista di elementi che identifica i soggetti del sistema e i diritti che questi hanno sull'oggetto.

10.2.1.2 Associazione tra processo e dominio

Ulteriori osservazioni sul dominio.

- L'associazione permette di capire su quali oggetti il processo può compiere operazioni. Se nel dominio non è presente un oggetto allora il processo non può eseguire nessuna operazione su quell'oggetto.
- L'associazione tra processo e dominio può essere statica o dinamica.
  - In presenza di ***associazione statica*** l'insieme delle risorse disponibili per un processo rimane fisso per tutto il suo tempo di vita. Sistema rigido, svantaggi:
    - non è pienamente applicabile il principio del privilegio minimo;
    - l'insieme delle risorse che un processo potrà usare deve essere un'informazione disponibile prima dell'esecuzione del processo, cosa non semplice;
  - In presenza di ***associazione dinamica*** il dominio può cambiare dinamicamente. Cambiare il dominio significa cambiare il soggetto!
    - **Esempi**:
      - sostituzione del PSW con un nuovo contenuto che altera il dominio;
      - funzione exec in UNIX (cambio di codice può implicare cambio di dominio in presenza di SUID e/o SGID).
    - Il cambio del solo dominio e non del processo ha costo decisamente minore: non facciamo cambio di contesto, in parole povere minore overhead (si pensi al non dover "rinfrescare" le memorie cache).

Principio del privilegio minimo. È importante attribuire ai soggetti i diritti di accesso solo agli oggetti strettamente necessari per la loro esecuzione.

10.2.2 Politiche di protezione (DAC, MAC e RBAC, principio del privilegio minimo)

Le politiche di protezione sono un insieme di regole con il quale andiamo ad attribuire diritti e modalità di accesso relativamente agli oggetti. Le politiche possibili sono sostanzialmente tre:

- ***Discretional Access Control*** (DAC, Politica adottata in UNIX)
  Il creatore di un oggetto controlla i diritti di accesso per quell'oggetto: diritti che ho io in quanto

proprietario, diritti che hanno gli utenti appartenenti al gruppo proprietario, diritti che hanno tutti gli altri utenti. Problema: ogni utente può decidere come fare come vuole, in un sistema dove la sicurezza deve essere assicurata agli estremi non può adottare questa politica.

- ***Mandatory Access Control*** (MAC)
  I diritti di accesso sono gestiti in modo centralizzato (si risolve il problema detto poco fa). Le regole sono stabilite da un'autorità centrale e non sono derogabili.
- ***Roled-Based Access Control*** (RBAC)
  Compromesso tra le due politiche precedenti. Si stabiliscono i diritti centralmente, ma non lo si fa rispetto agli utenti: si fa rispetto ai ruoli (si pensi a quando si nomina un amministratore in un sito, non è tanto dare dei permessi a un utente ma inserire l'utente nel gruppo degli amministratori). Controllo centralizzato, con possibilità di associare ruoli diversi a un utente (oppure creare un ruolo particolare per uno specifico utente).

10.2.3 Meccanismi di protezione

I meccanismi di protezione sono strumenti messi a disposizione dal sistema di protezione per imporre una determinata politica.

- Ricordiamoci che la politica stabilisce cosa va fatto ed il meccanismo il modo in cui la cosa viene fatta (nulla di nuovo).
- Flessibilità: i meccanismi di protezione devono essere generali, in modo da garantire flessibilità nella scelta delle politiche di protezione (anche questo nulla di nuovo).

10.3 Esempio: il modello a matrice degli accessi

La matrice degli accessi si basa su quanto introdotta fino ad ora, in particolare i concetti di soggetti e oggetti. I soggetti sono entità che esercitano diritti di accesso su oggetti o altri soggetti. Il modello garantisce:

- la rappresentazione dello stato di protezione;
- il rispetto dei vincoli di accesso (le informazioni devono essere accessibili dal meccanismo di protezione ogni volta che un soggetto richiede l'accesso a un oggetto);
- la modifica **controllata** dello stato di protezione runtime (associazione dinamica, si determina una transizione di stato).

10.3.1 Rappresentazione dello stato di protezione

Nel modello a matrice ci immaginiamo una matrice dove le righe sono i soggetti e le colonne gli oggetti. Data una matrice A e un soggetto S affermiamo che il seguente elemento

$$A[S, X]$$

Definisce i diritti di accesso che il soggetto S ha per l'oggetto X.

	X_1	X_2	X_3	S_1	S_2	S_3
S_1	read*	read	execute		terminate	receive
S_2		owner write		control receive		terminate
S_3	write execute		read	send	send receive	

10.3.2 Rispetto dei vincoli di accesso

I passi legati al meccanismo adottato sono i seguenti:

- un soggetto S tenta di eseguire un'operazione α su un oggetto X;
- viene generata la tripla (S, α, X) e passata al meccanismo di protezione;
- il meccanismo di protezione interroga la matrice degli accessi. Se $\alpha \in A[S, X]$ allora l'accesso è valido, diversamente si ha una violazione dello stato di protezione.

Il meccanismo deve prevedere la possibilità di espandere la matrice, sia verticalmente (nuovi soggetti) che orizzontalmente (nuovi oggetti e/o soggetti).

10.3.3 Modifica dello stato di protezione

Graham e Denning hanno definito un insieme di comandi che permette la modifica controllata della matrice degli accessi. Questi comandi sono dei seguenti tipi

- aggiunta e rimozione di diritti di accesso per oggetti/soggetti;
- propagazione dei diritti di accesso.

10.3.3.1 Aggiunta di diritti di accesso per oggetti/soggetti

Un primo comando riguarda l'assegnazione di un diritto di accesso per un oggetto Xa un soggetto S_j da parte del soggetto S_i. Quest'ultimo soggetto può compiere l'operazione solo se il diritto `owner` appartiene ad $A[S_i, X]$.

Un soggetto che ha diritto `owner` su un particolare oggetto può alterare l'intera colonna relativa all'oggetto stesso.

10.3.3.2 Rimozione di diritti di accesso per gli oggetti/soggetti

Un secondo comando riguarda la possibilità di eliminare da un soggetto S_j un diritto di accesso per un oggetto X da parte del soggetto S_i. Quest'ultimo soggetto può compiere l'operazione solo se:

- il diritto `control` appartiene ad $A[S_i, S_j]$, e
- il diritto `owner` appartiene ad $A[S_i, X]$

10.3.3.3 Propagazione dei diritti di accesso

Propagare diritti di accesso significa copiare diritti di accesso per un oggetto da un dominio ad un altro della matrice. La cosa viene rappresentata attraverso un *copy flag*: un asterisco.

Affermiamo che: un soggetto S_i può trasferire un diritto α per un oggetto X a un altro soggetto S_j solo se S_i possiede il diritto α per X e tale diritto possiede il copy flag.

- A questo punto la propagazione avviene copiando il solo diritto (propagazione limitata di un diritto) oppure il diritto accompagnato da copy flag. Nel secondo caso il soggetto che riceve il diritto può a sua volta trasmetterlo.
- La propagazione, in aggiunta, può comportare la perdita del diritto per il soggetto che lo ha trasmesso (trasferimento di un diritto).

10.3.4 Perché ci fasciamo il capo sulla modifica controllata della matrice?

Le operazioni che un soggetto compie sugli oggetti sono controllate grazie alla matrice degli accessi, ma avere un accesso controllato anche sulla matrice degli accessi è un po' un loop: dovrei ricorrere a un qualche altro meccanismo di protezione. Graham e Denning, in realtà, hanno dimostrato che con i comandi precedenti si ha:

- propagazione limitata e controllata (*confinement*)
- è evitata la modifica indiscriminata dei diritti di accesso (*sharing parameters*)

10.4 Realizzazione della matrice degli accessi

10.4.1 Premessa: problemi e necessità di un'implementazione efficiente

Attenzione ai seguenti problemi…

- **Dimensione della matrice (numero righe e numero colonne dinamici)**.
  La dimensione della matrice può essere considerevole. La questione principale è il numero di colonne.
- **Matrice sparsa**.
  La matrice può essere sparsa ("una matrice con dei buchi", cit.)
- **Ridondanze**.
  Il contenuto della matrice è ridondante: se un'operazione può essere eseguita su tutti gli oggetti, allora l'operazione deve essere contenuta in tutti i domini.

Le cose dette devono essere affrontate nell'implementazione della matrice.

10.4.2 Primo metodo di implementazione: *Access Control List* (ACL, implementata in UNIX)

Si considerano le colonne della matrice degli accessi: ad ogni oggetto è associata una lista che contiene

- tutti i soggetti che possono accedere all'oggetto, e
- per ogni soggetto, i diritti di accesso che questi ha per l'oggetto.

In sostanza ogni colonna è implementata come una lista: si risolve il problema della matrice sparsa, ho un'**implementazione distribuita** (all'interno della memoria di massa) della matrice stessa (metto solo gli elementi relativi ai soggetti che hanno almeno un diritto di accesso rispetto all'oggetto).

ACL è implementata in UNIX: **i bit di protezione in UNIX sono le ACL**, e sono distribuite nelle iNode!

10.4.2.1 Sistema monoutente (descrizione basilare)

Supponiamo che un soggetto S_i voglia compiere un'operazione α su un oggetto X_j.

- La prima cosa che dobbiamo fare è individuare l'ACL corrispondente. La ACL di un oggetto è formata dall'insieme delle coppie
$$< \text{Soggetto,Insieme dei diritti} >$$
Ovviamente si considerano solo le coppie che detengono almeno un diritto di accesso sull'oggetto.
- Trovata la ACL scorriamo la lista finché non individuiamo una coppia
$$< S_i, R_k >$$
Dove $\alpha \in R_k$. Se quest'ultima condizione è valida si ha match e l'operazione è consentita.
- **Ottimizzazione**. È lecito affermare che scorrere tutte le volte queste liste non sia una cosa molto efficiente. Per ottimizzare poniamo una lista di default che contiene tutti i diritti di accesso che, per la loro generalità, sono applicabili a tutti gli oggetti.
- Facciamo prima il controllo nella lista di default e solo dopo nella ACL del relativo oggetto: se entrambe le ricerche hanno esito negativo l'operazione è abortita.

10.4.2.2 Implementazione basata su ruoli

La ACL nasce in sistemi per singoli utenti. Introduciamo il concetto di Group ID (e il concetto di ruolo). Un elemento presente nella ACL presenta la forma
$$\text{UID, GID: } < \text{ Insieme di diritti} >$$
dove $< \text{UID, GID}>$ è l'identificativo di un soggetto. In alcuni casi potrebbe essere più comodo assegnare a un soggetto differenti diritti di accesso a seconda del gruppo di appartenenza, cioè in base al ruolo che ricopre in quel preciso istante. Si osservi che
$$\text{UID1, GID1: } < \text{ Insieme di diritti 1} > \neq \text{ UID1, GID2: } < \text{ Insieme di diritti 2} >$$
Questo tipo di rappresentazione apre la porta a una serie di possibilità:

- Dire che un utente può accedere a certi oggetti indipendentemente dal gruppo a cui appartiene
$$\text{UID, *: } < \text{ Insieme di diritti} >$$
- Bloccare selettivamente uno specifico utente
$$\text{UID, *: } < \textit{Insieme vuoto} >$$
- Definire diritti validi per tutt
$$\text{*, *: } < \text{ Insieme di diritti} >$$

Risulta abbastanza chiaro da quanto detto che è difficile determinare in modo immediato i diritti complessivi di un soggetto: si dovrebbe scorrere tutte le ACL.

10.4.3 Secondo metodo di implementazione: *Capability List* (CL)

Si considerano le righe della matrice degli accessi: ad ogni soggetto è associata una lista che contiene

- tutti gli oggetti accessibili dal soggetto, e
- per ogni oggetto, i relativi diritti di accesso del soggetto.

La strategia è simile alla precedente: anche qua dobbiamo implementare delle liste. Nella ACL lo facciamo in modo distribuito su ogni file, qua lo facciamo in modo distribuito su ogni processo. Il numero di liste è sicuramente minore al numero di liste che si avrebbero con ACL.

Ogni elemento della lista prende il nome di **Capability**: esso si compone di un elemento che identifica l'oggetto e una sequenza di bit che rappresenta i diritti che il soggetto a cui fa riferimento la lista possiede sull'oggetto. Supponiamo che un soggetto S_i desideri eseguire un'operazione α su un oggetto X_j: quello che dobbiamo fare è

- prendere la Capability List relativa al soggetto S_i e
- scorrerla fino a quando non individueremo una capability relativa all'oggetto X_j, che concede l'esecuzione dell'operazione α.

Contrariamente alle ACL è immediato conoscere i diritti complessivi di un soggetto.

10.4.3.1 Protezione

La tabella della Capability List non può essere direttamente accessibile dal programmatore, precisamente deve essere in sola lettura. Le modifiche sono possibili solo con il lancio di primitive di sistema.

10.4.3.2 Rappresentazione grafica della CL

Dato un soggetto abbiamo un insieme di entry che indicano diritti sia sui file che sui processi.

	F_1	F_2	F_3	F_4	F_5	F_6	PR_1	PR_2
soggetto	-	-	R	RWE	RW	-	W	-

Alternativamente possiamo rappresentare la CL nel seguente modo: una tabella dove indichiamo, per ogni oggetto:

- Il tipo del file;
- I diritti che il soggetto ha sull'oggetto o sul soggetto;
- Il puntatore all'oggetto.

capability list per il soggetto

Tipo	Diritti	Oggetto
File	R	Puntatore a F_3
File	RWE	Puntatore a F_4
File	RW	Puntatore a F_5
Printer	W	Puntatore a stampante

10.4.4 Implementazione in UNIX

In realtà UNIX, pur basandosi sulle ACL, implementa una soluzione ibrida che comprende sia ACL che CL.

- Quando la open verifica che l'operazione può andare a buon fine si usa la ACL, cioè si usano i nove bit di protezione nel caso di UNIX. Il processo è eseguito in nome di un particolare soggetto (UID, GID), le informazioni indicate nel secondo parametro di open vengono usati per verificare la validità nell'accesso.
- Il file descriptor restituito dalla open è una nuova entry posta nella tabella dei file aperti di processo. A questo punto possiamo eseguire operazioni su file, per esempio lettura e scrittura: anche in questo caso dobbiamo fare i controlli. Quando un soggetto chiede di eseguire una read o una write indicando un particolare file descriptor il sistema controllerà che a fd corrisponda una entry che ammette tale operazione. Questo controllo viene fatto solo sulla tabella dei file aperti di processo, non sulla iNode: segue che abbiamo proprio qua l'implementazione della Capability List.
- Le ACL sono persistenti, permanenti, relativamente statiche. Le CL sono dinamiche: non esistono CL senza processi, al momento della creazione del processo si inizializza la CL con le capability relative allo STDIN, STDOUT, STDERR.

10.4.5 Assegnazione/Revoca dei diritti di accesso

In un sistema di protezione dinamica può essere necessario assegnare/revocare i diritti di accesso per un oggetto. La revoca può essere:

- selettiva o generale, cioè valere per tutti gli utenti che hanno quel diritto di accesso o solo per un gruppo;
- parziale o totale, cioè riguardare un sottoinsieme di diritti per l'oggetto o tutti;
- temporanea o permanente, cioè il diritto di accesso non sarà più disponibile (sarà disponibile), oppure può essere successivamente riottenuto (può essere successivamente revocato).

Il modo in cui noi revochiamo diritti di accesso varia in base all'implementazione adottata: ACL o CL.

- **Revoca in presenza di ACL (Liste di accesso)**
  La revoca è semplice, si prende la ACL associata all'oggetto per il quale vogliamo revocare (o aggiungere diritti), si scorre la lista modificando gli elementi relativi ai soggetti di cui vogliamo modificare i diritti. La cosa è ancora più efficiente in UNIX grazie ai bit di protezione.
- **Revoca in presenza di CL (Capability List)**.
  La revoca è più complessa in una lista di Capability. Dobbiamo verificare per ogni soggetto se contiene la capability con riferimento all'oggetto considerato.

10.5 Problema della sicurezza multilivello

Esistono dei sistemi dove sia i soggetti che gli oggetti possono appartenere a delle classi di segretezza (dove il livello di segretezza varia, si pensi all'ambito militare dove esistono varie tipologie di classificazione per i documenti[16]).

Obiettivo: garantire in un sistema dove esistono oggetti con livelli di segretezza diversi l'estensione dei livelli stessi ai soggetti, e definire delle regole che autorizzino i soggetti negli accessi a seconda del livello di sicurezza del soggetto stesso e dell'oggetto coinvolto.

10.5.1 Modello Bell-La Padula

Il modello Bell-La Padula ha origine negli ambienti militari, ma è applicabile in tutti i sistemi dove si vuole regolare il flusso delle informazioni. Associamo livelli di segretezza a oggetti e soggetti. Supponiamo di averne quattro:

- Non classificato
- Confidenziale
- Segreto
- Top secret

Il livello di segretezza associato al soggetto indica a quali documenti il soggetto ha diritto di accesso. Se un utente ha un livello di segretezza non classificato e il documento è classificato top secret allora l'utente non potrà accedere, mentre un utente che ha livello di segretezza top secret può accedere al documento. Quest'ultimo può accedere anche a tutti i file che hanno livello di segretezza minore rispetto a quello che l'utente stesso ha associato (Segreto, Confidenziale, Non classificato). Al momento della creazione del file si definisce il livello di segretezza, stessa cosa al momento di creazione dell'utente, Formalizziamo quanto detto…

10.5.1.1 Regole formali

Lettura verso il basso, scrittura verso l'alto!

Immaginiamo una funzione SC, che dato come parametro un utente o un oggetto restituisce il suo livello di sicurezza.

- **Proprietà di semplice sicurezza**. Un soggetto con un livello di sicurezza k può leggere solo oggetti al suo livello o a livelli inferiori, cioè
  $$\mathrm{SC}(S) \geq \mathrm{SC}(O)$$
  Attenzione, si parla solo di operazioni di lettura (NON DI SCRITTURA).
- **Proprietà STAR (*)**, meno intuitiva. Un soggetto in esecuzione al livello di sicurezza k può scrivere solamente oggetti al suo livello o a quelli superiori (poco intuitivo, cit.), cioè
  $$\mathrm{SC}(S) \leq \mathrm{SC}(O)$$
  Se io sono a livello top secret non posso modificare documenti a livello non classificato (posso solo leggerli)!

Bell-La Padula non risolve il problema dell'integrità delle informazioni (paradossalmente posso scrivere dove non posso leggere), ma garantisce la sicurezza. La proprietà STAR * impedisce che io legga l'informazione a livello alto per poi scriverla a livello più basso: se io permettessi ciò potrei rendere disponibili informazioni con livello di segretezza elevato a chi in realtà non dovrebbe leggere quelle informazioni.

10.5.2 Modello Biba (regole formali)

Lettura verso l'alto, scrittura verso il basso!

Il modello Biba, in contrapposizione al modello di Bell-La Padula, garantisce l'integrità ma non la sicurezza. Analizziamo le regole formali ("regole scambiate" rispetto a Bell-La Padula). Immaginiamo una funzione SC, che dato come parametro un utente o un oggetto restituisce il suo livello di sicurezza.

- **Proprietà di semplice sicurezza**. Un soggetto con un livello di sicurezza k può scrivere solamente oggetti al suo livello o a livelli inferiori (questa volta nessuna scrittura verso l'alto)
  $$\mathrm{SC}(S) \leq \mathrm{SC}(O)$$
- **Proprietà di integrità STAR (*)**. Un soggetto in esecuzione al livello di sicurezza k può leggere solamente oggetti al suo livello o a quelli superiori (questa volta niente letture verso il basso)
  $$\mathrm{SC}(S) \geq \mathrm{SC}(O)$$

Le regole del modello Biba sono palesemente in conflitto con le regole di Bell-La Padula: non possono essere attuate in contemporanea! Anche Biba può essere sovrapposto alla matrice degli accessi come Bell-La Padula.

[16] Top secret, livello di segretezza più elevata. Pubblico, livello di segretezza minore.

10.5.3 Esempio di applicazione della matrice degli accessi con cavallo di Troia

Il cavallo di troia è un programma, installato nel sistema, che sarà sicuramente eseguibile. Definiamo lo stato di protezione utilizzando la matrice degli accessi

	O1 (Segreto)	O2 (Pubblico)	O3 (Pubblico)
S1	Read, Write	Execute	Write
S2		Execute, Owner	Read, Write, Owner

Supponiamo che esista un soggetto S2 che ha installato un oggetto O2, che contiene il cosiddetto cavallo di Troia. S2 è owner di O2, e che in quanto tale abbia dato diritti di esecuzione a tutti i soggetti, in particolare ad S1! Abbiamo anche un oggetto O3 di cui S2 è proprietario (e ha diritti di lettura e scrittura): egli ha assegnato, in quanto proprietario, il diritto di scrittura a S1! Lo stato di protezione è compatibile con la matrice degli accessi. Qual è il fatto?

- S1 a un certo punto esegue O2, può farlo perché il proprietario ha dato diritto di esecuzione ad S1.
- Quel codice farà verosimilmente cose ritenute utili, a un certo punto quel codice chiede di leggere O1: può farlo! Il soggetto S1 ha permessi di lettura e scritture su S1.
- Supponiamo che a questo punto ci sia una scrittura su O3: possiamo farla perché il proprietario ha dato i permessi per fare ciò ad S1. Palesemente O2 copierà in O3 il contenuto di O1.

Conclusione: utilizzare solo la matrice degli accessi non impedisce i cavalli di Troia!

Vediamo cosa succede applicando le regole di Bell-La Padula.

- Per prima cosa supponiamo due livelli di segretezza: Segreto e Pubblico.
    - O1 è segreto mentre O2 e O3 sono pubblici.
    - S1 è segreto, S2 è pubblico.
- Applichiamo le regole di Bell-La Padula.
    - S1 è a livello segreto. Chiede di leggere O2, e può farlo perché è a livello pubblico.
    - Nell'esecuzione di O2 abbiamo alla lettura di O1. La cosa è consentita: S1 è segreto, quindi l'operazione su O1 è concessa.
    - Infine abbiamo la scrittura su O3: interviene la proprietà STAR di Bell-La Padula, e la scrittura viene impedita!

La sovrapposizione di Bell-La Padula e della matrice degli accessi (un AND tra le due cose) permette di gestire lo stato di protezione impedendo cavalli di Troia.