**CHE COS’E’ UN SISTEMA OPERATIVO?** *25-09-24*

Un **sistema operativo** è un insieme di programmi di un calcolatore che ha il compito di agire come intermediario tra i programmi applicativi e i componenti fisici del calcolatore stesso, allo scopo   
di facilitarne l’uso e garantire che tale uso sia effettuato in maniera efficace ed efficiente.

Per avere le idee più chiare, possiamo rappresentare un calcolatore come un insieme di tre   
macro-componenti, organizzate gerarchicamente in tre livelli:

Immagine che contiene testo, schermata, diagramma, Parallelo

Descrizione generata automaticamente

Partendo dal basso, abbiamo:

* **Livello hardware:** è composto dai componenti fisici del calcolatore, e comprende la CPU,  
  la memoria principale, e le periferiche come la tastiera, il mouse, le memorie di massa…   
  Questi componenti mettono a disposizione dei programmi le risorse necessarie per   
  la loro esecuzione, e per questo motivo sono detti anche *risorse fisiche*;
* **Livello del sistema operativo:** è composto da un insieme di programmi che hanno il compito di gestire le risorse fisiche della macchina. Essendo le risorse fisiche molto eterogenee tra loro, per accedere a tali risorse in maniera standardizzata (e quindi indipendentemente da come sono realizzate) è necessario che il livello inferiore offra un’interfaccia,   
  detta *interfaccia hardware*, che sono le istruzioni Assembly della CPU;
* **Livello delle applicazioni:** è composto dai programmi applicativi. Questi programmi   
  non interagiscono con le risorse fisiche in maniera diretta (sia perché, come già detto,   
  le risorse fisiche sono molto eterogenee tra loro, ma anche perché dell’utente non ci si può fidare, e quindi non gli si può dare il controllo diretto di tali risorse), ma lo fanno tramite un’interfaccia offerta dal livello del sistema operativo, composta da una serie di funzioni,   
  dette primitive (o system call), che possono essere invocate dai programmi applicativi   
  e permettono di agire sulle risorse fisiche del calcolatore in maniera controllata.

Quello che fa il sistema operativo è dunque generare l’astrazione di una macchina più semplice da usare, e più sicura.

**GESTIONE DEI PROCESSI** *08-10-24*

Analizziamo innanzitutto la differenza tra programma e processo:

* Un **programma** è qualcosa di statico: è una sequenza di istruzioni scritte in un certo linguaggio di programmazione.
* Un **processo** è qualcosa di dinamico: è un programma in esecuzione con dati in ingresso   
  ben precisi, e cioè la sequenza di eventi generati dal calcolatore durante l’esecuzione   
  del programma con quei dati in ingresso ben precisi.

Ogni processo ha associate delle risorse, tra cui della memoria, tipicamente organizzata   
in questo modo:

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

**STATI DI UN PROCESSO**

Durante la sua “vita”, un processo passa per vari stati:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Non appena viene creato un processo, questo si trova nello stato *nuovo*. Il sistema operativo successivamente attiva il processo, e cioè crea la memoria del processo e tutte le strutture dati   
ad esso associate (vedremo un esempio più avanti), e inserisce il processo nella coda *pronti*, contenente tutti i processi che, se avessero a disposizione la CPU, potrebbero andare in esecuzione. Non vanno in esecuzione perché la CPU è una sola, e può eseguire un solo processo per volta.

Quando arriverà il momento, il sistema operativo sceglierà un processo tra quelli che si trovano   
nella coda pronti e lo manderà in esecuzione, togliendolo da tale coda e assegnandogli la CPU.

Ad un processo che si trova in esecuzione possono succedere tre cose:

* Può succedere che termini. Il sistema operativo allora distrugge la memoria del processo   
  e tutte le strutture dati ad esso associate, e il processo passa così allo stato *terminato*.   
  Al suo posto, il sistema operativo manda in esecuzione un altro processo tra quelli presenti in coda pronti.
* Può succedere che si sospenda in attesa che si verifichi un qualche evento (*es.* che un dispositivo sia pronto per un’operazione di I/O). Il sistema operativo allora inserisce il processo in una coda *bloccati*, contenente tutti i processi che sono in attesa del verificarsi   
  di un certo evento, e al suo posto manda in esecuzione un altro processo tra quelli presenti   
  in coda pronti. Quando poi si sarà verificato l’evento atteso, il sistema operativo risveglierà il processo sospeso, spostandolo dalla coda bloccati alla coda pronti, così che questo   
  possa tornare in esecuzione prima o poi;
* Può succedere che il sistema operativo lo interrompa forzatamente (si vedrà successivamente con gli *algoritmi di short term scheduling preemptive*). Il sistema operativo allora inserisce   
  il processo interrotto nella coda pronti e al suo posto manda in esecuzione un altro processo tra quelli presenti nella coda pronti.

**DESCRITTORE DI UN PROCESSO**

Ad ogni processo è associata una struttura dati, detta **descrittore del processo**, costituita da:

* L’**identificativo** del processo;
* **Informazioni che sono usate dal sistema operativo per decidere quale processo   
  mandare in esecuzione tra quelli presenti nella coda pronti**;
* Lo **stato** del processo;
* **Riferimento alla memoria del processo;**
* **L’identificativo del processo successivo nella coda in cui il processo considerato si trova;**
* Una **copia del** ***contesto* del processo**, e cioè dei valori assunti dai registri della CPU   
  durante l’ultima esecuzione del processo;
* **Informazioni sulle risorse utilizzate dal processo (*es.* dispositivi di I/O, file…)**.

**CAMBIO DI CONTESTO**

Il cambio del processo in esecuzione con un altro richiede un insieme di operazioni che vanno   
sotto il nome di **cambio di contesto:**

1. **Salvataggio del contesto del processo in esecuzione:** i valori correnti dei registri della CPU vengono salvati nel descrittore del processo in esecuzione;
2. **Inserimento del descrittore in una coda bloccati o nella coda pronti:**a seconda   
   se il processo si è sospeso volontariamente in attesa del verificarsi di un qualche evento   
   o se è stato interrotto forzatamente dal sistema operativo, il suo descrittore viene inserito nell’opportuna coda;
3. **Caricamento del contesto del processo da mandare in esecuzione:** la copia del contesto presente nel descrittore del processo da mandare in esecuzione viene caricata   
   nei registri della CPU. Tra i valori caricati c’è anche quello del PC (il registro contenente la prossima istruzione da eseguire), per cui, a seguito di questa operazione,   
   il processo da mandare in esecuzione andrà effettivamente in esecuzione.

**CREAZIONE E TERMINAZIONE DEI PROCESSI**

Un processo può creare un altro processo. Chiameremo:

* **Processo padre** il processo creante;
* **Processo figlio** il processo creato.

Ogni processo è sempre figlio di un altro processo, e può avere a sua volta processi figli.   
Si avrà così una gerarchia di processi, con in cima un processo radice, figlio di nessuno,   
da cui tutto parte. Le informazioni sulle relazioni parentali di un processo sono mantenute   
nel suo descrittore.

**INTERAZIONE TRA PROCESSI**

Due processi si dicono concorrenti se le loro esecuzioni si sovrappongono nel tempo.   
Questa definizione, tuttavia, è un po’ ambigua: potrebbe far pensare che i due processi debbano andare in esecuzione in parallelo per poter essere definiti concorrenti, cosa che non è possibile, visto che   
la CPU è una sola e può eseguire un solo processo per volta. Una definizione più chiara è allora   
la seguente: due processi si dicono **concorrenti** se la prima operazione di uno comincia prima che termini l’ultima dell’altro.

Immagine che contiene linea, schermata, Rettangolo, Parallelo

Descrizione generata automaticamente

In base alla nuova definizione, P1 e P2 sono due processi concorrenti.

Due processi concorrenti possono essere tra loro:

* **Interagenti:** l’esecuzione di uno è influenzata da quella dell’altro, e viceversa;
* **Indipendenti:** l’esecuzione di uno non è influenzata da quella dell’altro, e viceversa.

L’interazione tra due processi può essere di due tipi:

* **Competizione:** i due processi competono nell’uso di risorse comuni che non possono essere utilizzate contemporaneamente.
* **Cooperazione:** i due processi cooperano nell’eseguire un’attività comune.

Per un corretto uso del calcolatore, in tutti i tipi di interazione è necessaria dunque la **sincronizzazione**, e cioè dei vincoli nell’esecuzione delle operazioni dei processi interagenti:

* Nel caso della competizione, occorre garantire che un solo processo alla volta abbia accesso alla risorsa comune. Si parla in questo caso di **mutua esclusione**, o anche   
  di **sincronizzazione *indiretta***;
* Nel caso della cooperazione, occorre garantire che le operazioni compiute dai processi avvengano in un preciso ordine. Si parla in questo caso di **comunicazione,** o anche   
  di **sincronizzazione *diretta***.

**SCHEDULING**

Esistono tre diverse attività di scheduling:

1. **Short term scheduling:** il sistema operativo sceglie, tra i processi nella coda pronti,   
   quello da mandare in esecuzione;
2. **Medium term scheduling:** il sistema operativo sceglie per quali processi fare *swap-out*,   
   e cioè spostare dalla memoria RAM alla memoria secondaria;
3. **Long term scheduling:** il sistema operativo sceglie per quali processi fare *swap-in*,   
   e cioè spostare dalla memoria secondaria alla memoria RAM.

**SHORT TERM SCHEDULING**

Lo short term scheduling avviene attraverso certi algoritmi di short term scheduling.   
Questi si dividono in due categorie:

* **Non preemptive**: invocano il cambio di contesto solo quando il processo in esecuzione   
  è terminato o si è sospeso in attesa del verificarsi di un certo evento;
* **Preemptive**: invocano il cambio di contesto anche in altri casi, interrompendo   
  così forzatamente il processo in esecuzione.

Un’altra classificazione degli algoritmi di short term scheduling è la seguente:

* **Senza priorità**: i processi in coda pronti vengono mandati in esecuzione   
  nell’ordine in cui arrivano;
* **Con priorità**: i processi in coda pronti vengono mandati in esecuzione in ordine di priorità decrescente.

Usando un algoritmo di short term scheduling senza priorità, la coda pronti è ordinata   
nello stesso ordine in cui arrivano i processi. In questo modo, per inserire un nuovo processo   
in coda pronti basta fare un inserimento in fondo, e per scegliere qual è il prossimo processo   
da mandare in esecuzione basta fare un’estrazione dalla testa.

Usando un algoritmo di short term scheduling con priorità, la coda pronti è ordinata per priorità decrescente dei processi. In questo modo, per scegliere il prossimo processo   
da mandare in esecuzione basta fare un’estrazione dalla testa, ma per inserire   
un nuovo processo in coda pronti bisogna alla peggio scansionare tutta la coda   
per metterlo nella posizione compatibile con la sua priorità. Verrà introdotto dunque dell’overhead.

Chiamiamo:

* **CPU-burst di un processo** il tempo per cui il processo, una volta finito nella coda pronti   
  per la prima volta o dopo essersi sospeso, dovrà tenere occupata la CPU (e quindi, dovrà andare   
  in esecuzione) prima di sospendersi o terminare;
* **IO-burst di un processo** il tempo per cui il processo, una volta sospeso,  
  dovrà eseguire operazioni di I/O (e quindi, dovrà rimanere sospeso in attesa che   
  il dispositivo interessato nelle operazioni di I/O sia pronto) prima di tornare nella coda pronti.

Useremo le seguenti metriche per valutare le prestazioni degli algoritmi di short term scheduling:

* **Tempo di risposta di un processo :** è il tempo che passa da quando il processoarriva per la prima volta nella coda pronti a quando termina il suo primo CPU-burst;
* **Tempo di attesa di un processo :** è il tempo totale che il processo   
  trascorre nella coda pronti;
* **Tempo medio di risposta :** è la media dei tempi di risposta di ogni processo;
* **Tempo medio di attesa :** è la media dei tempi di attesa di ogni processo.

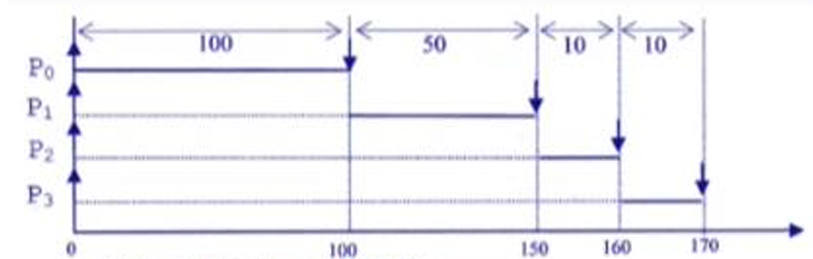
**FIRST COME FIRST SERVED (FCFS)**

E’ un algoritmo di short term scheduling senza priorità: i processi vengono mandati in esecuzione   
nello stesso ordine in cui arrivano in coda pronti.

E’ non preemptive.

*Es.*

|  |  |  |
| --- | --- | --- |
| **Processo** | **Istante di arrivo per la prima volta in coda pronti** | **CPU-burst** |
|  | 0 | 100 |
|  | 0 | 50 |
|  | 0 | 10 |
|  | 0 | 10 |

**

Valutiamo ora le prestazioni di FCFS:

* Il tempo medio di risposta e il tempo medio di attesa variano a seconda dell’ordine   
  in cui i processi arrivano nella coda pronti(in particolare, questi tempi saranno molto grandi   
  se nella coda pronti arrivano per primi dei processi con un lungo CPU-burst,   
  poiché questi verranno eseguiti per primi tenendo impegnata a lungo la CPU, e quindi   
  gli altri processi arrivati nella coda pronti dopo di loro dovranno aspettare molto tempo   
  prima di andare in esecuzione).

**SHORTEST JOB FIRST (SJF)**

E’ un algoritmo di short term scheduling a priorità statica: viene mandato in esecuzione il processo   
con il CPU-burst più breve tra quelli presenti in coda pronti.

E’ non preemptive.

*Es.*

|  |  |  |
| --- | --- | --- |
| **Processo** | **Istante di arrivo per la prima volta in coda pronti** | **CPU-burst** |
|  | 0 | 50 |
|  | 4 | 24 |
|  | 6 | 16 |

*Immagine che contiene linea, diagramma, Parallelo, Diagramma

Descrizione generata automaticamente*

Valutiamo ora le prestazioni di SJF:

* Il tempo medio di attesa e il tempo medio di risposta in generale si riducono rispetto a quelli   
  che si avrebbero con FCFS. Infatti, eseguendo per primi i processi con un CPU-burst più breve, il tempo di attesa e il tempo di risposta di questi ultimi si riducono più di quanto si allungano il tempo di attesa e il tempo di risposta dei processi con un CPU-burst più lungo.   
  Tuttavia, questi tempi dipendono ancora dall’ordine in cui i processi arrivano nella coda pronti   
  (in particolare, saranno molto grandi se nella coda pronti si hanno inizialmente   
  solo processi con un lungo CPU-burst, uno di questi va in esecuzione e successivamente entrano in coda pronti dei processi con un CPU-burst breve, poiché questi dovranno aspettare che il processo attualmente in esecuzione termini o si sospenda prima di andare   
  a loro volta in esecuzione);
* L’overhead è maggiore rispetto a quello che si avrebbe con FCFS, essendo SJF un algoritmo   
  con priorità;
* Potrebbe verificarsi la starvation dei processi con un lungo CPU burst se in coda pronticontinuano ad arrivare sempre processi con un breve CPU burst (dato che questi verranno eseguiti per primi, e quindi i processi con un lungo CPU burst dovranno aspettare a lungo prima di andare in esecuzione).

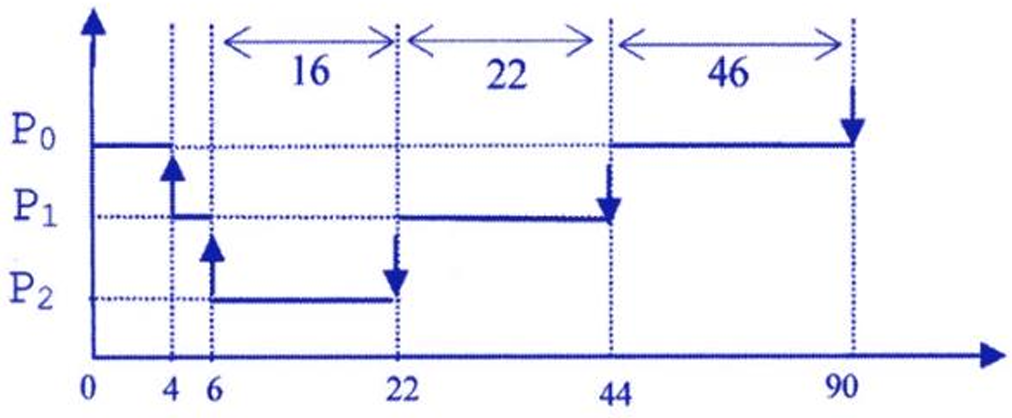
**SHORTEST-REMAINING-TIME-FIRST (SRTF)**

E’ un algoritmo di short term scheduling a priorità dinamica: viene mandato in esecuzione   
il processo con il CPU-burst rimanente più breve tra quelli presenti in coda pronti.

E’ preemptive: se durante l’esecuzione di un processo entra in coda prontiun altro processo   
con un CPU burst rimanente inferiore al CPU burst rimanente del processo in esecuzione,   
allora il processo in esecuzione viene interrotto forzatamente e messo nella coda pronti, e al suo posto va in esecuzione il processo con il CPU burst rimanente inferiore.

*Es.*

|  |  |  |
| --- | --- | --- |
| **Processo** | **Istante di arrivo per la prima volta in coda pronti** | **CPU-burst** |
|  | 0 | 50 |
|  | 4 | 24 |
|  | 6 | 16 |

**

Valutiamo ora le prestazioni di SRTF:

* Il tempo medio di attesa e il tempo medio di risposta si riducono ulteriormente rispetto a quelli   
  che si avrebbero con SJF, in quanto, essendo in più preemptive, SRTF ha il vantaggio   
  che questi tempi sono indipendenti dall’ordine in cui i processi arrivano nella coda pronti (infatti, anche se nella coda pronti ci fossero inizialmente solo processi con un lungo CPU-burst rimanente, uno di questi andasse in esecuzione e successivamente entrassero in coda pronti dei processi con un CPU-burst rimanente breve, il processo in esecuzione verrà interrotto forzatamente e messo nella coda pronti, e andranno via via in esecuzione i processi con un CPU-burst rimanente più breve);
* L’overhead è maggiore rispetto a quello che si avrebbe con SJF, in quanto SRTF dovrà andare   
  in esecuzione anche ogni volta che un processo arriva in coda pronti, per vedere   
  se il CPU burst rimanente di questo processo è inferiore a quello del processo attualmente   
  in esecuzione e, eventualmente, invocare il cambio di contesto;
* Potrebbe verificarsi la starvation dei processi con un lungo CPU burst rimanente   
  se in coda pronticontinuano ad arrivare sempre processi con un breve CPU burst rimanente (dato che questi verranno eseguiti per primi, e quindi i processi con un lungo CPU burst rimanente dovranno aspettare a lungo prima di andare in esecuzione).

Per applicare SJF e SRTF è necessario conoscere il CPU-burst dei processi, cosa che in generale   
non è nota a priori. La soluzione allora è stimarlo: per ogni processo viene stimato un CPU-burst   
pari alla media esponenziale dei suoi precedenti CPU-burst. Indicando con:

*Tn* il valore del CPU-burst *n*-esimo del processo;

*ETn* la stima del CPU-burst *n*-esimo del processo;

* *α* un valore compreso tra 0 e 1 (tipicamente, 0.5).

Scelta in maniera arbitraria la stima ET1 del primo CPU-burst, ETn+1 viene calcolata in questo modo:

ETn+1 = α · Tn + (1 – α) · ETn

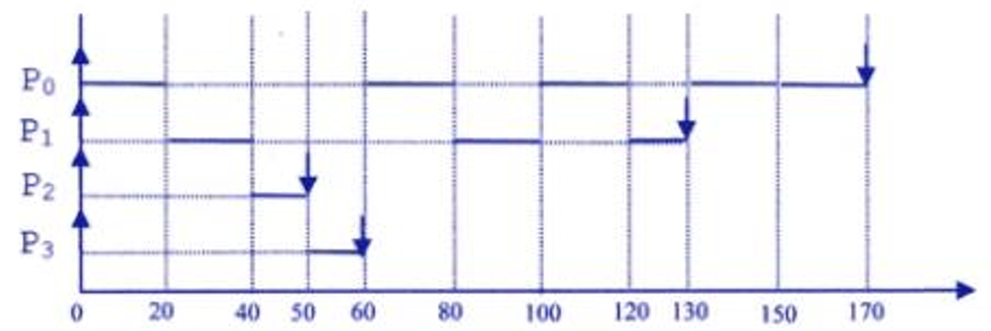
**ROUND ROBIN (RR)**

E’ un algoritmo di short term scheduling senza priorità: i processi vengono mandati in esecuzione   
nello stesso ordine in cui entrano nella coda pronti*,* e ogni processo viene eseguito per un intervallo   
di tempo fissato, detto *quanto*.

E’ preemptive: se allo scadere del quanto il processo in esecuzione non si è ancora sospeso   
o non è ancora terminato, questo viene interrotto forzatamente e messo nella coda pronti,   
e al suo posto va in esecuzione il prossimo processo in coda.

*Es. con un quanto = 20 ms*

|  |  |  |
| --- | --- | --- |
| **Processo** | **Istante di arrivo per la prima volta in coda pronti** | **CPU-burst** |
|  | 0 | 100 |
|  | 0 | 50 |
|  | 0 | 10 |
|  | 0 | 10 |



Valutiamo ora le prestazioni di RR:

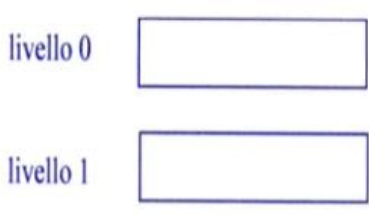
* Il tempo medio di attesa e il tempo medio di risposta in generale si riducono rispetto a quelli   
  che si avrebbero con FCFS. Infatti, eseguendo ogni processo per un intervallo di tempo fissato, i tempi di attesa e i tempi di risposta dei processi con un CPU-burst più breve   
  (a cui, tipicamente, basterà un solo quanto di tempo prima di sospendersi o terminare)   
  si riducono più di quanto si allungano il tempo di attesa e il tempo di risposta dei processi   
  con un CPU-burst più lungo.
* Bisogna scegliere bene la durata del quanto:
  + Con un quanto troppo breve, il cambio di contesto sarebbe troppo frequente,   
    e quindi sarebbe troppo frequente anche il conseguente overhead;
  + Con un quanto troppo lungo, le prestazioni sarebbero simili a quelle di FCFS.

**USO DI PIÙ CODE PRONTI**

I sistemi operativi più complessi fanno uso di più code pronti*,* ciascuna delle quali:

* Ha associata una certa priorità, dunque viene servita solo se le code prontidi priorità maggiore sono vuote;
* E’ gestita con un certo algoritmo di short-term scheduling.

Vediamo un esempio con due code pronti:



Supponiamo che la coda pronti di livello 0 sia a priorità maggiore. Possono presentarsi due problemi:

1. Potrebbe verificarsi la starvation dei processi che si trovano nella coda pronti di livello 1   
   se in quella di livello 0 continuano ad entrare processi (dato che questi verranno eseguiti per primi, e quindi i processi che si trovano nella coda pronti di livello 1 dovranno aspettare   
   a lungo prima di andare in esecuzione). Serve allora introdurre un qualche algoritmo di ageing in modo che, prima o poi, anche i processi che si trovano nella coda pronti di livello 1   
   possano andare in esecuzione, ed uno potrebbe essere che, se un processo rimane   
   nella coda pronti di livello 1 per più di *tot* tempo, allora viene spostato nella coda pronti   
   di livello 0;
2. Supponiamo che la coda pronti di livello 0 sia vuota e che venga mandato in esecuzione   
   un processo che si trovava nella coda pronti di livello 1. Se nel frattempo dovessero entrare   
   dei processi nella coda pronti di livello 0, cosa bisogna fare? Serve allora introdurre   
   un meccanismo che, quando è in esecuzione un processo proveniente dalla coda pronti   
   di livello 1 e un processo entra nella coda pronti di livello 0, il processo in esecuzione  
   viene interrotto forzatamente e messo in testa alla coda pronti di livello 1 (così che possa tornare in esecuzione il prima possibile - ma comunque dopo i processi che si trovano   
   nella coda pronti di livello 0 - e concludere ciò che ha iniziato), e il processo entrato   
   nella coda pronti di livello 0 va in esecuzione al suo posto.

Come scegliere in quale coda pronti inserire un processo per la prima volta o dopo essersi sospeso? Una soluzione potrebbe essere questa:

* Se il processo ha un breve CPU-burst, allora va inserito nella coda prontidi livello 0 (infatti, visto che terrà occupata la CPU per poco tempo, è meglio che venga mandato subito in esecuzione, così che possa subito fare quello che deve fare invece di farlo aspettare troppo inutilmente);
* Se il processo ha un lungo CPU-burst, allora va inserito nella coda pronti di livello 1 (infatti, visto che terrà occupata la CPU per tanto tempo, è meglio che venga mandato in esecuzione   
  solo dopo i processi con un breve CPU-burst, così da non farli aspettare troppo inutilmente);

Un’altra soluzione è usare delle **code pronti con feedback**. Vediamo un esempio con tre code pronti:

* Una coda pronti di livello 0, a priorità più elevata, gestita con RR usando un quanto *=* 10ms;
* Una coda prontidi livello 1, a priorità intermedia, gestita con RR usando un quanto *=* 50ms;
* Una coda prontidi livello 2, a priorità più bassa, gestita con FCFS.

Immagine che contiene testo, linea, Carattere, schermata

Descrizione generata automaticamente

Quando un processo deve essere inserito in coda pronti per la prima volta o dopo essersi sospeso,   
viene inserito nella coda pronti di livello 0. Quando andrà in esecuzione, se allo scadere   
del quanto di 10ms non si è ancora sospeso o non è ancora terminato, il processo   
viene interrotto forzatamente e messo nella coda pronti di livello 1. Quando tornerà in esecuzione,   
se allo scadere del quanto di 50ms non si è ancora sospeso o non è ancora terminato, il processo   
viene interrotto forzatamente e messo nella coda pronti di livello 2.

Con questa soluzione, ogni processo verrà automaticamente collocato nella coda prontipiù adeguata al suo CPU-burst.

**SCHEDULAZIONE DI SISTEMI IN TEMPO REALE**

I **sistemi in tempo reale** sono dei calcolatori che acquisiscono dati da un ambiente esterno   
attraverso sensori e agiscono su tale ambiente attraverso attuatori.

I processi in esecuzione su un sistema in tempo reale sono dei processi che rientrano in coda pronti periodicamente e che devono sospendersi entro una certa deadline. Ciascuno di questi processi   
può essere descritto dunque attraverso i seguenti parametri:

* **L’istante di richiesta :**istante *k*-esimo in cui il processo entra in coda *pronti*per la prima volta o dopo essersi sospeso;
* **L’istante di deadline :** istante *k*-esimo entro il quale il processo deve sospendersi;
* **Il CPU-burst :** CPU-burst *k*-esimo del processo;
* **Il periodo :** tempo tra un istante di richiesta e il successivo;

Noi ci concentreremo sui sistemi hard real-time, e cioè sistemi in tempo reale in cui   
nessuna deadline può essere violata. Diremo che un insieme di processi di processi   
è **schedulabile** su un sistema hard real-time usando un certo algoritmo di short-term scheduling   
se, attraverso questo algoritmo, non viene violata alcuna deadline.

Prima di vedere alcuni algoritmi di short-term scheduling per sistemi hard real-time,   
facciamo delle semplificazioni:

* Il CPU-burst *k*-esimo di un processo non sempre è possibile conoscerlo a priori,   
  né conviene stimarlo rischiando di violare la deadline. Quello che faremo, allora,   
  sarà considerare sempre il suo CPU-burst nel caso peggiore, che tipicamente   
  si può conoscere a priori.
* Supporremo che, in ogni periodo di un processo, l’istante di deadline   
  corrisponda con l’istante di richiesta successivo:

Inoltre, considereremo solo sistemi hard real-time costituiti esclusivamente da processi periodici.   
Questi sistemi, dunque, saranno a loro volta periodici, con un periodo pari al *mcm* dei periodi   
dei processi:

Questo, insieme alle semplificazioni introdotte, ci permette di dire se un insieme di processi   
è schedulabile su un sistema hard real-time usando un certo algoritmo di short-term scheduling semplicemente analizzando cosa succede in un (perché tanto il comportamento del sistema osservato in un si ripeterà per ogni ).

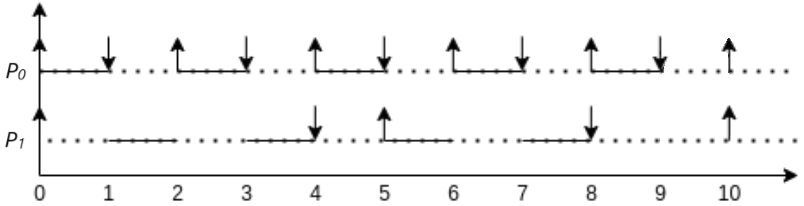
**RATE MONOTONIC (RM)**

E’ un algoritmo di short term scheduling per sistemi hard real-time a priorità statica: viene mandato   
in esecuzione il processo con il periodo più breve tra quelli presenti in coda pronti.

E’ preemptive: se durante l’esecuzione di un processo entra in coda prontiun altro processo   
con un periodo inferiore al periodo del processo in esecuzione, allora il processo in esecuzione   
viene interrotto forzatamente e messo nella coda pronti, e al suo posto va in esecuzione il processo   
con il periodo inferiore.

*Es.*

|  |  |  |
| --- | --- | --- |
| **Processo** | **T** | **C** |
|  | 2 | 1 |
|  | 5 | 2 |
|  | | |

**

Valutiamo ora le prestazioni di RM:

* Si può dimostrare che, tra tutti gli algoritmi di short-term scheduling per sistemi hard real-time a priorità statica, questo è ottimo, e cioè se un insieme di processi non è schedulabile   
  su un sistema hard real-time usando RM, allora non è schedulabile su un sistema   
  hard real-time con nessun altro algoritmo di short-term scheduling a priorità statica.

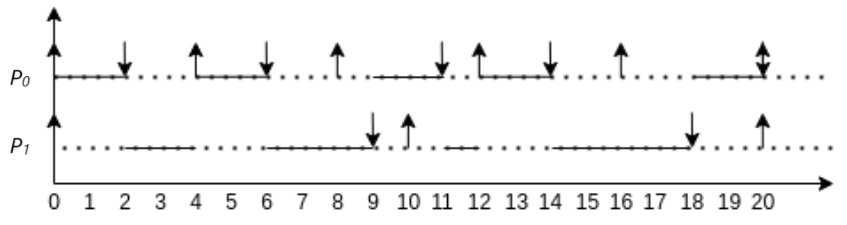
**EARLIEST DEADLINE FIRST (EDF)**

E’ un algoritmo di short term scheduling per sistemi hard real-time a priorità dinamica: viene mandato in esecuzione il processo con la deadline più vicina tra quelli presenti in coda pronti.

E’ preemptive: se durante l’esecuzione di un processo entra in coda prontiun altro processo   
con una deadline più vicina rispetto a quella del processo in esecuzione, allora il processo   
in esecuzione viene interrotto forzatamente e messo nella coda pronti, e al suo posto va in esecuzione il processo con la deadline più vicina.

*Es.*

|  |  |  |
| --- | --- | --- |
| **Processo** | **T** | **C** |
|  | 4 | 2 |
|  | 10 | 5 |
|  | | |

**

Valutiamo ora le prestazioni di EDF:

* Si può dimostrare che, tra tutti gli algoritmi di short-term scheduling per sistemi hard real-time a priorità dinamica, questo è ottimo, e cioè se un insieme di processi non è schedulabile   
  su un sistema hard real-time usando EDF, allora non è schedulabile su un sistema   
  hard real-time con nessun altro algoritmo di short-term scheduling a priorità dinamica.

**CRITERI PER LA SCHEDULABILITA’ DI UN INSIEME DI PROCESSI SU UN SISTEMA   
HARD REAL-TIME SFRUTTANDO IL FATTORE DI UTILIZZAZIONE**

Dato un insieme di processi periodici da schedulare su un sistema hard real-time,   
si definisce ***fattore di utilizzazione U*:**

Per ogni processo, si fa il rapporto tra il tempo per cui il processo   
dovrà usare la CPU in un periodo e il periodo stesso.

Esistono alcune condizioni che, sfruttando , danno informazioni sull’eventuale schedulabilità   
di un insieme di processi su un sistema hard real-time.

**C.N. SULLA SCHEDULABILITA’**

*C.N.*affinché un insieme di processi periodici sia schedulabile su un sistema   
hard real-time è che risulti:

*DIM*

In ogni , ogni processo entra in coda pronti per la prima volta o dopo essersi sospeso  
per un certo numero di volte , pari al rapporto tra e :

Ogni volta che entra in coda pronti per la prima volta o dopo essersi sospeso,   
dovrà tenere occupata la CPU per un tempo prima di sospendersi. In un ,   
dunque, dovrà tenere occupata la CPU per un tempo totale ,   
perciò il tempo totale per cui l’insieme di processi dovrà tenere occupata la CPU in un   
sarà la somma dei tempi totali relativi a ciascun processo:

Sostituisco

Esplicito i vari

Metto in evidenza

Affinché l’insieme di processi sia schedulabile su un sistema hard real-time, il tempo totale per cui   
dovrà tenere occupata la CPU in un deve necessariamente essere non superiore a :

Divido entrambi i membri per

;

E’ quindi dimostrato che, affinché un insieme di processi periodici sia schedulabile   
su un sistema hard real-time, deve necessariamente risultare .

**C.S. SULLA SCHEDULABILITA’ USANDO RM**

*C.S****.*** affinché un insieme di processi periodici sia schedulabile su un sistema   
hard real-time usando RM è che risulti:

*NON DIMOSTRIAMO*

**SINCRONIZZAZIONE TRA I PROCESSI** *29-10-24*

**MODELLI DI INTERAZIONE TRA I PROCESSI**

Possiamo distinguere 2 modelli di interazione tra i processi:

1. **Modello di interazione a memoria comune:** ogni processo, oltre ad avere una propria memoria privata, ha accesso ad una memoria condivisa tra tutti i processi;
2. **Modello di interazione a memoria locale:** ogni processo ha solo una propria memoria privata.

**MODELLO DI INTERAZIONE A MEMORIA COMUNE: MUTUA ESCLUSIONE**

Nel modello di interazione a memoria comune, tutti i processi condividono le risorse presenti   
nella memoria comune. Bisogna dunque garantire che l’accesso ad ogni risorsa condivisa   
avvenga in **mutua esclusione**, altrimenti tale risorsa potrebbe finire in uno stato inconsistente.

Chiamiamo **sezione critica** un insieme di operazioni di un processo su una risorsa comune.   
Per garantire l’accesso in mutua esclusione ad ogni risorsa condivisa, l’idea è che ogni processo,   
prima di eseguire una sezione critica, esegua un cosiddetto *prologo*, e cioè una serie di operazioni   
che gli permettano di ottenere l’accesso esclusivo alla risorsa condivisa interessata.   
Una volta eseguita la sezione critica, il processo deve eseguire un cosiddetto *epilogo*, e cioè   
una serie di operazioni in cui segnala che la risorsa può essere ora acceduta da altri processi.

Una prima soluzione per garantire l’accesso in mutua esclusione ad ogni risorsa condivisa   
potrebbe essere associare a ciascuna di queste risorse una variabile condivisa *occupato*, che vale:

* 1 se la risorsa è *occupata*, e cioè se un processo ha al momento l’accesso esclusivo ad essa;
* 0 se la risorsa è *libera*, e cioè se nessun processo ha al momento l’accesso esclusivo ad essa.

A questo punto, il prologo e l’epilogo possono essere fatti in un modo del genere:

Il processo attende fintantoché la risorsa condivisa   
interessata è occupata, e cioè fintantoché *occupato* vale 1.

Non appena la risorsa è libera, e cioè *occupato* vale 0,   
il processo pone *occupato* a 1, ottenendo così l’accesso esclusivo ad essa.

A questo punto, il processo esegue la sezione critica,   
dopodiché segnala che la risorsa è libera, ponendo *occupato* a 0.

while(occupato == 1);

occupato = 1;

*sezione critica*

occupato = 0;

Questa soluzione, tuttavia, è sbagliata, in quanto un processo può essere interrotto forzatamente   
tra quando legge *occupato* a 0 a quando modifica *occupato* a 1. Supponiamo infatti di avere 2 processi P1 e P2 che vogliono accedere alla stessa risorsa condivisa. Una possibile sequenza delle operazioni da parte dei due processi potrebbe essere la seguente:

1. P1 esegue l’istruzione while e trova *occupato* che vale 0;
2. P2 esegue l’istruzione while e trova *occupato* che vale 0;
3. P1 pone *occupato* a 1 ed entra nella sezione critica;
4. P2 pone *occupato* a 1 ed entra nella sezione critica.

Si avrà così che entrambi i processi sono contemporaneamente in una sezione critica,   
e questo potrebbe portare la risorsa condivisa interessata in uno stato inconsistente.

Alcuni processori permettono di copiare il contenuto di una locazione di memoria in un registro   
e modificare il contenuto di tale locazione con una sola istruzione Assembly, e quindi   
in maniera atomica. Queste istruzioni sono dette istruzioni **test-and-set**. Un’istruzione di questo tipo   
è TSL *x*, %*reg*, che copia il valore della locazione di memoria *x* nel registro *reg* e pone *x* a 1.

Possiamo allora modificare la soluzione vista prima per garantire l’accesso in mutua esclusione   
ad ogni risorsa condivisa introducendo due funzioni lock() e unlock(), così definite:

lock(*occupato*):

Il processo che invoca lock(*occupato*) tenta continuamente   
di ottenere l’accesso esclusivo alla risorsa condivisa interessata, ponendo *occupato* a 1. Questo viene fatto fintantoché   
il valore precedente di *occupato* salvato in *reg* non è 0, e cioè fintantoché la risorsa non si è liberata, e ponendo *occupato* a 1   
il processo è riuscito effettivamente ad ottenere l’accesso esclusivo ad essa.

TSL *occupato, %reg*

CMP $0, %*reg*

JNE lock

RET

unlock(*occupato*):

Il processo che invoca unlock(*occupato*) segnala che   
la risorsa condivisa è libera, ponendo *occupato* a 0

MOV $0, *occupato*;

RET

A questo punto, il prologo e l’epilogo possono essere fatti in un modo del genere:

lock(*occupato*);

Il processo invoca lock(*occupato*), attendendo fintantoché   
non acquisisce l’accesso esclusivo alla risorsa condivisa.

Fatto ciò, il processo esegue la sezione critica, dopodiché invoca unlock(*occupato*), segnalando così che la risorsa è libera.

*sezione critica*

unlock(*occupato*)

Questa soluzione ora è corretta, in quanto, usando l’istruzione TSL *occupato* %reg, un processo   
non può più essere interrotto forzatamente tra quando legge *occupato* a 0 a quando modifica   
*occupato* a 1. Tuttavia, continua ad essere inefficiente, in quanto un processo deve rimanere   
in attesa attiva prima di ottenere l’accesso esclusivo alla risorsa condivisa interessata.

Nell’ambito della sincronizzazione tra i processi sono molto usati i semafori.

Un **semaforo** ***s***è una struttura dati costituita da:

* Una variabile *s.value* intera non negativa;
* Una coda *s.queue* di descrittori di processi sospesi.

E’ possibile accedere ad un semaforo solo attraverso due primitive di sistema **wait()** e **signal()**,   
e quindi in maniera atomica. Queste primitive sono fatte in un modo del genere:

void wait(s) {

if(s.value == 0){

*il processo viene sospeso e il suo descrittore viene inserito in fondo a s.queue*

}

else{

s.value--;

}

}

void signal(s) {

if(*esiste almeno un descrittore di processo in s.queue*){

*il descrittore in testa a s.queue viene estratto*

*e inserito in coda pronti*

}

else{

s.value++;

}

}

Un’altra soluzione per garantire l’accesso in mutua esclusione ad ogni risorsa condivisa   
consiste quindi nell’associare a ciascuna di queste risorse un semaforo *mutex* di mutua esclusione,   
e cioè un semaforo con valore iniziale 1.

A questo punto, il prologo e l’epilogo possono essere fatti in un modo del genere:

wait(mutex);

Il processo invoca wait(*mutex*), sospendendosi fintantoché   
non acquisisce l’accesso esclusivo alla risorsa condivisa.

Fatto ciò, il processo esegue la sezione critica, dopodiché invoca signal(*mutex*), segnalando che ora la risorsa è libera.

*sezione critica*

signal(mutex)

Questa soluzione è corretta ed efficiente.

**MODELLO DI INTERAZIONE A MEMORIA COMUNE: COMUNICAZIONE** *05-11-24*

Nel modello di interazione a memoria comune, tutti i processi condividono le risorse presenti   
nella memoria comune, che dunque possono usare per scambiare messaggi tra loro.   
Questo, tuttavia, richiede che sia garantita la **comunicazione** tra i processi, altrimenti questi   
non riuscirebbero effettivamente a scambiarsi messaggi tra loro.

**PROBLEMA DI COMUNICAZIONE PRODUTTORI-CONSUMATORI**

Nel **problema di comunicazione produttori-consumatori** si hanno *n* processi *produttori*,   
che producono messaggi, e *m* processi *consumatori*, che consumano questi messaggi.   
Lo scambio dei messaggi avviene attraverso un buffer condiviso.

Ogni produttore, ciclicamente:

1. Produce un nuovo messaggio;
2. Attende fintantoché non c’è spazio disponibile nel buffer;
3. Deposita il messaggio.

Ogni consumatore, ciclicamente:

1. Attende fintantoché non c’è un messaggio disponibile nel buffer;
2. Preleva il messaggio;
3. Consuma il messaggio

Il buffer:

* E’ inizialmente vuoto;
* E’ in grado di contenere un numero limitato di messaggi;
* Può essere acceduto da un solo processo per volta, dunque bisogna garantire che l’accesso   
  ad esso avvenga in mutua esclusione.

Per gestire la comunicazione possiamo usare innanzitutto:

* Un semaforo *spazio\_disponibile*, con valore iniziale pari al numero di messaggi   
  che può contenere il buffer, e che in ogni istante avrà come valore il numero di posizioni libere   
  presenti nel buffer;
* Un semaforo *messaggio\_disponibile*, con valore iniziale 0, e che in ogni istante   
  avrà come valore il numero di messaggi presenti nel buffer.

Supponiamo che il buffer possa contenere un solo messaggio per volta. Il valore iniziale   
di *spazio\_disponibile*, dunque, deve essere 1.

In questo caso, allora, il codice di un processo produttore e di un processo consumatore   
possono essere fatti in questo modo:

**Produttore**

do{

Il processo produttore, una volta prodotto un messaggio,   
si sospende fintantoché non c’è spazio disponibile nel buffer.   
Fatto ciò, il processo deposita il messaggio nel buffer, dopodiché   
segnala che c’è un nuovo messaggio disponibile nel buffer.

*produco il messaggio*

wait(spazio\_disponibile)

*deposito il messaggio*

signal(messaggio\_disponibile)

} while(!fine);

**Consumatore**

do{

Il processo consumatore si sospende fintantoché non c’è   
un messaggio disponibile nel buffer.

Fatto ciò, il processo preleva il messaggio dal buffer, dopodiché segnala che c’è del nuovo spazio disponibile nel buffer.

Infine, il processo consuma il messaggio prelevato.

wait(messaggio\_disponibile)

*prelevo il messaggio dal buffer*

signal(spazio\_disponibile)

*consumo il messaggio*

} while(!fine)

Poiché il buffer può contenere un solo messaggio per volta, questa soluzione garantisce anche   
che l’accesso al buffer condiviso avvenga in mutua esclusione. Infatti:

* Un processo produttore accederà al buffer solo se c’è spazio disponibile,   
  e quindi se il buffer è vuoto. Un processo consumatore, invece, accederà al buffer   
  solo se c’è un messaggio disponibile, e quindi se il buffer è pieno. Non è possibile dunque che due processi di tipologie diverse possano trovarsi contemporaneamente in una sezione critica;
* Poiché il valore di *spazio\_disponibile* e *messaggio\_disponibile* può essere al massimo 1,   
  questi semafori agiscono anche da semafori di mutua esclusione, impedendo così   
  che due processi di una stessa tipologia (visto che ogni produttore invoca wait() sullo stesso semaforo *spazio\_disponibile*, mentre ogni consumatore invoca wait() sullo stesso semaforo *messaggio\_disponibile*) possano trovarsi contemporaneamente in una sezione critica.

Supponiamo adesso che il buffer possa contenere *N* messaggi per volta. Il valore iniziale   
di *spazio\_disponibile*, dunque, deve essere *N*. Partendo dalla soluzione vista prima,  
questa adesso non garantisce più che l’accesso al buffer condiviso avvenga in mutua esclusione, infatti:

* Se il buffer presenta almeno un messaggio e non è ancora pieno, un processo produttore   
  può accederci perché c’è dello spazio disponibile, e un processo consumatore può accederci perché c’è un messaggio disponibile. E’ possibile, dunque, che due processi di tipologie diverse possano trovarsi contemporaneamente in una sezione critica;
* Poiché il valore di *spazio\_disponibile* e *messaggio\_disponibile* può essere > 1, questi semafori non agiscono più anche da semafori di mutua esclusione, per cui non impediscono più   
  che due processi di una stessa tipologia possano trovarsi contemporaneamente   
  in una sezione critica.

Per garantire che l’accesso al buffer condiviso avvenga in mutua esclusione, dunque,   
bisogna aggiungere un semaforo di mutua esclusione *mutex* associato a tale buffer.

Il codice di un processo produttore e di un processo consumatore andranno dunque modificati aggiungendo attorno alla sezione critica i consueti prologo ed epilogo:

**Produttore**

do{

*produco il messaggio*

Il processo produttore, una volta prodotto un messaggio,   
si sospende fintantoché non c’è spazio disponibile nel buffer.   
Fatto ciò, il processo si sospende fintantoché non acquisisce l’accesso esclusivo al buffer.

A questo punto, il processo deposita il messaggio,   
dopodiché segnala che ora il buffer è libero, e segnala che c’è   
un nuovo messaggio disponibile nel buffer.

wait(spazio\_disponibile)

wait(mutex)

*deposito il messaggio*

signal(mutex)

signal(messaggio\_disponibile)

} while(!fine)

**Consumatore**

do{

Il processo consumatore si sospende fintantoché non c’è   
un messaggio disponibile nel buffer. Fatto ciò, il processo si sospende fintantoché non acquisisce l’accesso esclusivo al buffer.   
A questo punto, il processo preleva il messaggio dal buffer,   
dopodiché segnala che ora il buffer condiviso è libero, e segnala che c’è del nuovo spazio disponibile nel buffer.

Infine, il processo consuma il messaggio prelevato.

wait(messaggio\_disponibile)

wait(mutex)

*prelevo il messaggio dal buffer*

signal(mutex)

signal(spazio\_disponibile)

*consumo il messaggio*

} while(!fine)

**NB:** attento all’ordine dei semafori! Il processo prima attende che ci sia spazio/un messaggio disponibile nel buffer,   
e poi acquisisce l’accesso esclusivo al buffer. Questo viene fatto:

* Per continuità con la soluzione precedente, aggiungendo solo prologo ed epilogo attorno alla sezione critica;
* Perché se prima il processo acquisisse l’accesso esclusivo al buffer e poi si sospendesse fintantoché non c’è spazio/un messaggio disponibile nel buffer, dato che l’accesso esclusivo è ancora del processo che si è sospeso, nessun altro processo potrebbe accedere   
  al buffer per prelevare/depositare un messaggio, e quindi il processo rimarrebbe sospeso per sempre.

**PROBLEMA DI COMUNICAZIONE LETTORI-SCRITTORI**

Nel **problema di comunicazione lettori-scrittori** si ha un data set condiviso,   
*n* processi *lettori*, che possono leggere il data set, e *m* processi *scrittori*, che possono scrivere   
il data set:

* Un lettore che vuole accedere al data set deve attendere fintantoché nessuno scrittore   
  ci sta accedendo (e quindi, i lettori devono accedere al data set in mutua esclusione   
  rispetto agli scrittori);
* Uno scrittore che vuole accedere al data set deve attendere fintantoché nessun altro processo ci sta accedendo (e quindi, uno scrittore deve accedere al data set in mutua esclusione  
  sia rispetto ai lettori, sia rispetto agli altri scrittori);

Per gestire la comunicazione possiamo usare allora:

* Una variabile intera condivisa *readcount*, con valore iniziale 0, e che in ogni istante avrà   
  come valore il numero di lettori che stanno accedendo al data set;
* Un semaforo di mutua esclusione *mutex\_readcount* associato alla variabile *readcount*,   
  per garantire l’accesso in mutua esclusione a tale variabile;
* Un semaforo di mutua esclusione *mutex\_dataset* associato al data set, per garantire   
  l’accesso in mutua esclusione al data set per tutti i lettori o per uno scrittore.

A questo punto, il codice di un processo lettore e di un processo scrittore possono essere fatti   
in questo modo:

**Scrittore**

do{

Il processo scrittore si sospende fintantoché non acquisisce l’accesso esclusivo al data set.

Fatto ciò, il processo scrive il data set, dopodiché segnala che ora   
il data set è libero.

wait(mutex\_dataset)

*scrivo il data set*

signal(mutex\_dataset)

} while(TRUE);

**Lettore**

do{

Il processo lettore si sospende fintantoché non acquisisce l’accesso esclusivo alla variabile *readcount*, la incrementa (essendo lui un lettore che a breve starà accedendo al data set),   
e se a seguito di ciò *readcount* vale 1, e dunque al momento non c’è nessun lettore che sta accedendo al data set, il processo   
si sospende fintantoché non acquisisce l’accesso esclusivo   
al data set per tutti i lettori.

Fatto ciò, il processo segnala che ora la variabile *readcount* è libera e legge il data set.

Successivamente, il processo si sospende nuovamente fintantoché non acquisisce l’accesso esclusivo alla variabile readcount, la decrementa (avendo finito di accedere il data set),   
e se a seguito di ciò readcount vale 0, e dunque non ci sono più lettori che stanno accedendo al data set, il processo segnala   
che ora il data set è libero.

Infine, il processo segnala che ora la variabile readcount è libera.

wait(mutex\_readcount);

readcount++;

if(readcount == 1){

wait(mutex\_dataset);

}

signal(mutex\_readcount);

*leggo il dataset*

wait(mutex\_readcount);

readcount--;

if(readcount == 0){

signal(mutex\_dataset);

}

signal(mutex\_readcount);

} while(TRUE)

**PROBLEMA DI COMUNICAZIONE DEI 5 FILOSOFI** *07-11-24*

Nel **problema di comunicazione dei 5 filosofi** si ha una ciotola di riso condivisa al centro di un tavolo,  
5 processi filosofi seduti intorno al tavolo, numerati in senso orario da 0 a 4,   
e 5 bacchette condivise, ciascuna tra un filosofo e un altro, da usare per mangiare il riso.

Ogni filosofo, periodicamente, passa per tre stati:

1. *Thinking*: non mangia il riso;
2. *Hungry*: acquisisce la bacchetta alla propria destra e la bacchetta alla propria sinistra   
   per poter mangiare il riso;
3. *Eating*: mangia il riso, dopodiché libera le bacchette.

Ogni bacchetta può essere usata da un solo filosofo per volta, dunque bisogna garantire che l’accesso ad essa avvenga in mutua esclusione.

Una prima idea per poter gestire la comunicazione potrebbe essere usare un array *chopstick*  
di 5 semafori di mutua esclusione, dove *chopstick[i]* è associato alla bacchetta alla destra   
del filosofo *i*-esimo, per garantire l’accesso in mutua esclusione a ogni bacchetta.

A questo punto, il codice del processo filosofo *i*-esimo potrebbe essere fatto in questo modo:

do {

Inizialmente, il processo filosofo pensa.

Dopo un po’, il processo filosofo prima si sospende fintantoché   
non acquisisce l’accesso esclusivo alla bacchetta alla propria destra, e poi si sospende nuovamente fintantoché non acquisisce l’accesso esclusivo alla bacchetta alla propria sinistra.

Fatto ciò, il processo filosofo mangia il riso, dopodiché segnala   
che la bacchetta alla propria sinistra è libera, e segnala   
che la bacchetta alla propria destra è libera.

*penso*

wait(chopstick[i]);

wait(chopstick[(i+1)%5]);

*mangio il riso*

signal(chopstick[i+1]%5);

signal(chopstick[i]);

} while(TRUE);

Questa soluzione, tuttavia, è sbagliata, in quanto può portare a deadlock. Infatti, per esempio,   
può succedere che ogni processo filosofo venga interrotto forzatamente dopo aver acquisito solo   
la bacchetta alla propria destra, e dunque, quando tornerà in esecuzione, si sospenderà fintantoché non avrà acquisito la bacchetta alla propria sinistra, occupata dal processo filosofo alla sua sinistra, che è a sua volta nella stessa situazione.

Nell’ambito della sincronizzazione tra i processi, oltre ai semafori, sono molto utilizzati anche i *monitor*.

Un **monitor** è una struttura dati in cui è possibile definire delle variabili condivise e delle funzioni.   
E’ possibile accedere alle variabili condivise di un monitor solo attraverso le funzioni   
del monitor stesso, e in un monitor può trovarsi un solo processo non sospeso per volta.

La sintassi che useremo per definire un monitor è la seguente:

monitor monitor-name {

*variabile\_condivisa1*;

*variabile\_condivisa2*;

…

*variabile\_condivisaN;*

*funzione1*(…){…};

*funzione2*(…){…};  
…

*funzioneM*(…){…};

*codice\_di\_inizializzazione*(…) {…};

}

In un monitor è possibile definire anche delle *variabili condition*.

Una **variabile condition** è una variabile che non ha associato alcun valore, ma ha associata   
una coda di processi sospesi, sulla quale è possibile agire attraverso due funzioni:

* wait(): il processo che la invoca si sospende e viene aggiunto alla coda della variabile condition  
  su cui è stata invocata la funzione;
* signal():risveglia un processo tra quelli presenti nella coda della variabile condition   
  su cui è stata invocata la funzione.

Vediamo come è possibile implementare un monitor, e in particolare come garantire che in un monitor possa trovarsi un solo processo non sospeso per volta.

Innanzitutto, noi considereremo monitor che adottano l’approccio **signal and wait**: se c’è   
un processo P sospeso su una variabile condition di un monitor, e il processo Q non sospeso   
che si trova attualmente nel monitor risveglia P, allora Q si sospende, e così P sarà l’unico processo non sospeso che si troverà nel monitor.

Possiamo usare allora:

* Un semaforo di mutua esclusione *mutex\_monitor* associato al monitor, per garantire l’accesso in mutua esclusione al monitor;
* Un semaforo *next*, con valore iniziale 0, su cui si sospenderà ogni processo dopo averne risvegliato un altro che era sospeso su una variabile condition del monitor;
* Una variabile intera *next\_count*, con valore iniziale 0, e che in ogni istante avrà come valore   
  il numero di processi sospesi sul semaforo *next*.

A questo punto, ogni funzione del monitor si può implementare in questo modo:

Il processo invocante si sospende fintantoché non acquisisce l’accesso esclusivo al monitor.

Fatto ciò, il processo invocante esegue il corpo effettivo   
della funzione.

Infine:

* Se c’è qualche processo sospeso sul semaforo *next*,   
  il processo invocante lo risveglia, cedendogli così   
  l’accesso esclusivo al monitor;
* Altrimenti il processo invocante segnala che ora   
  il monitor è libero.

wait(mutex\_monitor);

*corpo\_effettivo\_della\_funzione*

if(next\_count > 0){

signal(next);

}

else{

signal(mutex);

}

Ogni variabile condition *x*, invece, possiamo implementarla attraverso:

* Un semaforo *x\_sem*, con valore iniziale 0, su cui si sospenderà ogni processo   
  dopo aver invocato x.wait();
* Una variabile intera *x\_count*, con valore iniziale 0, e che in ogni istante avrà come valore   
  il numero di processi sospesi sul semaforo *x\_sem*.

A questo punto, le funzioni x.wait() e x.signal() si possono implementare in questo modo:

Il processo invocante incrementa *x\_count*, visto che a breve   
si aggiungerà ai processi sospesi sul semaforo *x\_sem*.

Fatto ciò:

* Se c’è qualche processo sospeso sul semaforo *next*,   
  il processo invocante ne risveglia uno, cedendogli così   
  l’accesso esclusivo al monitor;
* Altrimenti il processo invocante segnala che ora   
  il monitor è libero.

A questo punto, il processo invocante si sospende   
sul semaforo *x\_sem*.

Dopo essere stato risvegliato, il processo invocante decrementa *x\_count*, visto che ora sul semaforo *x\_sem*c’è un processo sospeso in meno.

void x.wait() {

x\_count++;

if(next\_count > 0){

signal(next);

}

else{

signal(mutex);

}

wait(x\_sem);

x\_count--;

}

Se c’è qualche processo sospeso sul semaforo *x\_sem*,   
il processo invocante incrementa *next\_count*, visto che a breve egli stesso si aggiungerà ai processi sospesi sul semaforo *next*, risveglia un processo sospeso sul semaforo *x\_sem*,   
cedendogli così l’accesso esclusivo al monitor,   
e si sospende sul semaforo *next*.

Dopo essere stato risvegliato, il processo invocante decrementa *next\_count*, visto che ora sul semaforo *next*   
c’è un processo sospeso in meno.

void x.signal() {

if(x\_count > 0){

next\_count++;

signal(x\_sem);

wait(next);

next\_count--;

}

}

**NB**: Quando un processo deve smettere di usare il monitor alla fine di una funzione o prima di sospendersi per la wait:

* Se c’è un processo sospeso sul semaforo next, lo risveglia, cedendogli l’accesso esclusivo al monitor;
* Altrimenti, segnala che il monitor è libero.

Nella signal(), invece, il processo smette di usare il monitor dopo aver risvegliato un processo sospeso sul semaforo *x\_sem*, processo a cui avrà ceduto quindi l’accesso esclusivo al monitor.

Tornando al problema dei 5 filosofi, possiamo innanzitutto imporre che ogni filosofo debba acquisire entrambe le bacchette contemporaneamente. In questo modo, per vedere se un filosofo affamato   
può acquisire le bacchette, basta vedere se il filosofo alla sua destra e quello alla sua sinistra non stanno mangiando.

A questo punto, possiamo definire il seguente monitor:

monitor DiningPhilosophers {

enum {THINKING, HUNGRY, EATING} state[5];

condition self[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();

}

}

void pickup(int i){

state[i] = HUNGRY;

test(i);

if(state[i] != EATING){

self[i].wait;

}

}

void putdown(int i){

state[i] = THINKING;

test((i + 4) % 5);

test((i + 1) % 5);

}

initialization\_code(){

for(int i = 0; i < 5; i++){

state[i] = THINKING;

}

}

Le variabili condivise sono:

* Un array *state[5]*, dove l’elemento *i*-esimo contiene lo stato dell’*i*-esimo filosofo;
* Un array *self[5]* di variabili condition, dove la variabile *i*-esima sarà quella su cui l’*i*-esimo filosofo si sospenderà   
  nel caso in cui vorrà acquisire le bacchette ma non potrà farlo.

La funzione test(int *i*) sarà invocata o dal filosofo *i*-esimo o da un suo vicino per vedere se il filosofo *i*-esimo   
può acquisire le bacchette.

Se il filosofo *i*-esimo è affamato, il filosofo alla sua destra non sta mangiando e il filosofo alla sua sinistra   
non sta mangiando, allora il filosofo *i*-esimo può acquisire le bacchette, iniziando così a mangiare.   
Cambio allora lo stato del filosofo in *eating*, visto che a breve inizierà a mangiare, e lo risveglio nel caso in cui   
si fosse sospeso sulla propria variabile condition perché voleva acquisire le bacchette ma non ha potuto farlo.

La funzione pickup(int *i*) sarà invocata dal filosofo *i*-esimo per acquisire le bacchette.

Il filosofo cambia così il suo stato in *hungry*, vede se può acquisire le bacchette, e nel caso in cui non potesse   
(e quindi, se il suo stato non è passato a *eating*) si sospende sulla propria variabile condition.

La funzione putdown(int *i*) sarà invocata dal filosofo *i*-esimo per liberare le bacchette.

Il filosofo cambia così il suo stato in *thinking*. Fatto ciò, vede se il filosofo alla sua destra e il filosofo alla sua sinistra   
possono acquisire le bacchette, così da risvegliarli eventualmente nel caso in cui si fossero sospesi   
sulla propria variabile condition perché volevano acquisire le bacchette ma non hanno potuto farlo.

Il codice del processo filosofo *i*-esimo, dunque, sarà fatto in questo modo:

do {

*penso*

Inizialmente, il processo filosofo pensa.

Dopo un po’, il processo filosofo acquisisce le bacchette,   
dopodiché mangia.

Infine, il processo filosofo libera le bacchette.

pickup(i);

*mangio*

putdown(i);

} while(TRUE);

**MODELLO DI INTERAZIONE A MEMORIA COMUNE: DEADLOCK** *13-11-24*

Un problema che può verificarsi nel modello di interazione a memoria comune quando si hanno almeno 2 processi e 2 risorse condivise è il **deadlock**.

Un insieme di processi è in deadlock se ogni processo dell’insieme è sospeso fintantoché   
non acquisisce l’accesso esclusivo a una risorsa occupata da un altro processo dello stesso insieme.

Supponiamo che ogni processo, quando vuole accedere in mutua esclusione ad una risorsa,   
si comporti in questo modo:

1. Si sospende fintantoché non acquisisce l’accesso esclusivo alla risorsa;
2. Utilizza la risorsa;
3. Segnala che la risorsa è libera.

Dato un insieme di *n* processi *P1*, …, *Pn* e *m* tipi di risorse condivise *R1*, …, *Rm*, chiamiamo   
**grafo possesso-attesa** un grafo orientato che ha:

* Come nodi:
  + Un nodo circolare per ogni processo *Pi*;
  + Un nodo rettangolare per ogni tipo di risorsa *Rj*;
* Come archi:
  + Un arco orientato da *Pi* a *Rj* se *Pi* è sospeso fintantoché non acquisisce   
    l’accesso esclusivo a una risorsa di tipo *Rj*;
  + Un arco orientato da *Ri* a *Pj* se una risorsa di tipo *Ri* è occupata da *Pj*.

A questo punto, **C.N.** affinché si abbia un deadlock è che si verifichino tutte le seguenti ipotesi:

* *Mutua esclusione*: ogni risorsa può essere acceduta solo in mutua esclusione;
* *Possesso e attesa*: ogni processo non libera le risorse da lui occupate   
  quando si sospende fintantoché non acquisisce l’accesso esclusivo ad un’altra risorsa;
* *Mancanza di diritto di revoca*: non è possibile revocare a un processo una risorsa   
  da lui occupata;
* *Attesa circolare*: il grafo possesso-attesa presenta un ciclo.

Questa diventa una **C.N.S.** nel caso in cui ogni tipo di risorsa fosse disponibile in un’unica istanza.

**PREVENZIONE STATICA DEL DEADLOCK** *14-11-24*

La **prevenzione statica del deadlock** consiste nell’assicurare, all’atto della scrittura dei programmi, che almeno una delle quattro ipotesi che costituiscono la C.N/C.N.S. affinché si abbia un deadlock   
non sia mai verificata.

Tralasciando l’ipotesi di *mutua esclusione*, che dovrà essere sempre verificata in quanto le risorse   
sono condivise, è possibile intervenire su ognuna delle altre tre ipotesi in questo modo:

* *Possesso e attesa:* si può imporre che ogni processo all’inizio richieda di acquisire   
  l’accesso esclusivo a tutte le risorse di cui avrà bisogno.
* *Mancanza di diritto di revoca:* si può imporre che, se un processo richiede di acquisire  
  l’accesso esclusivo a una risorsa che al momento è occupata, questo, prima di sospendersi, liberi tutte le risorse da lui occupate. Il processo verrà poi risvegliato quando avrà acquisito   
  l’accesso esclusivo alla risorsa per cui l’ha richiesto e a quelle che occupava   
  prima di sospendersi;
* *Attesa circolare*: si può imporre che, se un processo occupa una risorsa di tipo *Rk*,   
  possa richiedere di acquisire l’accesso esclusivo solo a risorse di tipo *Rk+i*, con *i* > *0*.

**PREVENZIONE DINAMICA DEL DEADLOCK**

Supponiamo che siano verificate le prime tre ipotesi che costituiscono la C.N/C.N.S. affinché si abbia un deadlock e di conoscere a priori il numero massimo di risorse di ogni tipo di cui ogni processo   
può richiedere l’accesso esclusivo.

Diremo che un insieme di processi in una determinata situazione è:

* In uno **stato safe** **rispetto al deadlock** se, a partire da questa situazione, esiste   
  almeno una sequenza di esecuzione <Ph, Pk, … > dei processi tale che, per ogni processo,   
  le richieste di accesso esclusivo alle risorse che tale processo può ancora effettuare   
  possono essere soddisfatte dalle risorse attualmente libere o dalle risorse occupate   
  dai processi che lo precedono nella sequenza. Un insieme di processi che si trova in questo stato, dunque, può ancora terminare in un tempo finito.
* In uno **stato unsafe** **rispetto al deadlock** altrimenti. Un insieme di processi che si trova   
  in questo stato, dunque, potrebbe non terminare più in un tempo finito, finendo così   
  in deadlock.

Detto ciò, la **prevenzione dinamica del deadlock** consiste nell’assicurare, all’atto di esecuzione   
dei programmi, che un insieme di processi non entri mai in uno stato unsafe rispetto al deadlock.  
In particolare, esistono degli algoritmi in grado di rilevare se, in una determinata situazione,   
un insieme di processi è in uno stato safe rispetto al deadlock o meno. A questo punto,   
la prevenzione dinamica del deadlock consiste nell’eseguire uno di questi algoritmi ogni volta   
che un processo richiede di acquisire l’accesso esclusivo a una risorsa e questa è libera,   
considerando la situazione che si verrebbe a creare se il processo occupasse effettivamente la risorsa:

* Se in questa situazione l’algoritmo rivela che l’insieme di processi è in uno stato safe,   
  allora il processo occuperà effettivamente la risorsa;
* Se in questa situazione l’algoritmo rivela che l’insieme di processi è in uno stato unsafe,   
  allora il processo verrà sospeso.

Nel caso in cui ogni tipo di risorsa fosse disponibile in un’unica istanza, un algoritmo usato per fare prevenzione dinamica del deadlock è l’analisi del grafo possesso-attesa:

* Se il grafo possesso-attesa non presenta alcun ciclo, allora l’insieme complessivo di processi   
  è in uno stato safe rispetto al deadlock;
* Altrimenti, l’insieme dei processi coinvolti nel ciclo è in uno stato unsafe rispetto al deadlock, e in particolare, essendo soddisfatta la C.N.S. affinché si abbia un deadlock,   
  è proprio in deadlock.

Nel caso in cui almeno un tipo di risorsa fosse disponibile in molteplici istanze, un algoritmo usato   
per fare prevenzione dinamica del deadlock è il cosiddetto **algoritmo del banchiere**.

Innanzitutto, dato un insieme di *n* processi *P1*, …, *Pn* e *m* tipi di risorse condivise *R1*, …, *Rm,* definiamo   
le seguenti strutture dati:

* Un array *available* di dimensione *m*, dove *available*[*j*] contiene il numero di risorse di tipo *Rj*attualmente libere;
* Una matrice *allocation* di dimensione *n* x *m*, dove *allocation*[*i*][*j*] contiene il numero di risorse   
  di tipo *Rj*attualmente occupate dal processo *Pi*;
* Una matrice *need* di dimensione *n* x *m*, dove *need*[*i*][*j*] contiene il numero di risorse   
  di tipo *Rj*di cui il processo *Pi* può ancora richiedere l’accesso esclusivo.

A questo punto, l’algoritmo del banchiere è il seguente:

1. Siano *finish* e *work* due array di dimensione rispettivamente *n* e *m*. All’inizio:
   * *finish*[*i*] = false
   * *work*[*j*] = *available*[*j*]
2. Fintantoché esiste un *i* tale che siano soddisfatte tutte le seguenti condizioni:
   * *finish*[*i*] == false
   * *need*[*i*][*j*] ≤ *work*[*j*]

Si pone:

* + *finish*[*i*] = true
  + *work*[*j*] = *work*[*j*] + *allocation*[*i*][*j*]

1. Se *finish*[*i*] == true , allora l’insieme di processi è in uno stato safe   
   rispetto al deadlock. Altrimenti, l’insieme di processi è in uno stato unsafe rispetto al deadlock.

Per vedere se, in una determinata situazione, l’insieme di processi è in uno stato safe rispetto al deadlock, bisogna vedere se,   
a partire da questa situazione, esiste almeno una sequenza di esecuzione <*Ph*, *Pk*, … > dei processi tale che, per ogni processo,   
le richieste di accesso esclusivo alle risorse che tale processo può ancora effettuare possono essere soddisfatte   
dalle risorse attualmente libere o dalle risorse occupate dai processi che lo precedono nella sequenza.   
Per vedere se esiste una sequenza del genere, allora, simulo una possibile esecuzione dei processi. Innanzitutto, creo due array:

* *finish*, di dimensione *n*, dove *finish*[*i*] == true se il processo *Pi* è terminato.
* *work*, di dimensione *m*, che sarà l’evoluzione di *available* durante la simulazione dell’esecuzione dei processi.

A questo punto, fintantoché esiste almeno un processo *Pi* non terminato (*finish*[*i*] == false) tale che, con le risorse attualmente libere, possono essere soddisfatte tutte le richieste di accesso esclusivo alle risorse che tale processo può ancora effettuare (*need*[*i*][*j*] ≤ *work*[*j*]   
), lo faccio andare in esecuzione, facendogli così occupare tali risorse. Questo processo prima o poi terminerà   
(*finish*[*i*] = true), liberando tutte le risorse occupate (*work*[*j*] = *work*[*j*] + *allocation*[*i*][*j*] ).

Se in questo modo riesco a far terminare tutti i processi (*finish*[*i*] == true ) allora l’insieme di processi è in uno stato safe rispetto al deadlock, e la sequenza di esecuzione che lo certifica è quella ottenuta simulando l’esecuzione dei processi.   
Altrimenti, non avendo trovato una sequenza di esecuzione che certifichi che l’insieme di processi è in uno stato safe rispetto al deadlock, l’insieme di processi è in uno stato unsafe rispetto al deadlock.

**TECNICA DI INDIVIDUAZIONE DEL DEADLOCK E SUCCESSIVO RIPRISTINO**

Supponiamo che siano verificate le prime tre ipotesi che costituiscono la C.N/C.N.S. affinché si abbia un deadlock.

La **tecnica di individuazione del deadlock e successivo ripristino** consiste nel permettere   
che i deadlock si presentino e verificare periodicamente se un insieme di processi è in deadlock,   
eliminando il deadlock in caso affermativo.

Nel caso in cui ogni tipo di risorsa fosse disponibile in un’unica istanza, un algoritmo usato per fare individuazione del deadlock è la costruzione e analisi del **grafo delle attese**, e cioè un grafo che ha:

* Un nodo per ogni processo *Pi*;
* Un arco orientato da *Pi* a *Pj* se *Pi* è sospeso fintantoché non acquisisce l’accesso esclusivo   
  a una risorsa occupata da *Pj*.

A questo punto:

* Se il grafo delle attese non presenta alcun ciclo, allora l’insieme complessivo di processi   
  non è in deadlock;
* Altrimenti, l’insieme dei processi coinvolti nel ciclo è in deadlock.

Nel caso in cui almeno un tipo di risorsa fosse disponibile in molteplici istanze, un algoritmo usato   
per fare individuazione del deadlock è lo ***pseudo*-algoritmo del banchiere**.

Innanzitutto, dato un insieme di *n* processi *P1*, …, *Pn* e *m* tipi di risorse condivise *R1*, …, *Rm*, definiamo   
le seguenti strutture dati:

* Un array *available* di dimensione *m*, dove *available*[*j*] contiene il numero di risorse di tipo *Rj*attualmente libere;
* Una matrice *allocation* di dimensione *n* x *m*, dove *allocation*[*i*][*j*] contiene il numero di risorse   
  di tipo *Rj*attualmente occupate dal processo *Pi*;
* Una matrice *request* di dimensione *n* x *m*, dove *request*[*i*][*j*] contiene il numero di risorse   
  di tipo *Rj*di cui il processo *Pi* sta attualmente richiedendo l’accesso esclusivo.

A questo punto, lo *pseudo*-algoritmo del banchiere è il seguente:

1. Siano *finish* e *work* due array di dimensione rispettivamente *n* e *m*. All’inizio:
   * *finish*[*i*] =
   * *work*[*j*] = *available*[*j*]
2. Fintantoché esiste un *i* tale che siano soddisfatte tutte le seguenti condizioni:
   * *finish*[*i*] == false
   * *request*[*i*][*j*] ≤ *work*[*j*]

Si pone:

* + *finish*[*i*] = true
  + *work*[*j*] = *work*[*j*] + *allocation*[*i*][*j*]

1. Se *finish*[*i*] == true , allora l’insieme complessivo di processi non è in deadlock.   
   Altrimenti, l’insieme di processi *Pi* per cui *finish*[*i*] == false è in deadlock.

Per vedere se, in una determinata situazione, l’insieme di processi non è in deadlock, bisogna vedere se,   
a partire da questa situazione, esiste almeno una sequenza di esecuzione <*Ph*, *Pk*, … > dei processi tale che, per ogni processo,   
le richieste di accesso esclusivo alle risorse che tale processo sta attualmente effettuando possono essere soddisfatte   
dalle risorse attualmente libere o dalle risorse occupate dai processi che lo precedono nella sequenza. Per fare ciò, allora,   
simulo una possibile esecuzione dei processi. Innanzitutto, creo due array:

* *finish*, di dimensione *n*, dove *finish*[*i*] == true se il processo *Pi* non occupa alcuna risorsa;
* *work*, di dimensione *m*, che sarà l’evoluzione di *available* durante la simulazione dell’esecuzione dei processi.

A questo punto, fintantoché esiste almeno un processo *Pi* che occupa delle risorse (*finish*[*i*] == false) tale che, con le risorse   
attualmente libere, possono essere soddisfatte tutte le richieste di accesso esclusivo alle risorse che tale processo   
sta attualmente effettuando (*request*[*i*][*j*] ≤ *work*[*j*] ), lo faccio andare in esecuzione, facendogli così occupare tali risorse. Ipotizzando che, una volta occupate queste risorse, il processo non ne chiederà altre, questo processo prima o poi terminerà,   
liberando tutte le risorse occupate (*finish*[*i*] = true e *work*[*j*] = *work*[*j*] + *allocation*[*i*][*j*] ).

Se in questo modo riesco a far liberare le risorse occupate da ogni processo (*finish*[*i*] == true ) allora l’insieme complessivo   
di processi non è in deadlock. Altrimenti, l’insieme di processi a cui non sono riuscito a far liberare le risorse da loro occupate   
è in deadlock.

**NB:** in entrambi gli algoritmi, dunque, viene mantenuto:

* Il numero di risorse attualmente libere;
* Il numero di risorse di ogni tipo attualmente occupate da ogni processo.

In più:

* Nell’algoritmo del banchiere viene mantenuto il numero di risorse di ogni tipo di cui ogni processo   
  può ancora richiedere l’accesso esclusivo;
* Nello pseudo-algoritmo del banchiere viene mantenuto il numero di risorse di ogni tipo di cui ogni processo   
  sta attualmente richiedendo l’accesso esclusivo.

Una volta individuato un deadlock, è possibile eliminarlo abortendo alcuni dei processi coinvolti.   
La scelta di quali precisi processi abortire può essere basata sulla priorità, sul numero di risorse attualmente occupate…

**MODELLO DI INTERAZIONE A MEMORIA LOCALE: SCAMBIO DI MESSAGGI**

Nel modello di interazione a memoria locale, ogni processo ha solo una propria memoria privata.Serve dunque introdurre un qualche meccanismo che permetta ad un processo di   
**scambiare messaggi** con altri processi.

**SEND() E RECEIVE()**

Alcuni sistemi operativi permettono a un processo di poter scambiare messaggi con un altro   
attraverso due primitive:

* **send(*destinazione*, *messaggio*)**: permette al processo che la invoca di inviare il *messaggio* indicato alla *destinazione* indicata;
* **receive(*origine*, *messaggio*)**: permette al processo che la invoca di ricevere un messaggio proveniente o dall’*origine* indicata, o da una qualunque origine, e in quest’ultimo caso   
  l’origine del messaggio ricevuto verrà salvata nella variabile puntata da *origine*.   
  Il messaggio ricevuto verrà salvato nella variabile puntata da *messaggio*.

Queste primitive possono essere:

* bloccanti, e cioè il processo che le invoca si blocca fintantoché non verrà inviato/ricevuto   
  tutto il messaggio;
* non bloccanti, e cioè il processo che le invoca continua la sua esecuzione, anche se non è stato inviato/ricevuto tutto il messaggio.

**GESTIONE DELLA MEMORIA**

**MEMORIA VIRTUALE**

Ogni processo possiede una propria **memoria *virtuale***, e cioè ha l’illusione di avere una memoria   
tutta per sé contenente, a partire dall’indirizzo 0 e ad indirizzi contigui, tutta e sola la memoria   
del processo. La memoria virtuale del processo verrà poi caricata nell’unica memoria esistente,   
detta per contrapposizione memoria *fisica*.

**PARAMETRI DI GESTIONE DELLA MEMORIA**

**RILOCAZIONE**

Quando un processo vorrà accedere alla memoria, penserà di accedere alla propria memoria virtuale,   
per cui genererà indirizzi della propria memoria virtuale, detti *indirizzi virtuali*.   
La vera memoria, tuttavia, è la memoria fisica, per cui, affinché un processo acceda effettivamente   
alla memoria, sarà necessaria la cosiddetta **rilocazione**, e cioè la traduzione degli indirizzi virtuali   
nei corrispondenti indirizzi della memoria fisica, detti *indirizzi fisici*.

La rilocazione può essere di due tipi:

* Rilocazione *statica*;
* Rilocazione *dinamica*.

La **rilocazione statica** prevede che, prima caricare per la prima volta la memoria virtuale di un processo   
nella memoria fisica, vengono cambiati tutti gli indirizzi virtuali presenti nel codice del processo   
con quelli che saranno i corrispondenti indirizzi fisici.

Un limite molto forte della rilocazione statica è che, una volta caricata la memoria virtuale   
di un processo in una porzione della memoria fisica, non è più possibile caricarla in una porzione   
della memoria fisica diversa. Supponiamo infatti che, ad un certo punto, venga fatto   
lo swap-out di un processo. Quando bisognerà fare lo swap-in, gli indirizzi presenti   
nella memoria virtuale del processo sono ancora gli indirizzi fisici originari, per cui,   
se si volesse caricare il processo in una porzione della memoria fisica diversa da quella originaria, bisognerebbe prima cambiare questi indirizzi in quelli che saranno i nuovi indirizzi fisici. Tuttavia,   
se questo è possibile con gli indirizzi presenti nel codice del processo, questo non è possibile invece con eventuali indirizzi presenti nello stack (*es.* indirizzi di ritorno), in quanto non è possibile distinguere nello stack cosa è un indirizzo da cosa è un normale dato.

La **rilocazione dinamica** prevede che gli indirizzi virtuali generati da un processo vengano tradotti   
nei corrispondenti indirizzi fisici a tempo di esecuzione. A tale scopo si sfrutta   
una componente hardware, detta *MMU*, che traduce ogni indirizzo virtuale generato dalla CPU   
nel corrispondente indirizzo fisico.

**ORGANIZZAZIONE DELLA MEMORIA VIRTUALE**

La memoria virtuale di ogni processo può essere organizzata come:

* Memoria virtuale *unica*;
* Memoria virtuale *segmentata*.

La **memoria virtuale segmentata** è costituita da più sotto-memorie virtuali separate tra loro,   
dette *segmenti*, di dimensione variabile.

Ogni segmento di un processo è numerato a partire da 0, e ogni locazione nella memoria   
virtuale segmentata è identificata da un indirizzo virtuale costituito da una coppia <*sg*, *sc*>, dove:

* *sg* è il numero del segmento in cui si trova tale locazione;
* *sc* è l’indirizzo virtuale di tale locazione all’interno del segmento.

Ciascuno di questi segmenti verrà poi caricato in memoria fisica indipendentemente dagli altri.

**ALLOCAZIONE DELLA MEMORIA VIRTUALE NELLA MEMORIA FISICA**

L’allocazione della memoria virtuale di ogni processo nella memoria fisica può essere:

* *Contigua*;
* *Non contigua*.

Nel caso di allocazione contigua può verificarsi la cosiddetta *frammentazione esterna*, e cioè   
la situazione in cui lo spazio libero presente in memoria fisica è frammentato in porzioni non contigue, presenti tra le memorie virtuali dei vari processi. Un modo per risolvere questo problema può essere effettuare periodicamente il cosiddetto *compattamento*, e cioè effettuare lo swap-out di tutti i processi e poi lo swap-in, caricando la memoria virtuale di ogni processo nella memoria fisica in maniera tale   
da avere un unico spazio libero presente in memoria fisica.

L’**allocazione non contigua** prevede che la memoria virtuale venga divisa in porzioni contigue   
aventi stessa dimensione, dette *pagine,* numerate a partire da 0*.* Similmente, la memoria fisica   
viene divisa in porzioni contigue aventi la stessa dimensione delle pagine, dette *frame*,   
numerati a partire da 0. Ciascuna pagina verrà poi caricata in un certo frame, e si può avere   
che due pagine contigue vengano caricate in due frame non contigui.

Supponiamo che la dimensione delle pagine della memoria virtuale di ogni processo   
(e quindi, la dimensione dei frame) sia di 2*d* byte. Ogni locazione della memoria virtuale di un processo   
è identificata da un indirizzo virtuale *x*, e si può verificare che:

* I *d* bit meno significativi indicano l’offset *of* all’interno della pagina a cui la locazione appartiene;
* I restanti bit più significativi indicano il numero *pg* della pagina a cui la locazione appartiene.

Similmente, ogni locazione della memoria fisica sarà identificata da un indirizzo fisico *y*,   
e si può verificare che:

* I *d* bit meno significativi indicano l’offset *of* all’interno del frame a cui la locazione appartiene;
* I restanti bit più significativi indicano il numero *pf* del frame a cui la locazione appartiene.

**CARICAMENTO DELLA MEMORIA VIRTUALE NELLA MEMORIA FISICA**

La memoria virtuale di ogni processo può essere caricata nella memoria fisica:

* Tutta insieme;
* A *domanda*.

Nel caso di caricamento tutto insieme, al diagramma degli stati di un processo vanno aggiunti gli stati *pronto swapped* e *bloccato swapped*, che corrispondono rispettivamente agli stati *pronto* e *bloccato* nel caso in cui la memoria virtuale del processo non fosse presente in memoria fisica:

**

Un processo passa allo stato bloccato swapped quando, mentre è nello stato bloccato,   
subisce swap-out. Il processo uscirà poi da questo stato solo quando si sarà verificato l’evento   
per cui si è bloccato, passando così allo stato pronto swapped.

Un processo passa allo stato pronto swapped quando:

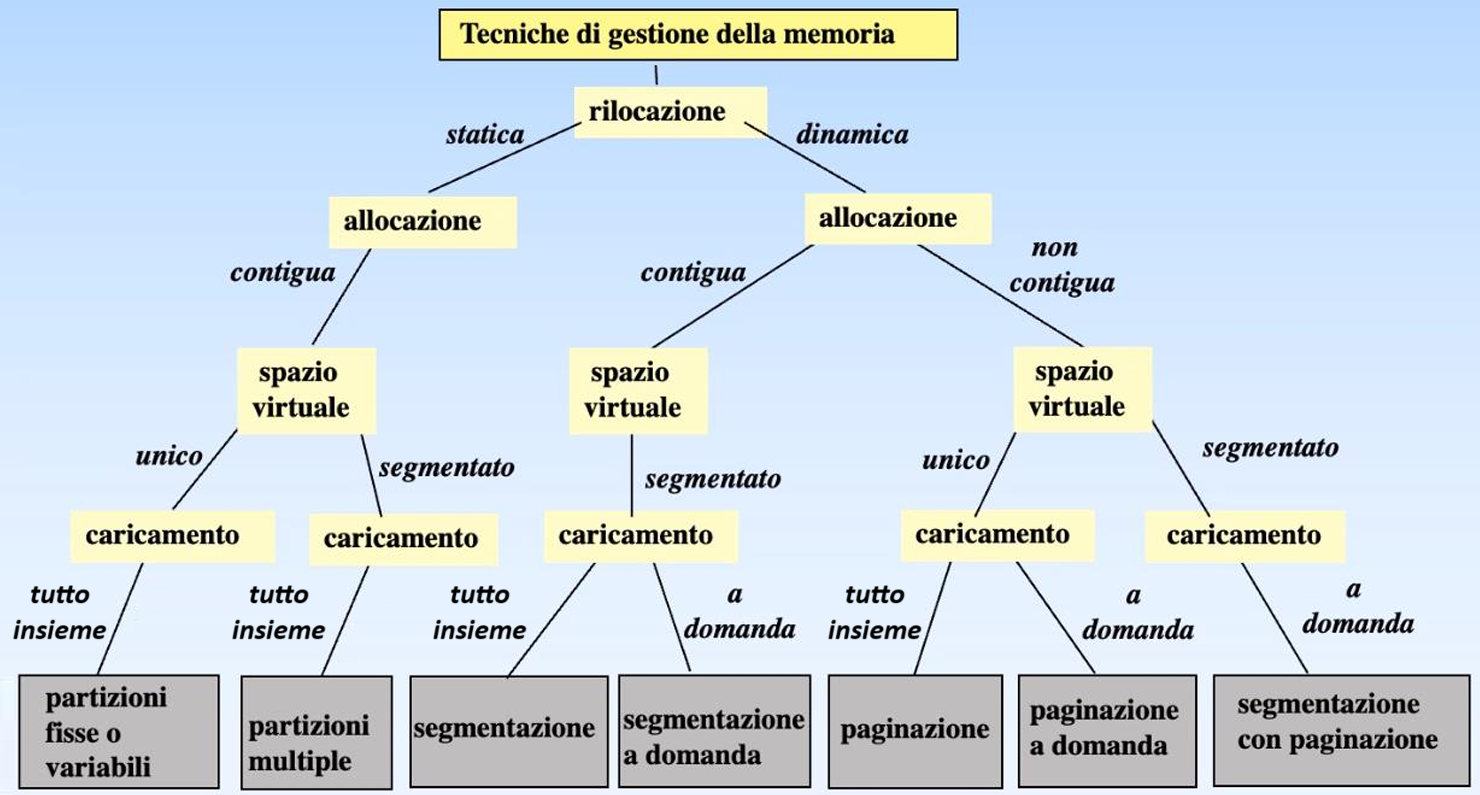
* Mentre è nello stato nuovo, viene attivato, in quanto la memoria virtuale del processo   
  non viene subito caricata nella memoria fisica;
* Mentre è nello stato pronto, subisce swap-out;
* Mentre è stato bloccato swapped, si verifica l’evento per cui il processo si è bloccato.

Il processo uscirà poi da questo stato solo quando subirà swap-in, passando così allo stato pronto.

Il **caricamento a domanda** prevede che ogni parte della memoria virtuale di un processo   
possa essere caricata in memoria fisica solo quando sarà richiesta effettivamente dal processo.

**TECNICHE DI GESTIONE DELLA MEMORIA**

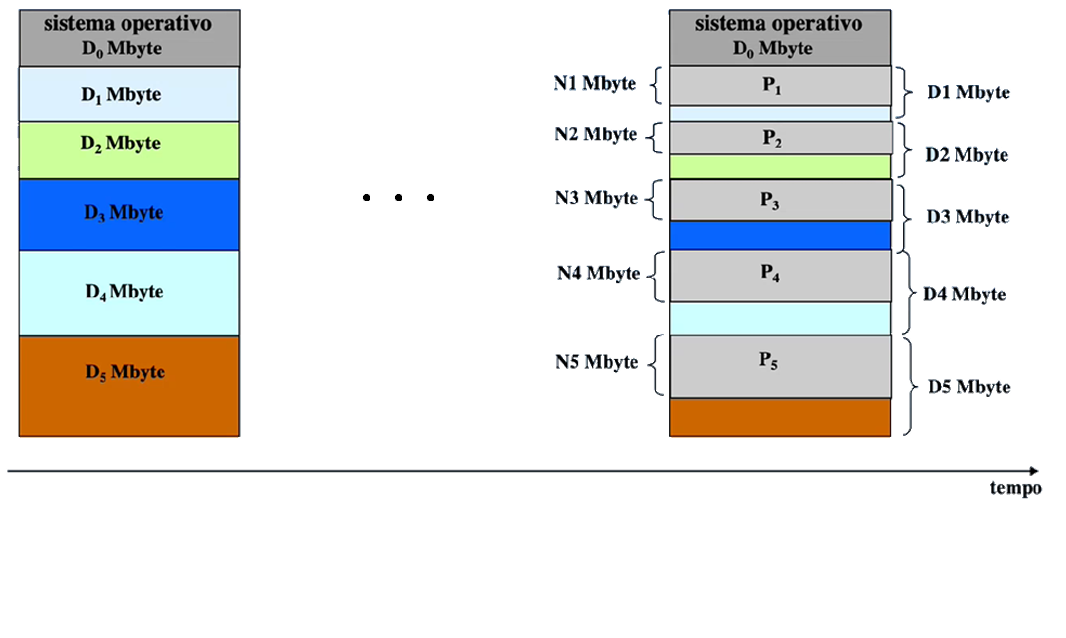
In base al valore dei parametri di gestione della memoria visti in precedenza, possiamo distinguere   
le seguenti tecniche di gestione della memoria:

**

**MEMORIA PARTIZIONATA**

**PARTIZIONI FISSE**

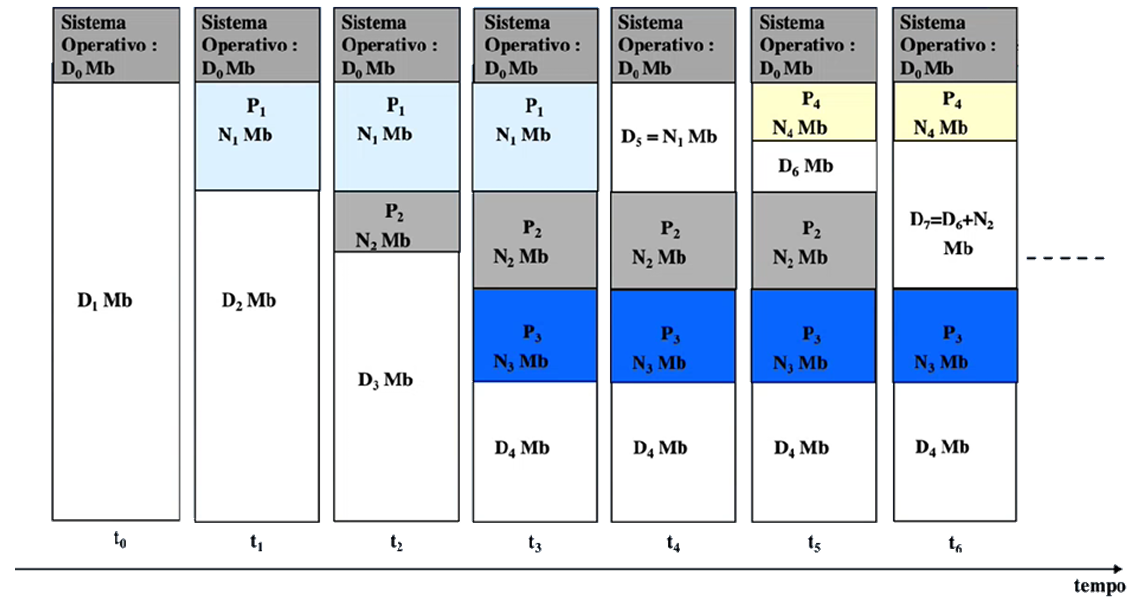
Nella **tecnica a partizioni fisse**, la memoria fisica viene divisa in (*n* + 1) partizioni di dimensione fissata. La prima di queste conterrà il sistema operativo, mentre in ognuna delle altre *n* può essere caricata   
la memoria virtuale di un solo processo per volta.

**

Un problema che può verificarsi quando la memoria virtuale di ogni processo viene caricata in porzioni   
di memoria fisica di dimensione fissata è la cosiddetta *frammentazione interna*, e cioè la situazione   
in cui, una volta effettuato un caricamento in una porzione libera della memoria fisica,   
se ciò che si è caricato è più piccolo della porzione, si ha dello spazio libero rimanente nella porzione,   
non utilizzabile.

**PARTIZIONI VARIABILI**

Nella **tecnica a partizioni variabili**, la memoria fisica è inizialmente costituita da un’unica   
partizione libera, di dimensione pari a quella di tutta la memoria fisica meno la dimensione   
dell’area occupata dal sistema operativo. Ogni volta che bisognerà caricare la memoria virtuale   
di un processo in memoria fisica, da una partizione libera verrà creata una nuova partizione   
di dimensione pari alla dimensione della memoria virtuale del processo, e in cui tale memoria virtuale   
verrà caricata.

**

In memoria fisica è salvata una *lista delle partizioni libere*, in cui ogni elemento contiene informazioni su una diversa partizione libera, come il suo indirizzo fisico iniziale e la sua dimensione.   
A questo punto, per decidere in quale partizione libera caricare la memoria virtuale di un processo, bisogna scorrere questa lista, scegliendo una partizione libera sufficientemente grande a contenere tale memoria virtuale. In particolare, esistono diversi criteri per scegliere in quale partizione sufficientemente grande caricare la memoria virtuale di un processo:

* Best-fit: tra tutte le partizioni sufficientemente grandi a contenere la memoria virtuale,   
  viene scelta quella più piccola.
* Worst-fit: tra tutte le partizioni sufficientemente grandi a contenere la memoria virtuale,  
  viene scelta quella più grande.
* First-fit: tra tutte le partizioni sufficientemente grandi a contenere la memoria virtuale,  
  viene scelta quella con indirizzo fisico più basso.

A seconda del criterio usato, per fare in modo che la lista delle partizioni libere venga scorsa   
il meno possibile quando bisogna trovare la partizione in cui dover caricare la memoria virtuale   
di un processo, conviene che questa lista sia ordinata in maniera opportuna:

* Usando il criterio best-fit, conviene ordinare la lista per dimensione delle partizioni crescente;
* Usando il criterio worst-fit, conviene ordinare la lista per dimensione delle partizioni decrescente;
* Usando il criterio first-fit, conviene ordinare la lista per indirizzo fisico della partizione crescente.

Essendo la tecnica a partizioni variabili una tecnica con allocazione contigua, può presentarsi   
il problema della frammentazione esterna. Adottando il criterio best-fit, questo problema   
è ancora più rognoso, in quanto, caricando ogni volta la memoria virtuale di un processo   
nella più piccola partizione libera sufficientemente grande, si andranno ben presto a formare frammenti liberi talmente piccoli che non è possibile caricare in essi la memoria virtuale   
di alcun processo.

Inoltre, se a seguito dello swap-out o della terminazione di un processo si formassero   
due o più partizioni libere contigue, sarebbe conveniente fondere queste partizioni libere  
in un’unica partizione libera. Per come verrebbe ordinata la lista delle partizioni libere:

* Adottando il criterio best-fit o worst-fit, il processo di fusione sarebbe molto oneroso;
* Adottando il criterio first-fit, il processo di fusione sarebbe non troppo oneroso, e questo   
  è il motivo per cui il criterio first-fit è quello generalmente più adottato.

**PARTIZIONI MULTIPLE**

La **tecnica a partizioni multiple** coincide con la tecnica a partizioni variabili con la differenza che, quando bisognerà caricare la memoria virtuale di un processo in memoria fisica,   
dalle partizioni libere verranno create tante partizioni quanti sono i segmenti   
di cui è costituita la memoria virtuale del processo, ognuna di dimensione pari alla dimensione   
del corrispondente segmento, e in cui tale segmento verrà caricato.

**MEMORIA SEGMENTATA**

**SEGMENTAZIONE**

Nella **tecnica della segmentazione**, essendo una tecnica a memoria virtuale segmentata,   
la memoria virtuale di ogni processo è divisa in uno o più segmenti, numerati a partire da 0,   
ciascuno dei quali verrà caricato in memoria fisica indipendentemente dagli altri.   
Per ogni processo è salvata in memoria fisica una cosiddetta *tabella dei segmenti*, e cioè   
una tabella dove la riga *i*-esima è relativa all’*i*-esimo segmento del processo, ed è costituita   
dai seguenti campi:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| base | limite | bit di controllo | | | | |
| *indirizzo fisico I del segmento* | *dimensione D del segmento* | *R* | *W* |  |  |  |

* Un campo *base*, contenente l’indirizzo fisico *I* del segmento;
* Un campo *limite*, contenente la dimensione *D* in byte del segmento.
* Alcuni bit di controllo:

Read

* + Un bit *R*, che vale 1 se il segmento è accessibile in lettura;

Write

* + Un bit *W*, che vale 1 se il segmento è accessibile in scrittura.

Inoltre, la CPU è dotata di due registri:

Segment Table Base Register

* STBR, contenente l’indirizzo fisico della tabella dei segmenti del processo in esecuzione;

Segment Table Limit Register

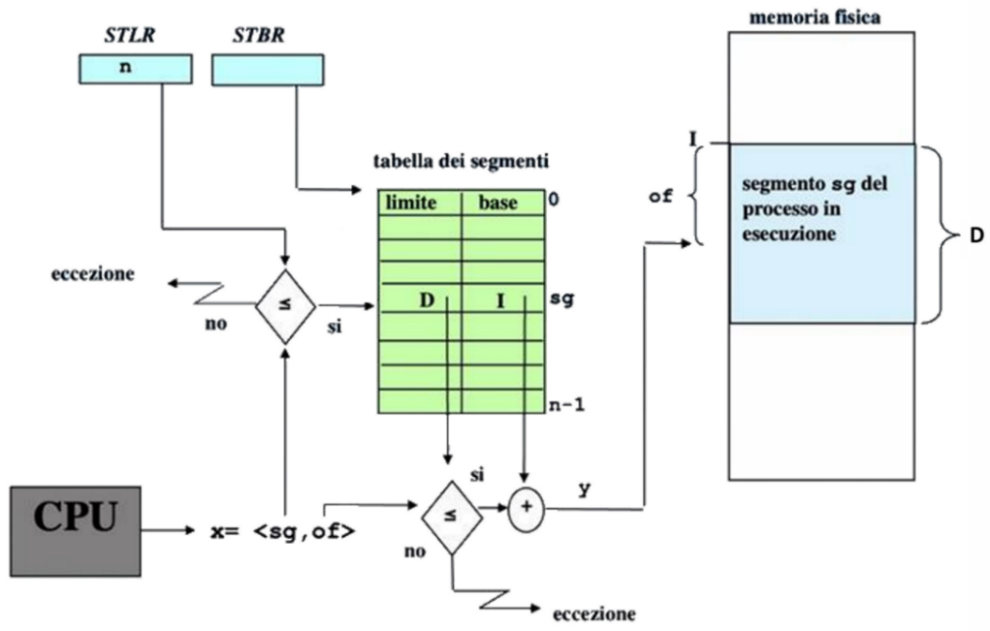
* STLR, contenente il numero di segmenti del processo in esecuzione.

Memoria virtuale segmentata

Supponiamo che la CPU, eseguendo un processo, generi un indirizzo virtuale *x =* <*sg*, *sc*>.

La traduzione effettuata dall’MMU di questo indirizzo virtuale *x* nel corrispondente indirizzo fisico *y* avviene in questo modo:

1. *sg* viene confrontato con STLR, e se risulta *sg* ≥ STLR, poiché il processo sta tentando   
   di accedere ad un segmento inesistente, viene sollevata un’eccezione.
2. Vengono letti i valori *I*, *D*, *R* e *W* della riga *sg*-esima della tabella dei segmenti puntata da STBR:
   1. se risulta *sc* ≥ *D*, poiché il processo sta tentando di accedere ad una locazione   
      non appartenente al segmento *sg*-esimo, viene sollevata un’eccezione.
   2. se risulta *R* = 0 o *W* = 0, e il processo vuole accedere all’indirizzo virtuale *x*   
      rispettivamente in lettura o in scrittura, poiché sta tentando di accedere al segmento *sg*-esimo in un modo che non è permesso, viene sollevata un’eccezione.
3. L’indirizzo virtuale *x* viene tradotto nel corrispondente indirizzo fisico *y* = *I* + *sc*.

**

Questo meccanismo di traduzione degli indirizzi virtuali presenta vari vantaggi:

1. **Protezione:** come si è visto, sono presenti 3 diversi controlli per verificare se un processo   
   può accedere o meno ad un certo indirizzo virtuale;
2. **Condivisione:** due o più processi possono condividere uno stesso segmento   
   (ad esempio, se più processi eseguono lo stesso codice, possono condividere il segmento contenente il codice), ed è possibile fare in modo che in memoria fisica venga caricata   
   una sola copia di tale segmento.

**SEGMENTAZIONE A DOMANDA**

La **tecnica della segmentazione a domanda** coincide con la tecnica della segmentazione,   
con la differenza che, essendo una tecnica con caricamento a domanda, ogni segmento di un processo verrà caricato in memoria fisica solo quando sarà richiesto effettivamente dal processo.

Per ogni segmento di un processo, la riga corrispondente nella tabella dei segmenti del processo   
viene modificata in questo modo:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| base | limite | bit di controllo | | | | |
| *indirizzo fisico I del segmento, se P = 1*  *indirizzo I del segmento in memoria secondaria, se P = 0* | *dimensione D del segmento* | *R* | *W* | *U* | *M* | *P* |

* Si aggiunge un bit *U*, che vale 1 se, dall’ultima volta che è stato caricato in memoria fisica,   
  il segmento è stato mai acceduto;

Modifica

Uso

* Si aggiunge un bit *M*, che vale 1 se, dall’ultima volta che è stato caricato in memoria fisica,   
  il segmento è stato mai modificato;

Presenza

* Si aggiunge un bit *P*, che vale 1 se il segmento è presente in memoria fisica;
* Il campo *base* contiene:
  + L’indirizzo fisico *I* del segmento, se *P* = 1;
  + L’indirizzo *I* del segmento nella memoria secondaria, se *P* = 0.

A questo punto, quando un processo vuole accedere a un certo segmento *sg*, se la riga *sg*-esima   
nella tabella dei segmenti del processo presenta *P* = 0, viene sollevata un’eccezione   
di *segment-fault*, che la routine di gestione dell’eccezione gestisce in questo modo:

1. Viene cercata una porzione libera di memoria fisica abbastanza grande a contenere   
   il segmento *sg*. Sia *I’* l’indirizzo fisico della porzione libera di memoria fisica trovata;
2. Il segmento *sg* viene caricato in memoria fisica all’indirizzo fisico *I*’, copiandolo   
   dalla memoria secondaria a partire dall’indirizzo *I* contenuto nel campo *base*   
   della riga *sg*-esima della tabella dei segmenti del processo;
3. Viene aggiornata la riga *sg*-esima della tabella dei segmenti del processo, impostando:
   * Il campo *base* a *I*’;
   * Il bit *P* a 1.
4. Poiché l’eccezione di segment-fault ha causato l’abort dell’istruzione che l’ha generata,   
   viene rieseguita tale istruzione.

**MEMORIA PAGINATA** *28-11-24*

**PAGINAZIONE**

Nella **tecnica della paginazione**, essendo una tecnica ad allocazione non contigua,   
la memoria virtuale di ogni processo viene divisa in *pagine*, numerate a partire da 0.   
Similmente, la memoria fisica viene divisa in frame, numerati a partire da 0, della stessa dimensione   
delle pagine. Ciascuna pagina verrà poi caricata in un certo frame.

Per ogni processo è salvata in memoria fisica una cosiddetta *tabella delle pagine*, e cioè una tabella dove la riga *i*-esima è relativa all’*i*-esima pagina della memoria virtuale del processo,   
ed è costituita dai seguenti campi:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| campo frame | bit di controllo | | | | |
| *numero pf del frame in cui  è stata caricata la pagina* | *R* | *W* |  |  |  |

* Un *campo frame*, contenente il numero *pf* del frame in cui è stata caricata la pagina;
* Alcuni bit di controllo:
  + Un bit *R*, che vale 1 se la pagina è accessibile in lettura;
  + Un bit *W*, che vale 1 se la pagina è accessibile in scrittura.

In memoria fisica è poi salvata una cosiddetta *tabella dei frame*, e cioè una tabella dove la riga *i*-esima   
è relativa all’*i*-esimo frame, ed è costituita principalmente dal seguente campo:

|  |
| --- |
| *indicazione che il frame è libero, se il frame è libero*  *numero pg della pagina in esso presente e un’indicazione  sul processo cui tale pagina appartiene, se il frame è occupato* |

* Il campo:
  + Contiene un’indicazione che tale frame è libero, se il frame è libero;
  + Contiene il numero *pg* della pagina in esso presente e un’indicazione sul processo   
    cui tale pagina appartiene, se il frame è occupato.

Inoltre, la CPU è dotata di due registri:

Registro Puntatore Tabella Pagine

* RPTP, contenente l’indirizzo fisico della tabella delle pagine del processo in esecuzione;

Registro Limite Tabella Pagine

* RLTP, contenente il numero di pagine della memoria virtuale del processo in esecuzione.

Memoria virtuale paginata

Supponiamo che la CPU, eseguendo un processo, generi un indirizzo virtuale *x* = *pg* | *of*.

La traduzione effettuata dall’MMU di questo indirizzo virtuale *x* nel corrispondente indirizzo fisico *y* avviene in questo modo:

1. *pg* viene confrontato con RLTP, e se risulta *pg* ≥ RLTP, poiché il processo sta tentando   
   di accedere ad una pagina inesistente, viene sollevata un’eccezione.
2. Vengono letti i valori *pf*, *R* e *W* della riga *pg*-esima della tabella delle pagine puntata da RPTP,  
   e se risulta *R* = 0 o *W* = 0 e il processo vuole accedere all’indirizzo virtuale *x*   
   rispettivamente in lettura o in scrittura, poiché sta tentando di accedere alla pagina *pg*-esima   
   in un modo che non è permesso, viene sollevata un’eccezione.
3. L’indirizzo virtuale *x* viene tradotto nel corrispondente indirizzo fisico *y* = *pf* | *of*.

immagine

Poiché la memoria virtuale di ogni processo viene caricata in porzioni di memoria fisica   
di dimensioni fissate, può verificarsi la frammentazione interna. In particolare, se l’ultima pagina   
della memoria virtuale di un processo non fosse completamente piena, una volta caricata   
questa pagina in un frame libero, si avrebbe dello spazio libero rimanente nel frame, non utilizzabile.

Un parametro di cui tenere conto nella tecnica della paginazione è la dimensione delle pagine:

* Al diminuire della dimensione delle pagine, aumenterebbe il numero di pagine di cui è costituita   
  la memoria virtuale di ogni processo, e quindi aumenterebbe la dimensione della tabella   
  delle pagine di ogni processo;
* Al crescere della dimensione delle pagine, il problema della frammentazione interna diventerebbe più rognoso. Mediamente, infatti, metà dell’ultima pagina della memoria virtuale   
  di ogni processo rimane vuota, per cui, con pagine più grandi, aumenterebbe lo spazio vuoto   
  nell’ultima pagina di ogni processo, e quindi aumenterebbe lo spazio libero non utilizzabile rimanente nel frame in cui tale pagina verrà caricata.

**PAGINAZIONE A DOMANDA**

La **tecnica della paginazione a domanda** coincide con la tecnica della paginazione,   
con la differenza che, essendo una tecnica con caricamento a domanda, ogni pagina   
della memoria virtuale di un processo verrà caricata in memoria fisica solo quando sarà richiesta effettivamente dal processo.

Per ogni pagina della memoria virtuale di un processo, la riga corrispondente nella tabella delle pagine del processo viene modificata in questo modo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| campo frame | bit di controllo | | | | |
| *numero pf del frame in cui è stata caricata la pagina, se P = 1*  *indirizzo I della pagina in memoria secondaria, se P = 0* | *R* | *W* | *U* | *M* | *P* |

* Si aggiunge un bit *U*, che vale 1 se, dall’ultima volta che è stata caricata in memoria fisica,   
  la pagina è stata mai acceduta;

Modifica

Uso

* Si aggiunge un bit *M*, che vale 1 se, dall’ultima volta che è stata caricata in memoria fisica,   
  la pagina è stata mai modificata;

Presenza

* Si aggiunge un bit *P*, che vale 1 se la pagina è presente in memoria fisica;
* Il campo frame *pf* contiene:
  + Il numero *pf* del frame in cui è stata caricata la pagina, se *P* = 1;
  + L’indirizzo *I* della pagina nella memoria secondaria, se *P* = 0.

A questo punto, quando un processo *PE* vuole accedere a una certa pagina *pg* della propria   
memoria virtuale, se la riga *pg*-esima nella tabella delle pagine del processo presenta *P* = 0,   
viene sollevata un’eccezione di page-fault, che la routine di gestione dell’eccezione gestisce   
in questo modo:

1. Viene scorsa la tabella dei frame fino a trovare un frame libero. Sia *pf* il numero del frame libero trovato;
2. La pagina *pg* del processo *PE* viene caricata nel frame *pf*, copiandola dalla memoria secondaria a partire dall’indirizzo *I* contenuto nel *campo frame* della riga *pg*-esima della tabella delle pagine   
   del processo *PE*;
3. Viene aggiornata la riga *pg*-esima della tabella delle pagine del processo *PE*, impostando:
   * Il *campo frame* a *pf*;
   * Il bit *P* a 1.
4. Viene aggiornata la riga *pf*-esima della tabella dei frame, indicando che nel frame *pf*   
   è ora presente la pagina *pg* del processo *PE*.
5. Poiché l’eccezione di page-fault ha causato l’abort dell’istruzione che l’ha generata,   
   viene rieseguita tale istruzione.

Nel caso in cui, al punto 1, tutti i frame fossero occupati, bisogna prima liberarne uno.  
Per fare ciò, si procede in questo modo:

1. Tramite un opportuno algoritmo, detto *algoritmo di rimpiazzamento*, viene scelta una pagina   
   presente in memoria fisica da rimpiazzare. Sia *pg*’ la pagina scelta, appartenente   
   al processo *PE*’;
2. La pagina *pg*’ del processo *PE*’ viene caricata in memoria secondaria a partire   
   da un certo indirizzo *I’*, copiandola dal frame il cui numero *pf* è contenuto nel *campo frame*   
   della riga *pg*’-esima della tabella delle pagine del processo *PE*’;
3. Viene aggiornata la riga *pg*’-esima della tabella delle pagine del processo *PE*’, impostando:
   * Il *campo frame* a *I’*;
   * Il bit *P* a0.
4. Viene aggiornata la riga *pf*-esima della tabella dei frame, indicando che ora il frame *pf* è libero.

L’algoritmo di rimpiazzamento *ottimo*, e cioè che darebbe luogo al minor numero possibile   
di page-fault, sceglie come pagina da rimpiazzare quella che sicuramente non verrà più acceduta   
o, se tutte le pagine dovranno ancora essere accedute, quella che verrà acceduta più tardi nel tempo. Questo algoritmo, tuttavia, è irrealizzabile, in quanto richiederebbe la conoscenza di come ogni processo si comporterà nel futuro.

Uno degli algoritmi di rimpiazzamento più semplici da usare è l’*algoritmo FIFO*, che sceglie   
come pagina da rimpiazzare quella che da più tempo è presente in memoria fisica.   
Questo algoritmo, tuttavia, ha prestazioni molto lontane da quelle dell’algoritmo ottimo.

Un algoritmo di rimpiazzamento con prestazioni molto vicine a quelle dell’algoritmo ottimo   
è l’*algoritmo LRU*, che sceglie come pagina da rimpiazzare quella meno recentemente usata.   
Questo algoritmo, tuttavia, è molto oneroso da implementare, in quanto richiederebbe memorizzare,   
per ogni pagina presente in memoria fisica, l’ultima volta che è stata acceduta.

Un algoritmo che è un buon compromesso tra prestazioni e onerosità dell’implementazione   
è l’*algoritmo second-chance*, che è un’approssimazione dell’algoritmo LRU. In questo algoritmo   
si ha un puntatore *vittima*, che può scorrere in maniera circolare attraverso tutte le pagine presenti   
in memoria fisica. Fintantoché *vittima* non punta ad una pagina con il bit *U* ad essa associato   
uguale a 0, viene azzerato il bit *U* associato alla pagina attualmente puntata da *vittima*   
e viene fatto avanzare *vittima*. La pagina cui *vittima* punterà alla fine sarà quella da rimpiazzare.

Un’ottimizzazione possibile consiste nel rimpiazzare, tra le pagine con il bit *U* associato uguale a 0,  
una con il bit *M* associato a 0, così che, se questa pagina è stata caricata   
in memoria secondaria precedentemente, non è necessario ricaricarcela, perché tanto,   
dall’ultima volta che è stata caricata in memoria fisica, questa pagina non è stata mai modificata.

Questi algoritmi possono variare un po’ a seconda della strategia di rimpiazzamento utilizzata:

* Rimpiazzamento locale: viene rimpiazzata una pagina dello stesso processo che ha generato un page-fault;
* Rimpiazzamento globale: viene rimpiazzata una pagina di un processo qualsiasi.

**MEMORIA SEGMENTATA E PAGINATA**

**SEGMENTAZIONE CON PAGINAZIONE**

Nella **tecnica della segmentazione con paginazione**, essendo una tecnica a memoria virtuale segmentata, la memoria virtuale di ogni processo è divisa in uno o più segmenti, numerati   
a partire da 0. Essendo inoltre una tecnica ad allocazione non contigua, ogni segmento di un processo viene diviso in *pagine*, numerate a partire da 0. Similmente, la memoria fisica viene divisa in frame, numerati a partire da 0, della stessa dimensione delle pagine. Ciascuna pagina verrà poi caricata   
in un certo frame, solo quando sarà richiesta effettivamente dal processo.

Per ogni segmento di un processo, se almeno una pagina del segmento è presente in memoria fisica,   
è salvata in memoria fisica una tabella delle pagine, dove la riga *i*-esima è relativa all’*i*-esima pagina   
del segmento, ed è costituita dagli stessi campi visti nella tecnica della paginazione a domanda:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| campo frame | bit di controllo | | | | |
| *numero pf del frame in cui è stata caricata la pagina, se P = 1*  *indirizzo I della pagina in memoria secondaria, se P = 0* | *R* | *W* | *U* | *M* | *P* |

Per ogni processo è salvata in memoria fisica una tabella dei segmenti, dove la riga *i*-esima   
è relativa all’*i*-esimo segmento del processo, ed è costituita dagli stessi campi visti   
nella tecnica della segmentazione a domanda, con le seguenti differenze:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| base | limite | bit di controllo | | | | |
| *indirizzo fisico I della tabella delle pagine del segmento, se P = 1*  *indirizzo I della tabella delle pagine del segmento in memoria secondaria, se P = 0* | *dimensione D del segmento* | *R* | *W* | *U* | *M* | *P* |

* Il bit *P* vale 1 se la tabella delle pagine del segmento è presente in memoria fisica;
* Il campo *base* contiene:
  + L’indirizzo fisico *I* della tabella delle pagine del segmento, se *P* = 1;
  + L’indirizzo *I* della tabella delle pagine del segmento in memoria secondaria, se *P* = 0.

In memoria fisica è poi salvata una tabella dei frame, dove la riga *i*-esima è relativa all’*i*-esimo frame, ed è costituita principalmente dallo stesso campo visto nella tecnica della paginazione:

|  |
| --- |
| *indicazione che il frame è libero, se il frame è libero*  *numero pg della pagina in esso presente e un’indicazione  sul processo cui tale pagina appartiene, se il frame è occupato* |

Inoltre, la CPU è dotata di due registri:

Segment Table Base Register

* STBR, contenente l’indirizzo fisico della tabella dei segmenti del processo in esecuzione;

Segment Table Limit Register

* STLR, contenente il numero di segmenti del processo in esecuzione.

Supponiamo che la CPU, eseguendo un processo, generi un indirizzo virtuale *x =* <*sg*, *sc*> = <*sg*, *pg* | *of*>.

Segmento paginato

Memoria virtuale segmentata

La traduzione effettuata dall’MMU di questo indirizzo virtuale *x* nel corrispondente indirizzo fisico *y* avviene in questo modo:

1. *sg* viene confrontato con STLR, e se risulta *sg* ≥ STLR, poiché il processo sta tentando   
   di accedere ad un segmento inesistente, viene sollevata un’eccezione.
2. Vengono letti i valori *I*, *D*, *R*, *W e P* della riga *sg*-esima della tabella dei segmenti puntata   
   da STBR:
   1. se risulta *sc* ≥ *D*, poiché il processo sta tentando di accedere ad una locazione   
      non appartenente al segmento *sg*-esimo, viene sollevata un’eccezione.
   2. se risulta *R* = 0 o *W* = 0, e il processo vuole accedere all’indirizzo virtuale *x*   
      rispettivamente in lettura o in scrittura, poiché sta tentando di accedere al segmento *sg*-esimo in un modo che non è permesso, viene sollevata un’eccezione.
   3. se risulta *P* = 0, viene sollevata un’eccezione di segment-fault, che la routine di gestione dell’eccezione gestisce in modo simile a quanto visto nella tecnica della segmentazione a domanda, con la differenza che in questo caso caricherà in memoria fisica   
      non il segmento *sg*-esimo, ma la tabella delle pagine del segmento *sg*-esimo;
3. Vengono letti i valori *pf*, *R*, *W* e *P* della riga *pg*-esima della tabella delle pagine che si trova all’indirizzo fisico *I*:
   1. se risulta *R* = 0 o *W* = 0, e il processo vuole accedere all’indirizzo virtuale *x*   
      rispettivamente in lettura o in scrittura, poiché sta tentando di accedere   
      alla pagina *pg*-esima in un modo che non è permesso, viene sollevata un’eccezione;
   2. se risulta *P* = 0, viene sollevata un’eccezione di page-fault, che la routine di gestione dell’eccezione gestisce in modo simile a quanto visto nella tecnica della paginazione   
      a domanda.
4. L’indirizzo virtuale *x* viene tradotto nel corrispondente indirizzo fisico *y* = *pf* | *of*.

immagine

**GESTIONE DELLA MEMORIA SU UNIX**

Su UNIX la memoria viene gestita attraverso la tecnica della segmentazione con paginazione,   
con la differenza che i frame vengono liberati non solo quando si verifica un page-fault,   
ma anche in altri casi, così da avere sempre dei frame liberi. In particolare, vengono tenute   
in considerazione tre diverse soglie:

LOTS (ancora tanti)

* *lotsfree*: il numero di frame liberi sotto il quale viene eseguito l’algoritmo di rimpiazzamento;

MINimi

* *minfree*: il numero minimo di frame liberi;

DESired

* *desfree*: il numero medio minimo di frame liberi.

A questo punto, la memoria viene effettivamente gestita attraverso i processi di sistema   
*pagedaemon* e *swapper:*

* *pagedaemon* va in esecuzione periodicamente, e se *num\_frame\_liberi* < *lotsfree,*esegue l’algoritmo di rimpiazzamento;
* *swapper* va in esecuzione se *num\_frame\_liberi* < *minfree* && *num\_medio\_frame\_liberi* < *desfree*, spostandoin memoria secondaria tutte le pagine attualmente presenti in memoria fisica   
  di uno o più processi fintantoché la condizione non è più soddisfatta.

**GESTIONE DELLE PERIFERICHE**

**CONCETTI INTRODUTTIVI**

Ogni periferica è costituita da due elementi:

* **Trasduttore**: la periferica vera e propria;
* **Controllore**: la parte della periferica che controlla il trasduttore. E’ dotato di almeno 3 registri, montati sullo spazio di I/O:
  + Un **registro di controllo**, attraverso il quale la CPU può dare un comando al controllore. Di questo registro sono importanti 2 bit:
    - Un *bit di start*, che quando posto a 1, attiva il controllore;
    - Un *bit di abilitazione delle interruzioni*, che quando posto a 1,   
      abilita il controllore a mandare un’interruzione esterna quando questo   
      avrà eseguito un comando;
  + Un **registro di stato**, attraverso il quale il controllore può comunicare   
    l’esito di un comando ricevuto dalla CPU. Di questo registro è importante il   
    *bit di condizione di errore*, che quando posto a 1, segnala che si è verificato un errore nell’ultimo comando eseguito;
  + Un **registro dati**, usato per trasferire dati dalla CPU alla periferica e/o viceversa.

Ogni volta che un processo applicativo vuole accedere ad una periferica:

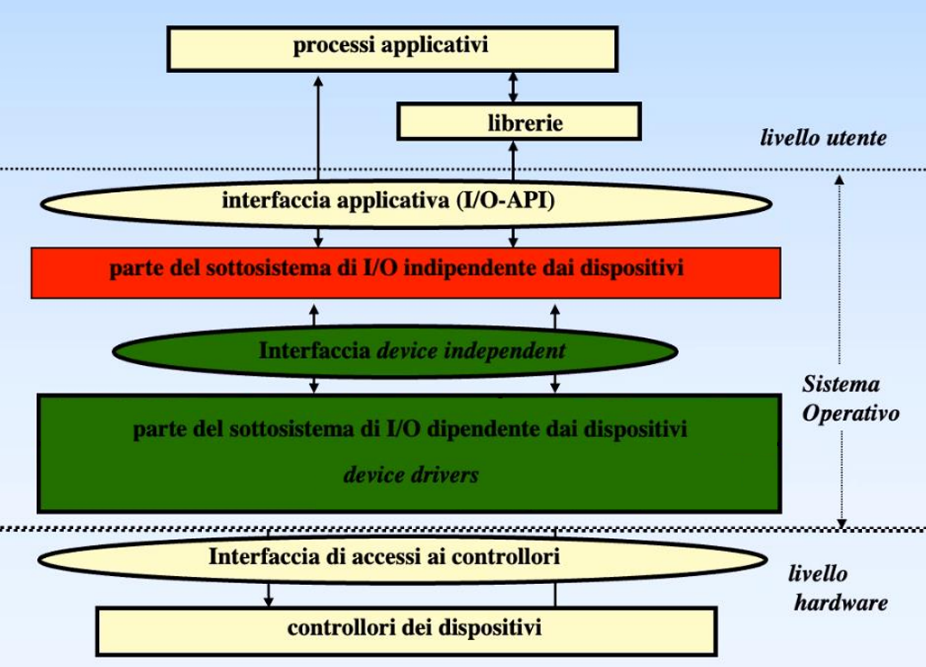
1. Prepara un comando;
2. Manda il comando al controllore della periferica;
3. Si sospende fintantoché il comando non è terminato.

Il comportamento di un controllore, a sua volta, si può rappresentare come quello di un processo,   
detto **processo esterno**, in esecuzione sul controllore stesso. Questo processo, continuamente:

1. Attende fintantoché non riceve un comando;
2. Esegue il comando;
3. Segnala la fine del comando.

**SOTTOSISTEMA DI I/O**

Il **sottosistema di I/O** è la componente del sistema operativo che si occupa di gestire le periferiche.   
L’interazione tra un processo applicativo e una periferica avviene in questo modo:

**

Quando un processo applicativo vuole accedere ad una periferica, invoca una funzione fornita   
dalla cosiddetta *interfaccia applicativa*.

Ogni funzione fornita dall’interfaccia applicativa è effettivamente implementata nella cosiddetta   
*parte del sottosistema di I/O indipendente dai dispositivi*, e fa uso del cosiddetto *device driver*   
della periferica cui si vuole accedere, e cioè di alcune strutture dati e funzioni specifiche   
per tale periferica, fornite dalla cosiddetta *interfaccia device independent*.

Il device driver di ogni periferica è effettivamente implementato nella cosiddetta *parte del sottosistema di I/O dipendente dai dispositivi*, ed è questo che si occupa di accedere effettivamente alla periferica, accedendo al controllore della periferica tramite la cosiddetta *interfaccia di accesso ai controllori*,   
costituita dai registri del controllore di ogni periferica.

**PARTE DEL SOTTOSISTEMA DI I/O INDIPENDENTE DAI DISPOSITIVI**

La parte del sottosistema di I/O indipendente dai dispositivi, nell’implementare le funzioni fornite dall’interfaccia applicativa, offre diverse funzionalità:

* **Buffering:** ogni volta che un processo applicativo vorrà trasferire dei dati da/a una periferica,   
  la parte del sottosistema di I/O indipendente dai dispositivi creerà un *buffer di sistema*,   
  e il trasferimento dei dati avverrà passando per questo buffer.

Questo viene fatto per due motivi:

* + Garantire disaccoppiamento spaziale: supponiamo che la periferica consenta   
    il trasferimento solo di *x* byte di dati per volta, e che il processo applicativo   
    voglia leggere dalla periferica un numero di byte inferiore a *x*.   
    Senza un buffer di sistema, questo non sarebbe possibile. Sfruttando il buffer   
    di sistema, invece, verrà prima effettuato un trasferimento di *x* byte dalla periferica   
    al buffer di sistema, ma poi verranno trasferiti dal buffer di sistema   
    al buffer del processo applicativo solo i byte richiesti.
  + Garantire disaccoppiamento temporale: supponiamo che il processo applicativo   
    voglia scrivere nella periferica un certo numero di byte di dati, e che la periferica   
    possa leggere un solo byte per volta. Senza un buffer di sistema, il processo applicativo   
    ogni volta dovrebbe attendere che la periferica consumi il byte precedente   
    prima di inviargliene un altro. Sfruttando il buffer di sistema, invece, i byte   
    del processo applicativo verranno tutti trasferiti nel buffer di sistema, e poi, uno ad uno,   
    saranno trasferiti dal buffer di sistema alla periferica.
* **Naming:** un processo applicativo può riferirsi ad una periferica usando un nome simbolico,   
  e la parte del sottosistema di I/O indipendente dai dispositivi si occuperà di tradurre   
  questo nome simbolico in un nome univoco (ad esempio, un intero).
* **Assegnazione delle periferiche ai processi applicativi:** la parte del sottosistema di I/O indipendente dai dispositivi si occupa di assegnare le periferiche ai processi applicativi, garantendo che l’accesso ad ogni periferica avvenga in mutua esclusione e che   
  un processo applicativo possa accedere ad una periferica solo se ha il permesso per farlo.

**PARTE DEL SOTTOSISTEMA DI I/O DIPENDENTE DAI DISPOSITIVI**

Tra le varie strutture dati che costituiscono il device driver di una periferica, la parte del sottosistema   
di I/O dipendente dai dispositivi definisce un **descrittore di periferica**, costituito tipicamente dai seguenti campi:

* L’indirizzo del registro di controllo della periferica;
* L’indirizzo del registro di stato della periferica;
* L’indirizzo del registro dati della periferica;
* Un semaforo *dati\_disponibili*, con valore iniziale 0, su cui il processo che vuole effettuare   
  un trasferimento di dati dalla/alla periferica si sospenderà fintantoché il trasferimento   
  non è concluso;
* Una variabile *contatore*, contenente il numero di byte di dati ancora da trasferire;
* Un variabile *puntatore,* che punta alla locazione del buffer di sistema in/da cui trasferire il prossimo dato;
* Una variabile *esito*, contenente l’esito dell’ultimo trasferimento di dati.

Ogni descrittore di periferica è accessibile tramite un array *descr*, indicizzato dal nome univoco   
della periferica.

Tra le funzioni che costituiscono il device driver di una periferica, la parte del sottosistema di I/O dipendente dai dispositivi definirà due funzioni del genere:

* int read(int *disp*, char\* *pbuf*, int *cont*): legge *cont* byte dal registro dati della periferica identificata dal nome univoco *disp*, copiandoli nel buffer di sistema *pbuf*.

Restituisce:

* + Il numero di byte effettivamente letti, in caso di successo;
  + -1, in caso di errore.
* int write(int *disp*, char\* *pbuf*, int *cont*): scrive *cont* byte nel registro dati della periferica identificata dal nome univoco *disp*, copiandoli dal buffer di sistema *pbuf.*

Restituisce:

* + Il numero di byte effettivamente scritti, in caso di successo;
  + -1, in caso di errore.

Supponiamo di avere una periferica che consente il trasferimento di un solo byte per volta.

La funzione read() sarà fatta in un modo del genere:

int read(int disp, char\* pbuf, int cont){

descr[disp].contatore = cont;

descr[disp].puntatore = pbuf;

*invio comando di lettura, bit di abilitazione interruzioni = bit di start = 1*

descr[disp].dati\_disponibili.wait();

if(descr[disp].esito != *codice\_errore*){

return cont-descr[disp].contatore;

}

else{

return -1;

}

}

La funzione read(), dopo aver inizializzato i campi *descr*[*disp*].*contatore* e *descr*[*disp*].*puntatore* rispettivamente con il numero *cont* di byte da leggere e l’indirizzo *pbuf* del buffer di sistema (e quindi, con l’indirizzo della prima locazione di tale buffer), invia al controllore della periferica il comando di voler leggere dalla periferica,   
abilita il controllore a mandare un’interruzione esterna ogni volta che questo avrà eseguito un comando,   
e attiva effettivamente il controllore.

Fatto ciò, il processo che ha invocato la funzione si sospende fintantoché il trasferimento dei dati non è concluso.

Terminato il trasferimento dei dati, la funzione ne controlla l’esito:

* In caso di successo, restituisce la differenza tra *cont* e il numero *descr[disp].contatore* di byte   
  che si sarebbero dovuti ancora leggere, e quindi restituisce il numero di byte effettivamente letti.
* In caso di errore, restituisce -1;

Ogni volta che il controllore della periferica avrà eseguito un comando di lettura dalla periferica,   
questo manderà un interruzione esterna, che la routine di gestione dell’interruzione gestirà   
in un modo del genere:

void inth(){

If(*bit di condizione di errore* == 0){

\*descr[disp].puntatore = *registro dati*;

descr[disp].contatore--;

descr[disp].puntatore++;

if(descr[disp].contatore != 0){

*invio comando di lettura*

}

else{

*bit di abilitazione interruzioni = bit di start = 0*

descr[disp].esito = *codice*;

descr[disp].dati\_disponibili.signal();

}

}

else{

*routine di gestione dell’errore*

if(*errore\_non\_recuperabile*){

descr[disp].esito = *codice\_errore*;

descr[disp].dati\_disponibili.signal();

}

}

}

La routine, per prima cosa, legge il registro di stato della periferica:

* Se il controllore ha eseguito il comando senza alcun errore, nel registro dati della periferica   
  ci sarà un byte da leggere. La routine allora copia il contenuto del registro dati della periferica   
  nella locazione del buffer di sistema puntata da *descr*[*disp*].*puntatore*, decrementa *descr*[*disp*].*contatore*,   
  in quanto adesso rimane da leggere dalla periferica un byte in meno, e incrementa *descr*[*disp*].*puntatore*,   
  così che il prossimo dato venga copiato nella locazione successiva del buffer di sistema.

Fatto ciò:

* + Se ci sono ancora byte da leggere, la routine invia al controllore un nuovo comando di lettura   
    dalla periferica;
  + Altrimenti, la routine disabilita il controllore a mandare interruzioni esterne, disattiva il controllore, scrive in *descr*[*disp*].*esito* un codice che comunica l’esito positivo del trasferimento   
    e segnala al processo che ha invocato la funzione read() che il trasferimento è concluso;
* Altrimenti, la routine esegue una routine di gestione dell’errore, e se a seguito di questa l’errore   
  non viene risolto, la routine scrive in *descr*[*disp*].*esito* il codice dell’errore che si è verificato   
  e abortisce il trasferimento, per cui segnala al processo che ha invocato la funzione read() che il trasferimento   
  è concluso.

**GESTIONE DI UN TIMER**

Un **timer** è una particolare periferica usata per gestire eventi legati al tempo.

Un timer misura il tempo in quanti di dimensione fissata, la cui durata può essere specificata scrivendola nel registro dati della periferica. Ogni volta che trascorre un quanto di tempo,   
il timer manda un’interruzione esterna.

Supponiamo di avere un timer che fornisce ai processi la possibilità di sospendersi   
per un numero di quanti di tempo da loro specificato. Il descrittore del timer sarà costituito allora   
dai seguenti campi:

* L’indirizzo del registro di controllo del timer;
* L’indirizzo del registro di stato del timer;
* L’indirizzo del registro dati del timer;
* Un array *fine\_attesa* di *N* semafori, ciascuno inizializzato a 0, dove *fine\_attesa*[*i*] è il semaforo su cui l’*i*-esimo processo si sospenderà fintantoché non sarà trascorso il numero di quanti   
  di tempo da lui specificato;
* Un array *ritardo* di *N* interi, dove *ritardo*[*i*], se diverso da 0, contiene il numero di quanti di tempo   
  per cui l’*i*-esimo processo dovrà ancora rimanere sospeso.

Tra le funzioni che costituiscono il device driver di un timer, la parte del sottosistema di I/O   
dipendente dai dispositivi definirà una funzione void delay(int *n*), che sospende il processo   
che la invoca per *n* quanti di tempo. Questa funzione sarà fatta in un modo del genere:

void delay(int n){

int proc = *indice del processo che ha invocato la funzione*;

descrittore.ritardo[proc] = n;

descrittore.fine\_attesa[proc].wait();

}

La funzione delay(), dopo aver recuperato l’indice *proc* del processo che l’ha invocata,   
inizializza il campo *descrittore*.ritardo[*proc*] con il numero *n* di quanti di tempo per cui il processo dovrà   
rimanere sospeso.

Fatto ciò, il processo che ha invocato la funzione si sospende fintantoché non sarà trascorso il numero di quanti   
di tempo da lui specificato.

Ogni volta che sarà passato un quanto di tempo, il controllore del timer manderà un’interruzione esterna, che la routine di gestione dell’interruzione gestirà in un modo del genere:

void inth(){

for(int i = 0; i < N; i++){

if(descrittore.ritardo[i] != 0){

descrittore.ritardo[i]--;

if(descrittore.ritardo[i] == 0){

descrittore.fine\_attesa[i].signal();

}

}

}

}

La routine scorre l’array *descrittore.ritardo*, e per ogni elemento, se diverso da 0, e cioè se corrispondente  
ad un processo che è attualmente sospeso a seguito dell’invocazione della funzione delay(),   
lo decrementa, dato che adesso il processo corrispondente deve rimanere sospeso per un quanto di tempo in meno.

Se a seguito del decremento l’elemento è uguale a 0, la routine segnala al processo corrispondente  
che è trascorso il numero di quanti di tempo da lui specificato.

decrementando ogni elemento diverso da 0, e cioè ogni elemento corrispondente ad un processo che è attualmente sospeso a seguito dell’invocazione della funzione delay(),   
dato che adesso tale processo deve rimanere sospeso per un quanto di tempo in meno.

Se a seguito del decremento un elemento dell’array è uguale a 0, la routine segnala al processo corrispondente  
che è trascorso il numero di quanti di tempo da lui richiesto.

**FILE SYSTEM** *05-12-24*

Il **file system** è la parte del sistema operativo che permette l’accesso e la memorizzazione di dati   
in memoria secondaria.

**ORGANIZZAZIONE LOGICA DEL FILE SYSTEM**

Il file system organizza i dati presenti in memoria secondaria in *file* e *directory*.

Un **file** rappresenta un contenitore di dati. Ogni file è caratterizzato da almeno un nome,   
ed ha associata una struttura dati, *detta descrittore di file*, costituita da:

* Uno o più puntatori alla memoria secondaria che permettono di accedere al contenuto del file;
* La dimensione del file;
* La data e ora di creazione del file;
* La data e ora di modifica del file.
* I permessi di accesso al file.

Una **directory** rappresenta un contenitore di file. Una directory è sua volta un file,   
e quindi anche ad essa è associato un descrittore di file. In alcuni sistemi operativi,   
una directory ha come contenuto i descrittori dei file appartenenti alla directory;   
in altri, invece, tutti i descrittori di file sono memorizzati in memoria secondaria in una   
*tabella dei descrittori di file*, e una directory ha come contenuto solo i riferimenti   
nella tabella dei descrittori di file ai descrittori dei file appartenenti alla directory.

**ACCESSO AL FILE SYSTEM**

**RECORD LOGICI**

Il file system organizza i dati contenuti in ogni file in insiemi di byte di dimensione fissata,   
detti *record logici*, numerati in maniera progressiva. Un record logico è l’unità di trasferimento dei file, per cui:

* In caso di lettura da un file, verranno copiati dal file uno o più record logici;
* In caso di scrittura su un file, verranno modificati o inseriti nel file uno o più record logici.

In ogni caso, ogni volta che si vuole accedere a un file, bisognerebbe come minimo   
fare un accesso in memoria secondaria per recuperare il descrittore del file e leggere così i puntatori alla memoria secondaria che permettono di accedere al contenuto del file, e poi fare ulteriori accessi   
in memoria secondaria per accedere ai record logici del file.

Poiché tutto questo sarebbe troppo oneroso, il sistema operativo mantiene in memoria RAM   
una *tabella dei file attivi* contenente, per ogni file attualmente in uso, una copia del descrittore del file. Inoltre, parte del contenuto di ogni file attualmente in uso viene memorizzata in memoria RAM,   
così che l’accesso al file venga effettuato in memoria RAM, molto più veloce   
rispetto alla memoria secondaria, e una volta finito di accedere al file, le eventuali modifiche effettuate vengono salvate in memoria secondaria.

**METODI DI ACCESSO AI FILE**

I metodi di accesso ai file più diffusi sono i seguenti:

* **Metodo di accesso sequenziale:** per accedere ad un particolare record logico *Ri* di un file,   
  è necessario accedere prima a tutti i precedenti record del file, partendo dal primo.   
  A questo scopo, ad ogni sessione di accesso a un file è associato un *I/O pointer*,   
  che punta al prossimo record da accedere.

A questo punto, per effettuare l’accesso a un file, il file system fornisce due funzioni del genere:

* + **readnext(f, &V):** legge dal file *f* il record logico puntato dall’I/O pointer,salvandolo in *V*;
  + **writenext(f, V):** scrive *V* nel file *f* al record logico puntato dall’I/O pointer.

Entrambe queste funzioni, una volta effettuato l’accesso al record logico puntato   
dall’I/O pointer, fanno avanzare l’I/O pointer al record logico successivo del file.

* **Metodo di accesso diretto:** è possibile accedere ad un particolare record logico *Ri* di un file   
  in maniera diretta.

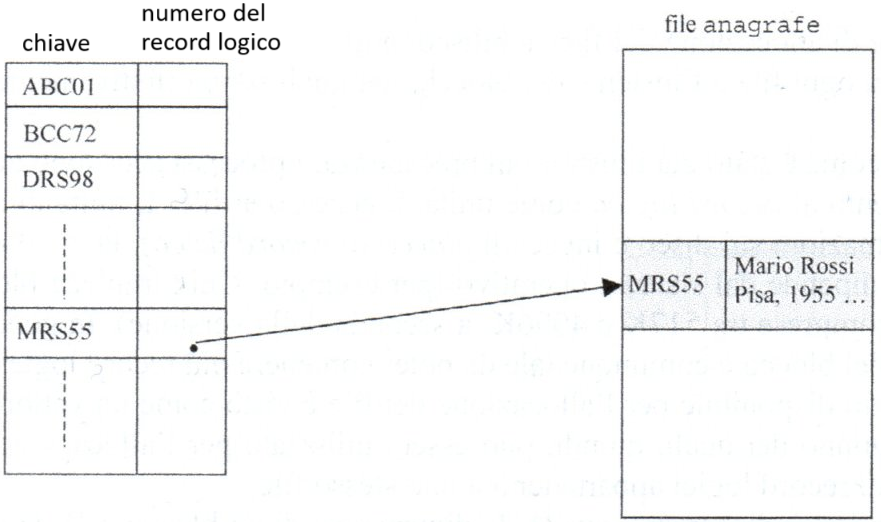
A questo punto, per effettuare l’accesso a un file, il file system fornisce due funzioni del genere:

READDirect

* + **readd(f, i, &V):** legge dal file *f* il record logico *Ri*,salvandolo in *V*;

WRITEDDirect

* + **writed(f, i, V):** scrive *V* nel record logico *Ri* del file *f*.
* **Metodo di accesso a indice:** ogni record logico di un file è caratterizzato da una *chiave*,   
  e cioè un nome simbolico che permette di identificare univocamente il record logico.   
  Ad ogni file è poi associata una struttura dati detta *indice*, che associa ad ogni chiave   
  il numero del record logico corrispondente.

****

A questo punto, per effettuare l’accesso a un file, il file system fornisce due funzioni del genere:

READKey

* + **readk(f, key, &V):** legge dal file *f* il record logico con chiave *key,* salvandolo in *V*;

WRITEKey

* + **writek(f, key, V):** scrive *V* nel record logico con chiave *key* del file *f*.

Entrambe queste funzioni, per determinare il record logico con una certa chiave,   
accedono all’indice del file.

**ORGANIZZAZIONE FISICA DEL FILE SYSTEM**

**BLOCCHI**

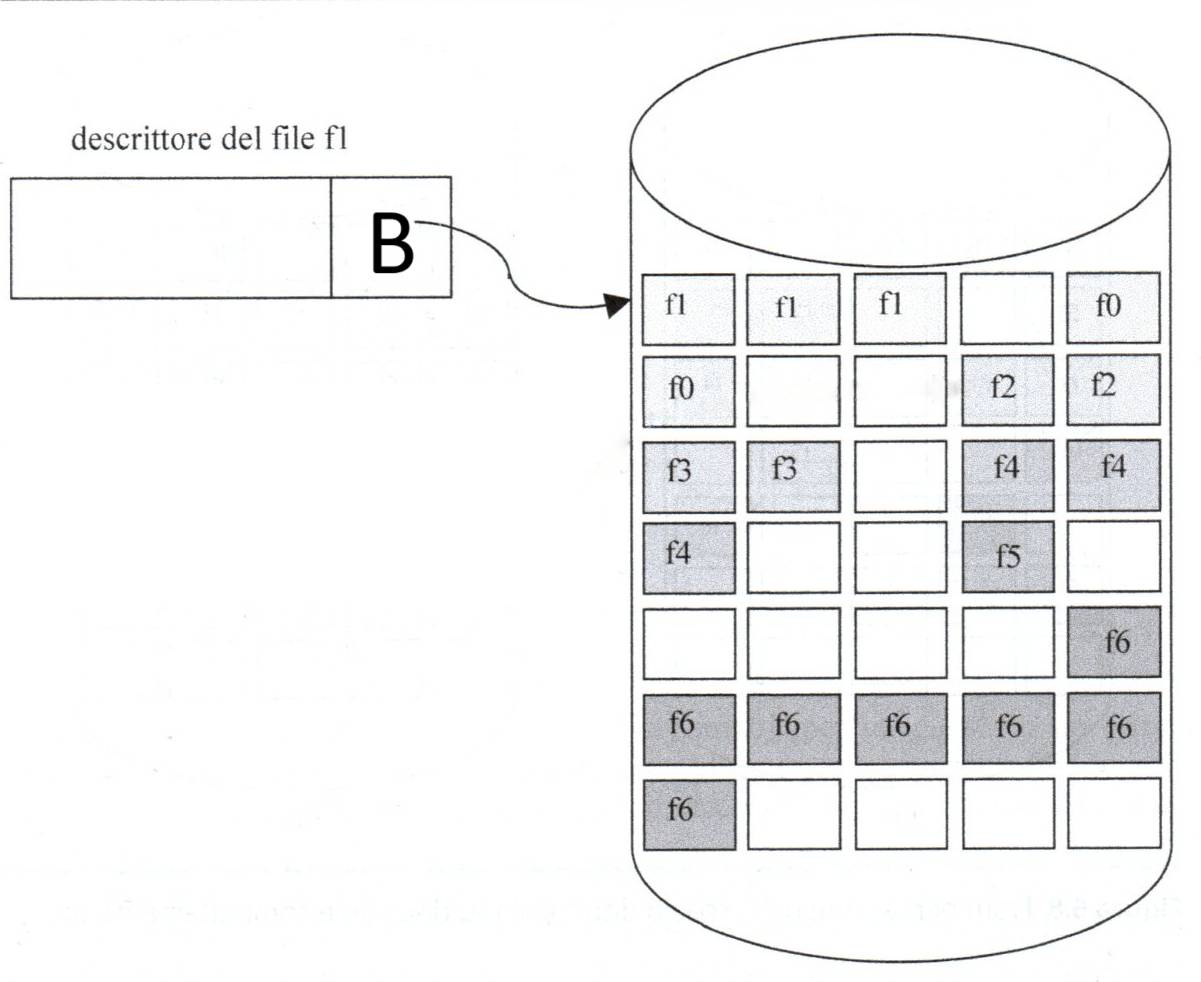
Il file system organizza la memoria secondaria in insiemi di byte di dimensione fissata, detti *blocchi,*   
ciascuno identificato da un indirizzo progressivo. Uno dei compiti del file system è caricare   
i record logici di ogni file nei blocchi della memoria secondaria.

**METODI DI ALLOCAZIONE DEI FILE**

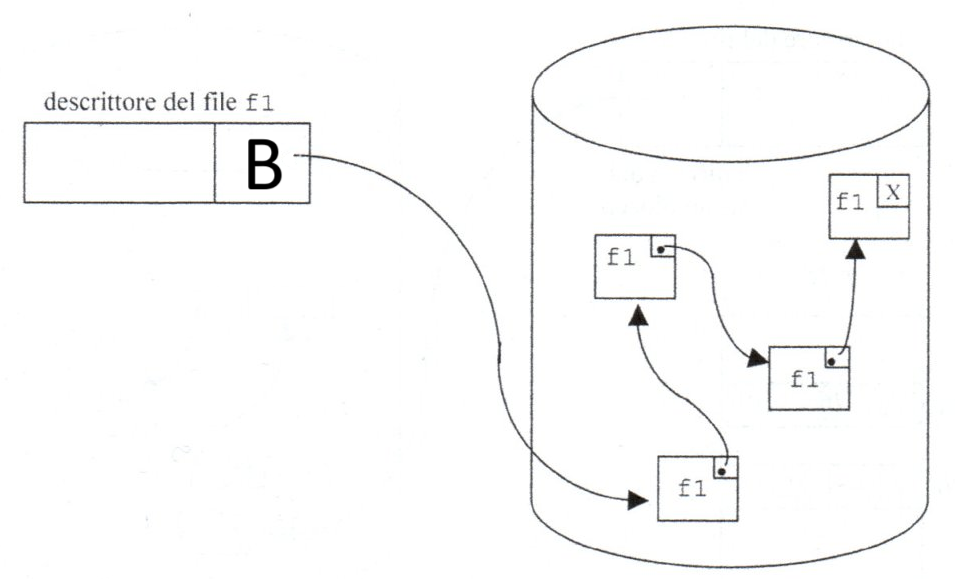
Indicando con *Db* la dimensione di un blocco e con *Dr* la dimensione di un record logico,   
il numero di record logici contenuti in ogni blocco sarà dato da *Nb* = *Db*/*Dr*.

All’interno di ogni blocco sono caricati record logici contigui di uno stesso file.   
Per quanto riguarda invece la scelta dei blocchi in cui caricare i file, questa differisce a seconda del metodo di allocazione dei file utilizzato:

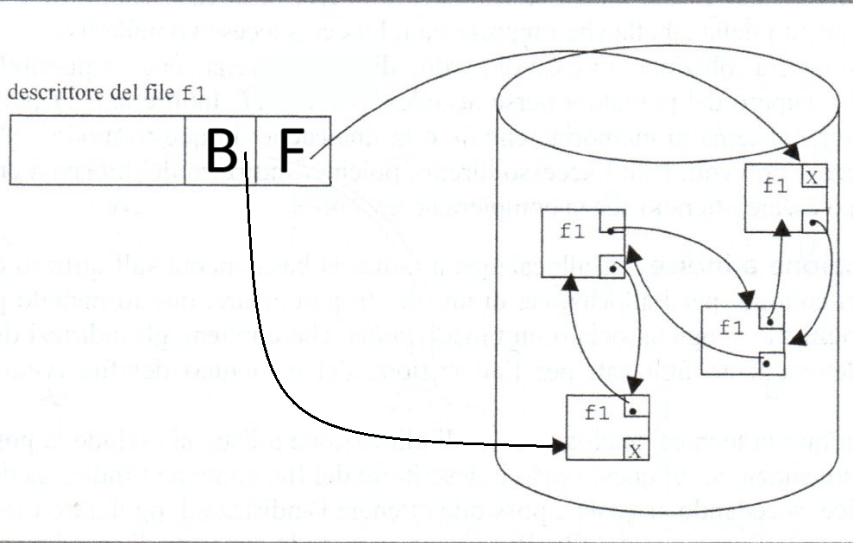
* **Metodo di allocazione contigua:** ogni file viene caricato in blocchi contigui.   
  Nel descrittore di ogni file è memorizzato l’indirizzo *B* del blocco in cui è memorizzata   
  la parte iniziale del file.

****

* + Vantaggi:
    - Se si vuole accedere ad un certo record logico *Ri* di un file usando il metodo   
      di accesso diretto, l’indirizzo del blocco preciso in cui è contenuto   
      tale record si può ottenere senza fare alcun accesso in memoria secondaria,   
      e sarà l’indirizzo *B* + i/*Nb*.
  + Svantaggi:
    - Ogni volta che bisogna caricare un file in memoria secondaria, bisogna trovare un insieme di blocchi liberi contigui di dimensione sufficientemente grande   
      a contenere il file;
    - Man mano che si caricano file in memoria secondaria può verificarsi   
      la cosiddetta *frammentazione dello spazio disponibile*, e cioè la situazione   
      in cui lo spazio libero presente in memoria secondaria è frammentato   
      in blocchi non contigui, presenti tra i vari file caricati. Un modo per risolvere   
      questo problema può essere effettuare periodicamente il cosiddetto *compattamento*, e cioè spostare i file in modo da avere un unico spazio libero presente in memoria secondaria.
    - Al crescere della dimensione di un file, questo dovrà occupare i blocchi contigui successivi all’ultimo blocco occupato dal file, e se questi non fossero liberi, bisognerà effettuare degli spostamenti di file.
* **Metodo di allocazione a lista concatenata:** ogni file viene caricato in blocchi   
  non necessariamente contigui, organizzati in una lista concatenata: in ogni blocco   
  viene riservato dello spazio a un puntatore al blocco in cui è memorizzata   
  la parte successiva del file. Nel descrittore di ogni file è memorizzato l’indirizzo *B* del blocco   
  in cui è memorizzata la parte iniziale del file.

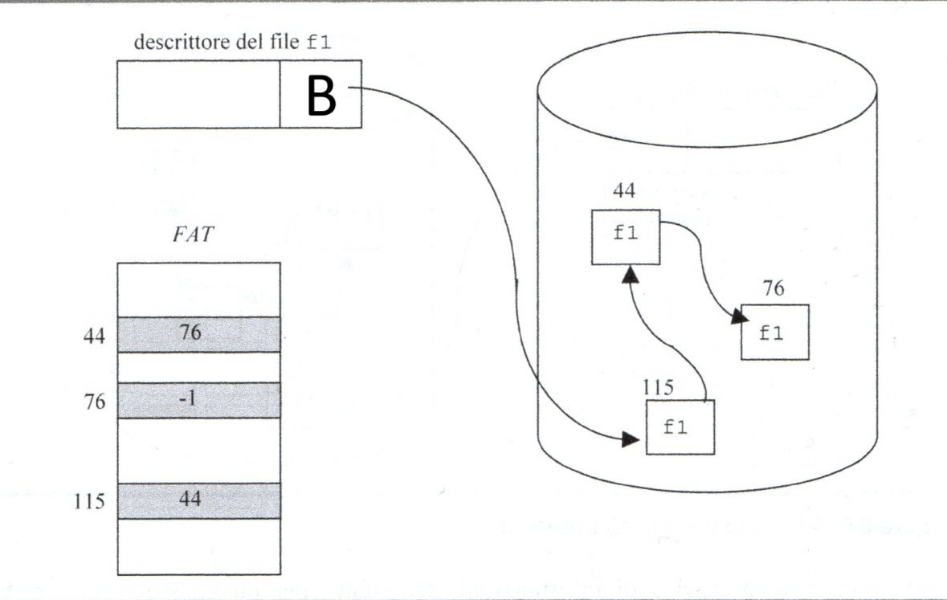
****

* + Vantaggi:
    - Poiché un file viene caricato in blocchi non necessariamente contigui,   
      non si hanno gli svantaggi presenti invece nel metodo di allocazione contigua.
  + Svantaggi:
    - Non tutto un blocco può essere utilizzato per memorizzare una parte di un file,   
      in quanto nel blocco deve essere necessariamente riservato dello spazio   
      al puntatore al blocco in cui è memorizzata la parte successiva del file;
    - Se si vuole accedere ad un certo record logico *Ri* di un file usando il metodo   
      di accesso diretto, per trovare l’indirizzo del blocco preciso in cui è contenuto tale record bisogna fare *i*/*Nb* accessi in memoria secondaria,   
      in quanto bisogna partire dal blocco in cui è memorizzata la parte iniziale del file e, fintantoché non si arriva al blocco desiderato, recuperare l’indirizzo del blocco in cui è memorizzata la parte successiva del file e accedere a tale blocco;
    - Se si danneggia il puntatore contenuto in un blocco in cui è memorizzata   
      una parte di un file, tutta la parte successiva del file diventa inaccessibile.
* **Metodo di allocazione a doppia lista concatenata:** coincide con il metodo di allocazione a lista concatenata, con la differenza che in ogni blocco in cui è stata caricata una parte di un file   
  viene riservato dello spazio anche a un puntatore al blocco in cui è memorizzata   
  la parte precedente del file. Inoltre, nel descrittore di ogni file è memorizzato anche   
  l’indirizzo *F* del blocco in cui è memorizzata la parte finale del file.

****

I vantaggi e gli svantaggi sono gli stessi del metodo di allocazione a lista concatenata,   
con la differenza che:

* + Si ha una ulteriore riduzione della parte di un file che si può memorizzare in un blocco, in quanto nel blocco deve essere necessariamente riservato dello spazio   
    anche al puntatore al blocco in cui è memorizzata la parte precedente del file.
  + Nel caso in cui si danneggiasse uno dei due puntatori contenuto in un blocco   
    in cui è memorizzata la parte di un file, sarebbe ancora possibile accedere   
    al resto del file.
* **Metodo di allocazione tramite FAT (File Allocation Table):** coincide con il metodo   
  di allocazione a lista concatenata, con la differenza che in memoria secondaria è memorizzata anche una tabella chiamata *FAT*, dove l’entrata *i*-esima è relativa al blocco di indirizzo *i*,   
  e contiene:
  + Se il blocco è libero, un’indicazione che è libero;
  + Se il blocco è occupato da una parte di un file, l’indirizzo del blocco che contiene   
    la parte successiva del file.



I vantaggi e gli svantaggi sono gli stessi del metodo di allocazione a lista concatenata,   
con la differenza che, nel caso in cui si danneggiasse il puntatore contenuto   
in un blocco in cui è memorizzata una parte di un file, sarebbe ancora possibile accedere   
al resto del file tramite la File Allocation Table.

* **Metodo di allocazione a indice:** ogni file viene caricato in blocchi   
  non necessariamente contigui. Ad ogni file è poi associato un blocco indice,   
  contenente i puntatori ai blocchi utilizzati per memorizzare il contenuto del file.   
  Nel descrittore di ogni file è memorizzato l’indirizzo *IX* del blocco indice associato al file.

****

* + Vantaggi:
    - Se si vuole accedere ad un certo record logico *Ri* di un file usando il metodo   
      di accesso diretto, per trovare l’indirizzo del blocco preciso in cui è contenuto tale record basta fare un solo accesso in memoria secondaria, accedendo   
      al blocco indice associato al file e leggendo l’indirizzo del blocco desiderato.
  + Svantaggi:
    - Un blocco indice può memorizzare un numero limitato di puntatori a blocchi,   
      dunque, usando un solo blocco indice, sarebbe necessario limitare   
      la dimensione dei file.

**FILE SYSTEM IN UNIX**

Il metodo di allocazione dei file usato su UNIX è il metodo a indice “a più livelli”: il descrittore di ogni file   
contiene 13 puntatori alla memoria secondaria, dove:

* I primi 10 sono puntatori a blocchi usati per memorizzare il contenuto del file;
* L’11-esimo è un puntatore a un blocco indice, contenente puntatori a blocchi   
  usati per memorizzare il contenuto del file;
* Il 12-esimo è un puntatore a un blocco indice, contenente puntatori a blocchi indice,   
  contenenti puntatori a blocchi usati per memorizzare il contenuto del file;
* Il 13-esimo è un puntatore a un blocco indice, contenente puntatori a blocchi indice,   
  contenenti puntatori a blocchi indice, contenenti puntatori a blocchi usati per memorizzare   
  il contenuto del file.

**PROTEZIONE** *10-12-24*

La **protezione** è l’insieme di attività svolte per garantire il controllo degli accessi alle risorse.

**MODELLO, POLITICA E MECCANISMO**

Il **controllo degli accessi** si compone di un *modello*, una *politica* e un *meccanismo*:

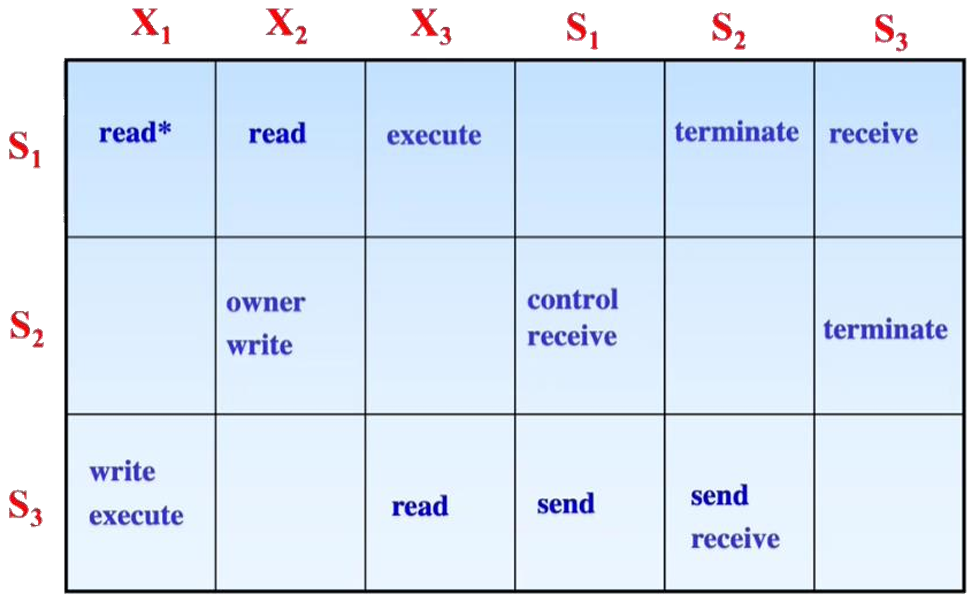
* Il **modello di protezione** definisce gli *oggetti* e i *soggetti*:
  + Un **oggetto** è una risorsa (fisica o logica);
  + Un **soggetto** è una coppia <*processo*, *dominio*>, dove il *dominio* è l’insieme   
    dei diritti di accesso del *processo* su ogni oggetto, e cioè l’insieme delle azioni   
    che il *processo* può svolgere su ogni oggetto.
* La **politica di protezione** stabilisce, per ogni soggetto, il suo dominio. Una politica di protezione può essere definita in diversi modi:
  + **Discretional Access Control**: il soggetto che crea un oggetto stabilisce,   
    per ogni soggetto, i diritti d’accesso su tale oggetto;
  + **Mandatory Access Control**: un’organizzazione centralizzata stabilisce,   
    per ogni soggetto, i diritti di accesso su ogni oggetto;
  + **Role-Based Access Control**: un’organizzazione centralizzata definisce dei ruoli, stabilendo, per ogni ruolo, i diritti di accesso su ogni oggetto, e ogni soggetto appartiene a uno o più ruoli.

Una caratteristica comune a tutte le politiche di protezione è l’adesione al *principio   
del minimo privilegio*: a ogni soggetto va garantito un dominio strettamente necessario   
alla sua esecuzione.

* Un **meccanismo** **di protezione** è lo strumento attraverso il quale viene fatta rispettare   
  la politica di protezione scelta.

**MATRICE DEGLI ACCESSI**

Una **matrice degli accessi A** è una matrice dove l’elemento *A*[*Si, Xj*] elenca i diritti di accesso   
del soggetto *Si* sull’oggetto/soggetto *Xj*:



Questa matrice può essere usata effettivamente per garantire il controllo degli accessi alle risorse: quando un soggetto *Si* vuole esercitare un diritto d’accesso α sull’oggetto *Oj*, il meccanismo   
di protezione controlla che α sia contenuto nell’elemento *A*[*Si, Oj*], e in caso negativo,   
l’accesso viene bloccato.

Questa matrice, inoltre, può essere usata per modificare in maniera controllata i diritti di accesso. Supponiamo che i modi permessi per modificare i diritti di accesso siano due:

* Propagazione dei diritti di accesso;
* Aggiunta e rimozione di diritti di accesso.

Nel caso della propagazione dei diritti di accesso, un soggetto *Si* può propagare un diritto di accesso α   
su un oggetto *Oj* a un altro soggetto *Si2* solo se, nell’elemento *A*[*Si, Oj*], α è indicato con un copy flag ‘\*’.   
Il soggetto *Si* può poi decidere se propagare anche il copy flag, permettendo così a *Si2*di propagare a sua volta α su *Oj*, e può decidere se trasferire α, e cioè se, a seguito della propagazione,   
*Si* perderà α su *Oj*.

Nel caso dell’aggiunta e rimozione di diritti di accesso:

* Un soggetto *Si* può aggiungere un diritto di accesso su un oggetto *Oj* a un soggetto *Si2*   
  solo se, nell’elemento *A*[*Si, Oj*]*,* è presente il diritto *owner*;
* Un soggetto *Si* può rimuovere un diritto di accesso su un oggetto *Oj* da un soggetto *Si2*   
  solo se nell’elemento *A*[*Si, Oj*] è presente il diritto *owner*, oppure se nell’elemento *A*[*Si, Sj*]   
  è presente il diritto *control*.

Dal punto di vista realizzativo, tuttavia, la matrice degli accessi non viene implementata   
come una matrice, per vari motivi:

* La dimensione di questa matrice sarebbe troppo grande, dato che di solito si hanno   
  molti soggetti e molti oggetti;
* Buona parte della matrice rimarrebbe vuota, dato che, per il principio del minimo privilegio,   
  ogni soggetto ha un dominio strettamente necessario alla sua esecuzione.

Sono due i modi in cui viene effettivamente implementata la matrice degli accessi:

* *Per colonne*: ad ogni oggetto è associata una lista, detta **Access Control List (ACL),**dove ogni elemento è relativo ad un soggetto che ha almeno un diritto di accesso   
  su tale oggetto, e contiene l’insieme dei diritti di accesso di tale soggetto su tale oggetto.
* *Per righe*: ad ogni soggetto è associata una lista, detta **Capability List (CL),** dove ogni elemento è relativo ad un oggetto su cui il soggetto ha almeno un diritto di accesso, e contiene   
  l’insieme dei diritti di accesso di tale soggetto su tale oggetto.

Capacità (del soggetto)

A seconda del modo utilizzato, si hanno vantaggi e svantaggi:

* Usando le ACL è più semplice modificare un diritto d’accesso su un oggetto per più soggetti,   
  in quanto basta esaminare solo la ACL associata all’oggetto e accedere agli elementi relativi   
  ai soggetti di cui si vuole modificare tale diritto, modificando opportunamente   
  l’insieme dei diritti di accesso contenuto negli elementi.

Usando le CL, invece, bisognerebbe esaminare la CL associata ad ogni soggetto di cui si vuole modificare il diritto d’accesso e accedere per ognuna di queste all’elemento relativo all’oggetto*,*modificando opportunamente l’insieme dei diritti di accesso contenuto nell’elemento.

* Usando le CL è più semplice il controllo degli accessi, in quanto, se ad esempio un soggetto *Si*   
  vuole esercitare un diritto d’accesso α su un oggetto *Oj*, basta scorrere la CL   
  associata a *Si* fintantoché non viene trovato l’elemento relativo all’oggetto *Oj* e controllare   
  che nell’insieme dei diritti di accesso contenuto nell’elemento ci sia α, e dato che,   
  per il principio del minimo privilegio, ogni soggetto ha un dominio strettamente necessario   
  alla sua esecuzione, questa CL non sarà troppo lunga.

Usando le ACL, invece, bisognerebbe scorrere la ACL associata ad *Oj* fintantoché   
non viene trovato l’elemento relativo al soggetto *Si* e controllare che nell’insieme dei diritti   
di accesso contenuto nell’elemento ci sia α, ma questa ACL potrebbe essere abbastanza lunga.

**POLITICA DI SICUREZZA MULTILIVELLO**

La **politica di sicurezza multilivello** è un tipo di politica di protezione MAC in cui   
un’organizzazione centralizzata definisce *N* livelli di sicurezza e stabilisce, per ogni soggetto e oggetto, il suo livello di sicurezza.

La **politica Bell-La Padula** è un tipo di politica di sicurezza multilivello. Questa politica impone   
due regole:

* Un soggetto che appartiene ad un certo livello di sicurezza può leggere solo oggetti   
  che appartengono ad un livello di sicurezza minore o uguale al suo;
* Un soggetto che appartiene ad un certo livello di sicurezza può scrivere solo oggetti   
  che appartengono ad un livello di sicurezza maggiore o uguale al suo.

La politica Bell-La Padula permette di risolvere il cosiddetto problema del cavallo di Troia.   
Supponiamo di avere un oggetto O1, contenente informazioni riservate, sul quale solo un soggetto S1   
ha il diritto di accesso in lettura e scrittura. Un soggetto ostile S2 vuole leggere il contenuto di O1,   
e a tale scopo crea due oggetti:

* Un oggetto O2;
* Un oggetto O3, contenente un cavallo di Troia (un tipo di programma malevolo)   
  che esegue le seguenti operazioni:
  + Legge il contenuto di O1;
  + Scrive in O2 il contenuto di O1.

Essendo l’owner di questi due oggetti, S2 può aggiungere diritti di accesso su di essi, e lo fa   
in questo modo:

* Su O2, S2 assegna il diritto di accesso in lettura a se stesso e il diritto di accesso in scrittura a S1.
* Su O3, S2 assegna il diritto di accesso in esecuzione a S1;

Supponiamo che, ad un certo punto, S1 voglia eseguire O3:

* Senza la politica Bell-La Padula, succede questo:

1. S1 esegue il cavallo di Troia contenuto in O3: questo è consentito, in quanto S1   
   ha il diritto di accesso in esecuzione su O3;
2. Il cavallo di Troia legge il contenuto di O1: questo è consentito, in quanto S1,   
   che esegue il cavallo di Troia, ha il diritto di accesso in lettura su O1;
3. Il cavallo di Troia scrive in O2 il contenuto di O1: questo è consentito, in quanto S1,   
   che esegue il cavallo di Troia, ha il diritto di accesso in scrittura su O2.

Alla fine, dunque, in O2 ci sarà il contenuto di O1, e S2 può leggerlo, in quanto ha il diritto   
di accesso in lettura su O2.

* Con la politica Bell-La Padula, definiamo innanzitutto i livelli di sicurezza *pubblico* e *riservato*,   
  con il secondo maggiore del primo, e stabiliamo che S1 e i suoi oggettiappartengono   
  al livello di sicurezza *riservato*, mentre S2 e i suoi oggetti appartengono al livello di sicurezza *pubblico*. A questo punto, succede questo:

1. S1 esegue il cavallo di Troia contenuto in O3: questo è consentito, in quanto S1   
   ha il diritto di accesso in esecuzione su O3, e considerando l’esecuzione di un programma come una lettura del codice del programma, anche questo è consentito,   
   in quanto questo oggetto appartiene ad un livello di sicurezza minore di quello di S1.
2. Il cavallo di Troia legge il contenuto di O1: questo è consentito, in quanto S1,   
   che esegue il cavallo di Troia, ha il diritto di accesso in lettura su O1, e questo oggetto appartiene ad un livello di sicurezza uguale a quello di S1.
3. Il cavallo di Troia scrive in O2 il contenuto di O1: questo non è consentito, in quanto S1,   
   che esegue il cavallo di Troia, ha il diritto di accesso in scrittura su O2, ma questo oggetto appartiene ad un livello di sicurezza minore di quello di S1.