

Appunti di Sistemi Operativi

<giomba@live.it>

Contents

0 Introduzione	3
0.1 Informazioni generali	3
0.2 Informazioni particolari	3
0.3 Cinque	3
1 Mappe	4
1.1 Argomenti generali	4
2 Tassonomia di Flynn	5
2.1 Terminologia utilizzata	5
2.2 SISD - Single instruction and single data streams	5
2.3 MISD - Multiple instruction stream and single data stream	6
2.4 SIMD - Single instruction stream and multiple data stream	6
2.5 MIMD - Multiple instruction and multiple data streams	6
2.5.1 DM - Distributed memory	6
2.5.2 SM - Shared memory	7
3 Speed-Up	7
4 Reti di interconnessione	7
5 Schedulazione	8
6 Sincronizzazione tra processi	9
6.1 Problemi	10
6.1.1 Problema del buffer limitato	10
6.1.2 Problema dei lettori-scrittori	10
6.1.3 Problema dei 5 filosofi	10
6.2 Soluzioni ai problemi di mutua esclusione e comunicazione	10
6.2.1 Test and Set Lock	10
6.2.2 Semafori: wait(); signal()	10
6.2.3 Monitor e variabili condition	11
6.3 Deadlock (stallo)	13
6.3.1 Prevenzione statica	13
6.3.2 Prevenzione dinamica	14
6.3.3 Individuazione di blocchi critici e recupero	16

7 Gestione della memoria	16
7.1 Concetti generali	16
7.1.1 Terminologia utilizzata	16
7.1.2 Caratteristiche della Gestione	17
7.2 Tecniche di gestione	19
7.3 Punti chiave delle varie tecniche	20
7.3.1 Partizioni fisse	20
7.3.2 Partizioni variabili	20
7.3.3 Segmentazione	20
7.3.4 Segmentazione a domanda	21
7.3.5 Paginazione	21
7.3.6 Segmentazione paginata	23
7.4 Stati di un processo swapped	23
8 Gestione I/O	23
8.1 Gestione timer	27
8.2 Gestione dischi	27
8.2.1 Terminologia utilizzata	27
8.2.2 Schedulazione	28
9 File System	28
9.1 Stack di astrazione	28
9.2 Visione d'insieme	28
9.2.1 Struttura logica	28
9.2.2 Accesso	29
9.2.3 Organizzazione fisica	29
10 Protezione	30
11 Unix	31
11.1 Unix puro	31
11.1.1 Generalità	31
11.1.2 Processi	31
11.1.3 Memoria	32
11.1.4 Filesystem	32
11.2 Linux	33
11.2.1 Processi	33
References	34

0 Introduzione

0.1 Informazioni generali

Questi appunti, tratti dalle lezioni in aula del prof. Marco Avvenuti, sono da intendersi come appunti riassuntivi.

Questi appunti *non* sono stati revisionati dal prof. Marco Avvenuti. I testi ufficiali a cui fare riferimento per lo studio di questa disciplina sono elencati nella bibliografia.

Questi appunti potrebbero essere utili per il superamento dell'esame di Sistemi Operativi del corso di laurea triennale di Ingegneria Informatica, ma questo non implica che siano affidabili.

L'autore non si assume nessuna responsabilità sull'uso proprio, e specialmente improprio, che si vorrà fare di questi appunti.

0.2 Informazioni particolari

Questi appunti sono stati scritti, raccolti, ampliati e ricomposti da:

- giomba <giomba@live.it>

che ha fatto uso anche degli appunti di:

- F. Barbarulo <info@notestack.it>
- alex.sieni

Questi appunti sono abbastanza schematici; per una spiegazione esaustiva fare riferimento al testo ufficiale; per una spiegazione più dettagliata di questa, ma meno dettagliata del libro, fare riferimento agli appunti di F. Barbarulo.

Questi appunti non sono completi: facendo riferimento al testo principale [1], è assente il capitolo sulla protezione, mentre il capitolo su Unix è solo accennato; ma c'è una buona notizia.

Il codice sorgente di questi appunti è stato versionato con git e viene rilasciato gratuitamente sotto licenza GPL3: può essere consultato facendo esplicita richiesta all'autore, che invita a ampliare, correggere e terminare il lavoro da lui incompiuto.

0.3 Cinque

Non ve ne importa niente, ma in corrispondenza col superamento dell'esame di Sistemi Operativi, della scrittura finale di questa sezione e del rilascio definitivo di questi appunti in data 9 gennaio 2018, è morto il mio professore di italiano e latino del liceo.

Lui era un professore speciale, più umano della media, a tratti quasi un amico, che sapeva ampliare il rapporto professionale con gli studenti con qualcosa che dava una marcia in più nella crescita professionale e umana dell'individuo, qualcosa che andava di là dagli schemi tradizionali dell'insegnamento.

È merito suo se ho imparato *rosa, rosae, rosae, rosam, rosa, rosa*, ma anche se so che il fandango è un tipo di ballo spagnolo, se riesco a leggere il significato nascosto di Stand By Me¹, se ancora oggi chiamo i miei vecchi compagni del liceo.

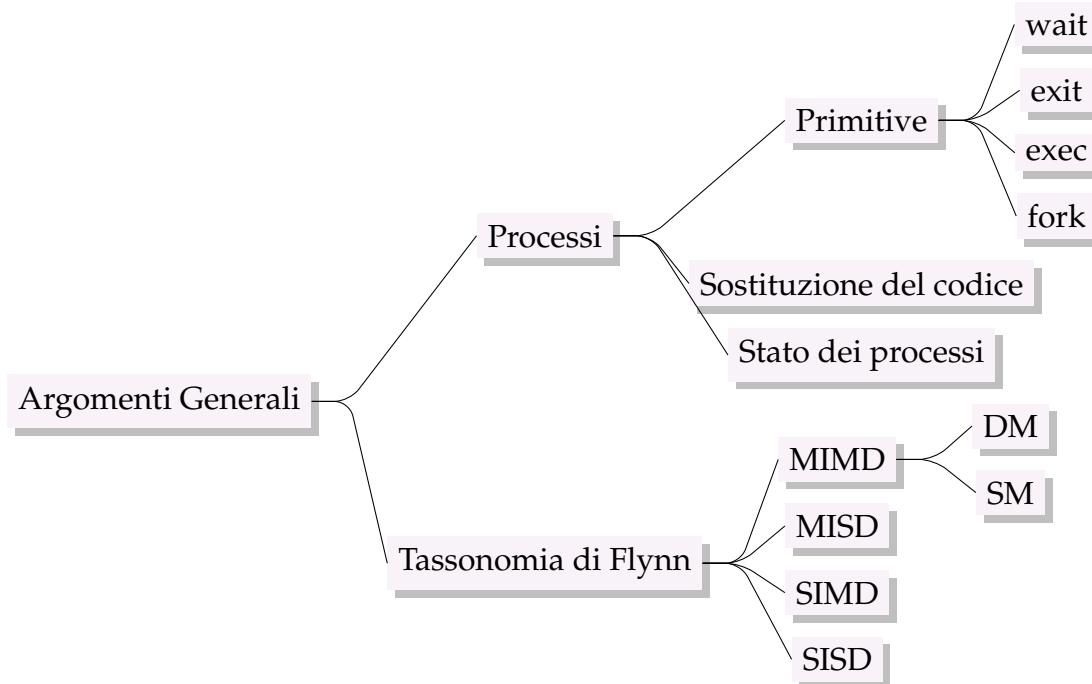
O capitano, mio capitano!

Requiescat.

¹Rob Reiner

1 Mappe

1.1 Argomenti generali



2 Tassonomia di Flynn

2.1 Terminologia utilizzata

- Instruction Processor (IP): elemento che esegue istruzioni
- Data Processor (DP): elemento che elabora dati
- Instruction Stream: flusso di istruzioni eseguite da un IP
- Data Stream: flusso di dati che vengono elaborati da un DP
- Instruction Memory (IM): contiene istruzioni
- Data memory (DM): contiene dati

Le memorie vengono rappresentate con una forma triangolare per evidenziare la presenza di una gerarchia [1, p. 25].

I flussi si distinguono in:

- *single stream*: è possibile identificare un unico flusso
- *multiple stream*: è possibile identificare più flussi

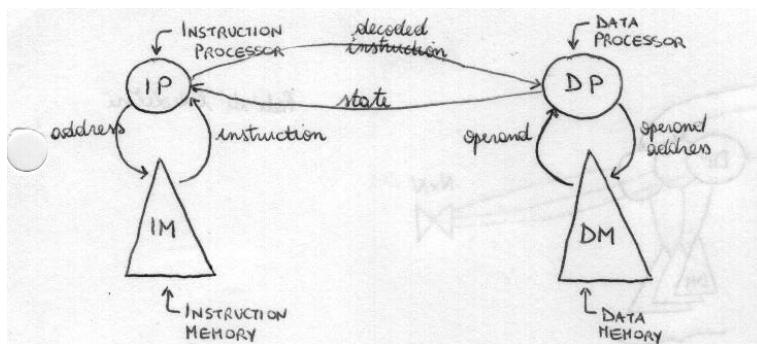
Le macchine possono essere inserite in quattro categorie:

	SD	MD
SI	SISD	SIMD
MI	MISD	MIMD

The diagram shows four boxes representing different categories:

- SD (Single Stream):**
 - SI (Single Instruction):** Shows a single IP connected to a single IM and a single DM. An arrow labeled "single instruction" points from the IP to the IM, and another arrow labeled "single data stream" points from the IP to the DM.
 - MI (Multiple Instructions):** Shows multiple IPs connected to a single IM and a single DM. Arrows labeled "multiple instructions" point from each IP to the IM, and an arrow labeled "multiple data streams" points from each IP to the DM.
- MD (Multiple Data Streams):**
 - SI (Single Instruction):** Shows a single IP connected to multiple DPs. Arrows labeled "single instruction" point from the IP to each DP, and an arrow labeled "multiple data streams" points from each DP to a common DM.
 - MI (Multiple Instructions):** Shows multiple IPs connected to multiple DPs, which then connect to a common DM. Arrows labeled "multiple instructions" point from each IP to its respective DP, and arrows labeled "multiple data streams" point from each DP to the DM.

2.2 SISD - Single instruction and single data streams



Le macchine in grado di eseguire un solo flusso di istruzioni possono comunque essere multiprogrammate, e i processi che vengono eseguiti sono detti *processi concorrenti* in quanto *concorrono* per l'utilizzo dell'unica unità di elaborazione disponibile.

Esempio: il computer monoprocessoresso di P. Corsini

- parte controllo (IP);
- parte operativa (DP);

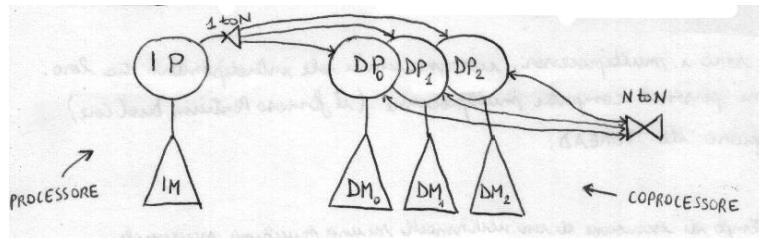
- registro di stato (FLAG);
- memoria (IM + DM)

2.3 MISD - Multiple instruction stream and single data stream

Non esistono macchine di questo tipo, sono un puro esercizio di stile.

Taluni ritengono macchine MISD le macchine dotate di pipeline [2, p. 209], in cui più istruzioni entrano nella pipeline, e attività diverse del ciclo macchina² necessario ad eseguirle, vengono effettuate parallelamente.

2.4 SIMD - Single instruction stream and multiple data stream



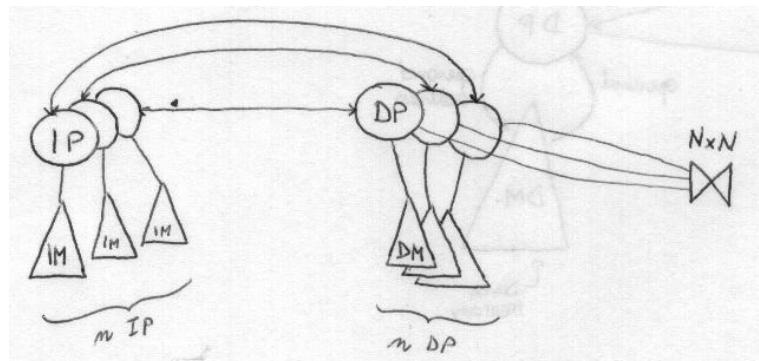
Si tratta di macchine dedicate che eseguono task CPU-bound. Tutti i coprocessori presenti eseguono la stessa operazione, ma su sottoinsiemi di dati differenti.

Ad esempio, le macchine SIMD possono lavorare in parallelo su piccole parti di una matrice più grande, sfruttandone le proprietà matematiche.

2.5 MIMD - Multiple instruction and multiple data streams

Le macchine MIMD possono essere ulteriormente suddivise in:

2.5.1 DM - Distributed memory

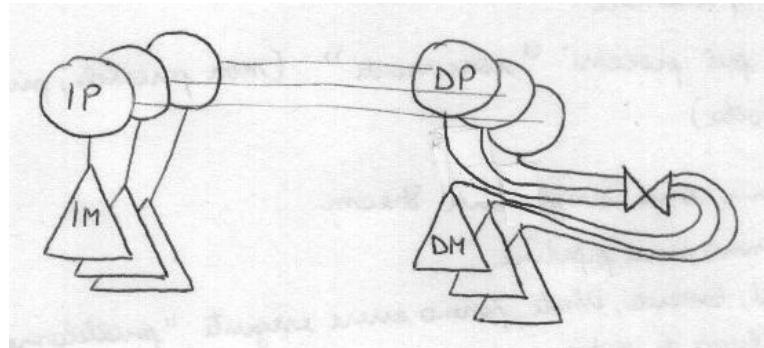


Macchine MIMD di tipo DM non sono altro che più macchine SISD collegate tra loro con una qualunque rete (es. Internet).

Le "singole" macchine SISD che compongono la macchina MIMD sono autosufficienti e del tutto indipendenti tra loro; il guasto di una macchina SISD non pregiudica il funzionamento del sistema (entro determinati limiti).

²fetch, decode, read, execute, write

2.5.2 SM - Shared memory



Macchine MIMD di tipo SM sono macchine multiprocessore. Tutta la memoria è condivisa e viene vista come un'unica memoria, e viene resa accessibile a qualunque DP.

Le macchine MIMD di tipo SM multiprogrammate eseguono processi detti *processi paralleli*, in quanto vengono eseguiti *parallelamente* sulle diverse unità di elaborazione.

A differenza delle macchine MIMD DM, le macchine MIMD SM non possono tollerare guasti ad una delle loro unità, in quanto interdipendenti tra loro.

3 Speed-Up

$$S = \frac{T_1}{T_N} < N$$

dove

- benchmark è un insieme di task da eseguire;
- T_1 è il tempo di esecuzione di un determinato benchmark su una macchina monoprocessore;
- T_N è il tempo di esecuzione dello stesso benchmark su una macchina multiprocessore con N processori

In un mondo ideale, $S = N$.

Tuttavia vi sono sempre alcuni task che non possono essere eseguiti sequenzialmente (non sono parallelizzabili), che abbassano lo speedup. Per tenere conto di questi task, si può utilizzare una definizione più esaustiva dello speedup, dato dalla legge di Amdahl.

Legge di Amdahl

$$S = \frac{T_1}{T_{seq} + \left(\frac{T_1 - T_{seq}}{N} \right)} < \frac{T_1}{T_{seq}}$$

4 Reti di interconnessione

Terminologia utilizzata:

- *Diametro*: massimo numero di hop necessario per far comunicare due host
- *Partizionamento*: suddivisione di una singola rete in più sottoreti indipendenti

Tabella riassuntiva:

Tipo	Esempio	Collegamenti per nodo	Points of Failure	Diametro	Scalabilità
Bus		1	1	–	no ($N \ll 100$)
Point to Point		2	1	$N - 1$	sì
Ring		2	2	$\frac{N}{2}$	sì
Mesh		> 2 (es 4)	–	$2(\sqrt{N} - 1)$	sì
Mesh toroidale		> 2 (es 4)	–	\sqrt{N}	sì
Tree		> 2 (es 3)	1	$2\log(N)$	sì

Alcune considerazioni:

- Il bus può essere pilotato da un solo dispositivo alla volta, detto master, in mutua esclusione;
- Nelle reti punto-punto, i nodi intermedi devono occuparsi anche delle connessioni dei nodi periferici;
- Negli alberi, la rottura di un collegamento è tanto più dannosa quanto più è vicino alla radice;
- Negli alberi, un collegamento è tanto più trafficato quanto più è vicino alla radice.

5 Schedulazione

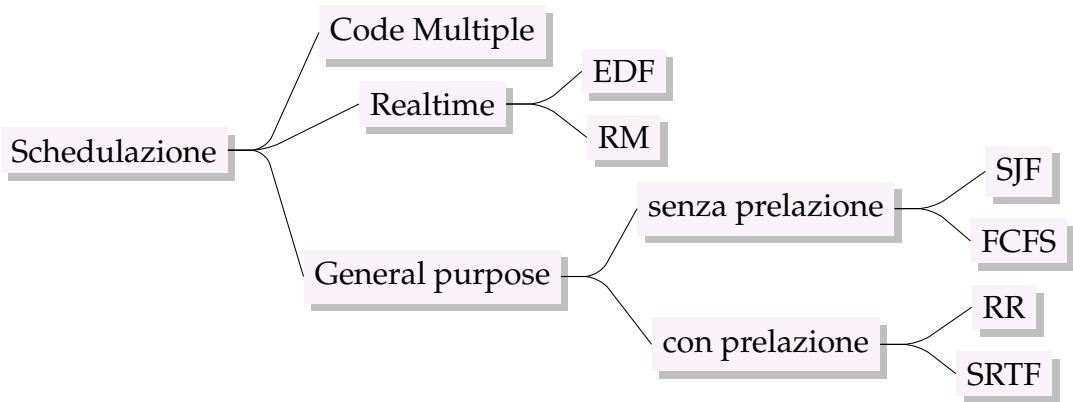
Un sistema di schedulazione è caratterizzato dai seguenti parametri:

- **utilizzazione della CPU:** percentuale di uso proficuo della CPU per unità di tempo;
- **turnaround time:** tempo medio di completamento di un processo, dall'istante in cui entra in coda pronti, all'istante in cui termina;
- **throughput rate:** numero medio di processi completati per unità di tempo; è l'inverso del *turnaround time*;

Per ogni processo è possibile calcolare i seguenti valori:

- **tempo di risposta:** intervallo temporale che intercorre tra l'ingresso di un processo in coda pronti, alla sua terminazione;

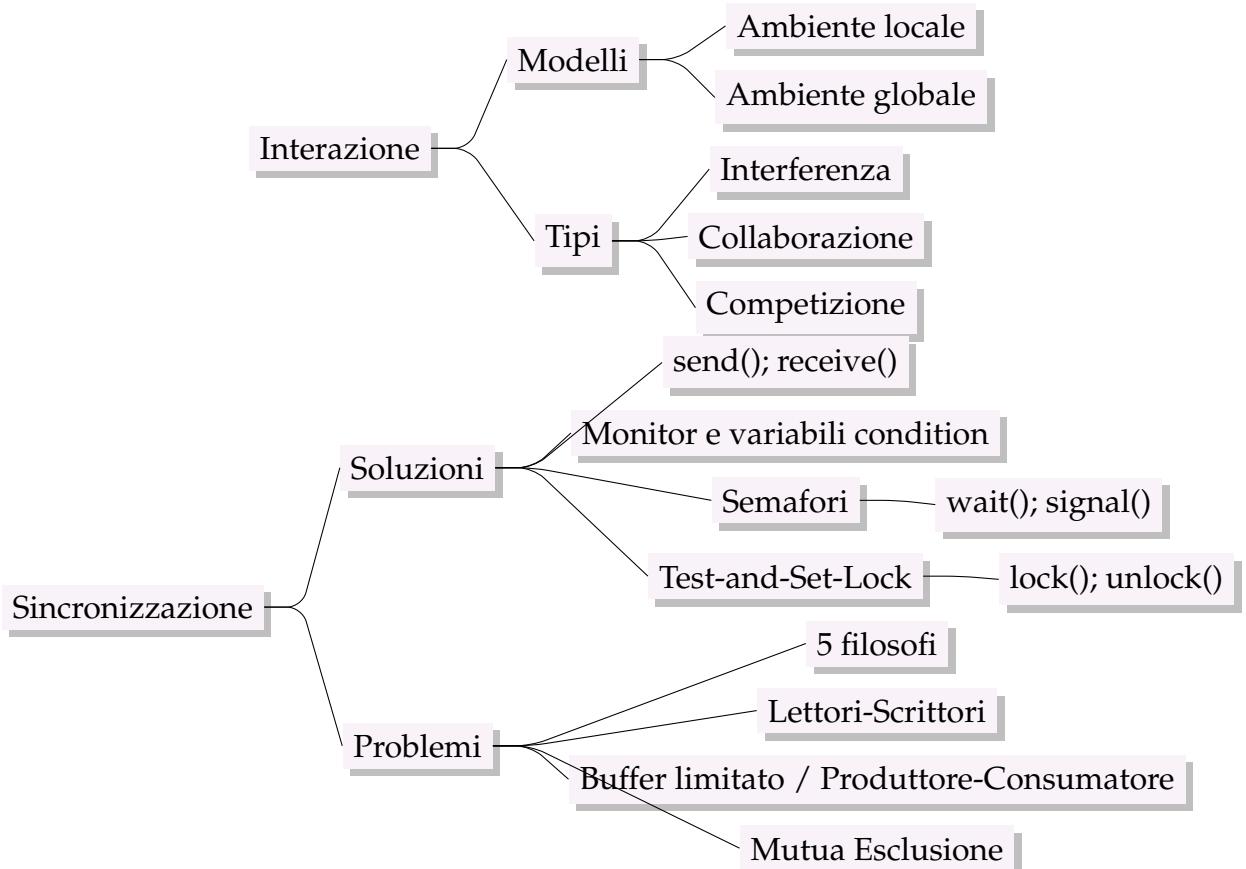
- **tempo di attesa**: intervallo temporale che intercorre tra l'ingresso di un processo in coda pronti, e l'inizio della sua esecuzione;

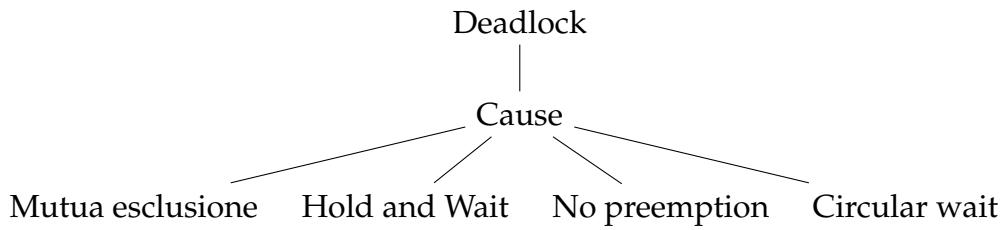


5.0.0.1 Condizione sufficiente per schedulabilità di n processi con Rate Monotonic

$$U \leq n(2^{\frac{1}{n}} - 1)$$

6 Sincronizzazione tra processi





6.1 Problemi

- 6.1.1 Problema del buffer limitato
- 6.1.2 Problema dei lettori-scrittori
- 6.1.3 Problema dei 5 filosofi

6.2 Soluzioni ai problemi di mutua esclusione e comunicazione

6.2.1 Test and Set Lock

6.2.1.1 Uso di Test and Set Lock

```

...
lock();
... sezione critica ...
unlock();
...

```

6.2.1.2 Implementazione di Test and Set Lock

```

lock:
    TSL x, %reg
    CMP $1, %reg
    JE lock
    RTS

```

```

unlock:
    ST $1, x
    RTS

```

6.2.2 Semafori: wait(); signal()

6.2.2.1 Uso dei semafori

Per la mutua esclusione

```

...
wait(semaforo);
... sezione critica ...
signal(semaforo);
...

```

Per la sincronizzazione

```
consumatore:  
...  
    wait(disponibile);  
    ... consuma ...  
    signal(consumato);  
...
```

```
produttore:  
...  
    wait(consumato);  
    ... produci ...  
    signal(disponibile);  
...
```

6.2.2.2 Implementazione di semafori

```
struct Semaphore {  
    int value;  
    PCB queue; /* PCB = Process Control Block struct */  
} s;  
  
wait(Semaphore s) {  
    if (s.value == 0) {  
        < blocca processo su s.queue >  
    }  
    else {  
        --s.value;  
    }  
}  
  
signal(Semaphore s) {  
    if (< ci sono processi bloccati su s.queue >) {  
        < estraie e sblocca un processo >  
    }  
    else {  
        ++s.value;  
    }  
}
```

6.2.3 Monitor e variabili condition

L'uso dei semafori fa affidamento sull'assenza di errori del programmatore utente e sulla sua onestà e buonsenso. È necessario introdurre dei costrutti sintattici a più alto livello per evitare situazioni spiacevoli: i *monitor* e le *variabili condition*. [3, p. 223]

6.2.3.1 Uso di un monitor Il monitor garantisce che un solo processo alla volta possa eseguire le funzioni definite al suo interno.

```

monitor un_nome {
    ... shared variables ...
    function F1(...) {
        ...
    }
    function Fn(...) {
        ...
    }
    init_code(...){
        ...
    }
}

```

6.2.3.2 Uso di variabili condition I processi usano le variabili condition per bloccarsi in attesa di un segnale da un altro processo.

Dichiarazione di una variabile condition:

```
condition x;
```

Attesa su una variabile: la `wait` è sempre bloccante.

```
x.wait();
```

Notifica a un processo in attesa: la `signal` risveglia un processo (se ce n'è uno in coda), altrimenti non fa niente.

```
x.signal();
```

Quando si notifica ad un processo che può continuare la sua esecuzione, possiamo adottare due politiche:

- **signal and wait** risveglia il processo bloccato e mette in attesa il processo che esegue la signal;
- **signal and continue** risveglia il processo bloccato, ma, il processo che esegue la signal, prima di cedergli l'esecuzione, finisce di fare le sue operazioni;

6.2.3.3 Implementazione di un monitor tramite semafori

Codice di inizializzazione

```
x_count = 0;
x_sem.value = 0;
```

Monitor

```

wait(mutex);
... corpo della procedura ...
if (next_count > 0) {
    signal(next);
}
else {
    signal(mutex);
}

```

Variabili condition

```
x.wait():
    x_count++;
    if (next_count > 0) {
        signal(next);
    }
    else {
        signal(mutex);
    }
    wait(x_sem);
    x_count--;

x.signal():
    if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
    }
```

6.3 Deadlock (stallo)

Condizioni necessarie (ma non sufficienti) per il deadlock con risorse in unica istanza.

- mutua esclusione (mutual exclusion)
- possesso e attesa (hold and wait)
- assenza di revoca

Condizione sufficiente:

- attesa ciclica (circular wait)

Nel caso in cui le risorse siano presenti in più istanze, l'attesa ciclica viene declassata a condizione necessaria.

6.3.1 Prevenzione statica

Consiste nell'impedire, in fase di compilazione, che si verifichi una delle condizioni necessarie. Queste tecniche non sono efficienti, perché contraddicono la buona norma modulare del sistema.

- **Possesso e attesa** Per risolverla, si impedisce ai processi di fare richieste di risorse annidate tra loro, oppure si obbliga un processo a richiedere le risorse di cui ha bisogno tutte contemporaneamente prima di iniziare la sua esecuzione.
- **Assenza di revoca** Per risolverla, si permette al sistema di revocare le risorse già assegnate ad un processo, e di farlo ripartire solo quando saranno nuovamente disponibili sia la nuova che ha chiesto sia tutte quelle che aveva anche prima della revoca.

- **Attesa circolare** Per risolverla, si associa un numero d'ordine a ogni risorsa. Le risorse vengono assegnate ai processi in ordine crescente. Un processo non può richiedere una risorsa con numero d'ordine inferiore ad una risorsa che gli è già stata assegnata.

6.3.2 Prevenzione dinamica

6.3.2.1 Algoritmo del banchiere Lo stato del sistema viene rappresentato per mezzo delle seguenti strutture dati. Ogni qualvolta un processo richiede una risorsa al sistema, questi, prima di assegnarla, controlla se il nuovo stato è uno *stato sicuro*. Uno stato è sicuro se esiste una sequenza di esecuzione dei processi tale che tutti possano terminare in un tempo finito. Si noti che un processo può terminare solo dopo che richiesto tutte le risorse di cui ha bisogno, e che, al termine, rilascia tutte le risorse di cui era entrato in possesso.

- $\text{need}[P_i]$ = numero di risorse che il processo P_i deve ancora richiedere per completare la sua esecuzione
- $\text{assigned}[P_i]$ = numero di risorse assegnate al processo P_i
- m = numero di risorse disponibili

Esempi

Supponiamo di conoscere a priori il numero massimo di risorse richieste da ogni processo e il numero di risorse disponibili nel sistema, indicati nella colonna in neretto. Supponiamo inoltre che il sistema si trovi nella condizione iniziale contrassegnata dalle colonne in corsivo.

Supponiamo che processo P_2 richieda una risorsa: se questa risorsa viene assegnata, il sistema andrà a trovarsi nello spiacevole *stato insicuro* contrassegnato in rosso, in cui potenzialmente nessun processo può terminare la sua esecuzione, e se qualcuno non rilascia una delle sue risorse, il sistema non ne ha più a disposizione. Pertanto, al processo P_2 non viene assegnata la risorsa, e viene messo in attesa.

Process	Max	Assigned	Needed	Assigned	Needed
P_0	3	2	1	2	1
P_1	2	1	1	1	1
P_2	4	2	2	3	1
Available	6	1		0	

Table 1: l'algoritmo del banchiere constata che il sistema andrà a trovarsi in uno stato insicuro

Adesso invece il processo P_0 richiede una risorsa: di nuovo il sistema viene a trovarsi nella condizione di non avere più risorse disponibili ma, a differenza del caso precedente, esiste almeno un processo che può terminare entro un tempo finito: P_0 . Quando P_0 sarà terminato, il sistema avrà nuovamente 3 risorse disponibili, e ne può assegnare 2 a P_2 , che può terminare, e 1 a P_1 , che può anch'esso terminare.

Quindi, siccome assegnare la risorsa a P_0 porta in uno *stato sicuro*, la risorsa può essere immediatamente assegnata.

Process	Max	A	N	A	N	A	N	A	N	A	N	A	N
P_0	3	2	1	3	0	—	—	—	—	—	—	—	—
P_1	2	1	1	1	1	1	1	1	1	2	0	—	—
P_2	4	2	2	2	2	2	2	4	0	4	0	—	—
Available	6	1	0	3	1	0	6						

Table 2: l'algoritmo del banchiere constata che il sistema andrà a trovarsi in uno stato sicuro

Come si può facilmente intuire, l'algoritmo del banchiere non è applicabile nel contesto di un moderno sistema operativo in cui non è possibile conoscere a priori né il numero di risorse disponibili, né il numero di risorse di cui un processo potrebbe avere bisogno.

6.3.2.2 Matrice degli stati sicuri In una moderna tecnica di prevenzione dinamica, possiamo visualizzare gli stati di un sistema come una matrice multidimensionale, con una dimensione per processo. Per i limiti imposti da questo foglio bidimensionale, consideriamo un esempio con due processi: la Figura 1 riporta sull'asse delle ascisse il progresso dell'esecuzione di P_1 e sull'asse delle ordinate il progresso dell'esecuzione di P_2 .

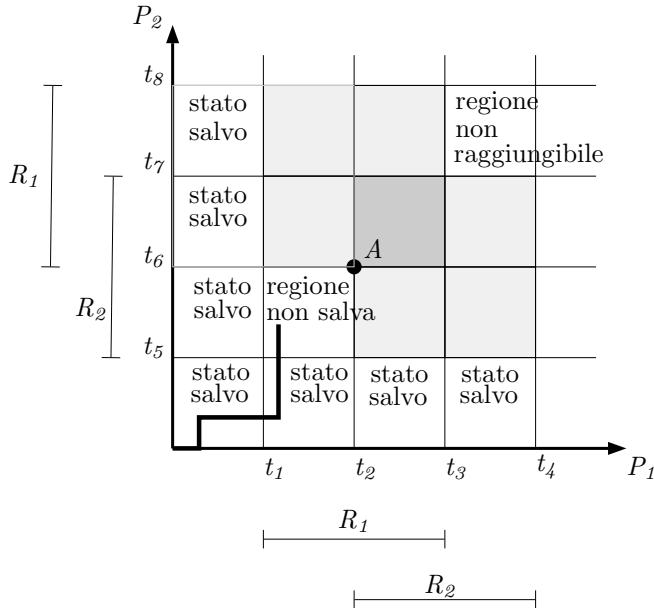


Figure 1: Esempio di prevenzione dinamica per mezzo di matrice degli stati sicuri

Agli istanti t_1 e t_2 il processo P_1 richiede rispettivamente l'utilizzo di R_1 e R_2 , mentre a t_3 e t_4 tali risorse vengono rilasciate. Analogamente fa P_2 con le medesime risorse agli istanti t_5 , t_6 e t_7, t_8 .

La traiettoria rappresenta il cammino di esecuzione dei due processi che parte dall'origine e prosegue alternando l'esecuzione di P_1 e P_2 . Il punto A rappresenta una situazione di blocco critico in quanto corrisponde alla situazione in cui i due processi, possedendo ciascuno una risorsa, chiedono di poter utilizzare l'altra, bloccandosi indefiniteamente.

Scopo del sistema è fare in modo che la traiettoria di esecuzione proceda attraversando sempre *stati salvi* senza mai entrare in stati *non salvi*.

Nella Figura 1, qualunque sia il proseguimento dell'esecuzione, il sistema è destinato a andare in stallo.

6.3.3 Individuazione di blocchi critici e recupero

Sottosezione altrimenti nota come *deadlock detection and recovery*.

6.3.3.1 Deadlock detection Periodicamente il sistema ricerca la presenza di condizioni sufficienti allo stallo, analizzando eventuali cicli di dipendenze tra processi.

La ricerca di cicli in grafi è molto dispendiosa, pertanto questi algoritmi vengono mandati in esecuzione raramente.

6.3.3.2 Deadlock recovery Esistono due tecniche:

- terminazione forzata di tutti i processi del ciclo
- terminazione di un processo per volta finché non vengono liberate abbastanza risorse per permettere al sistema di ritornare a funzionare

7 Gestione della memoria

7.1 Concetti generali

Analogie con CPU:

- risorsa necessaria implicitamente per un processo (non su richiesta)
- virtualizzazione

Differenze con CPU:

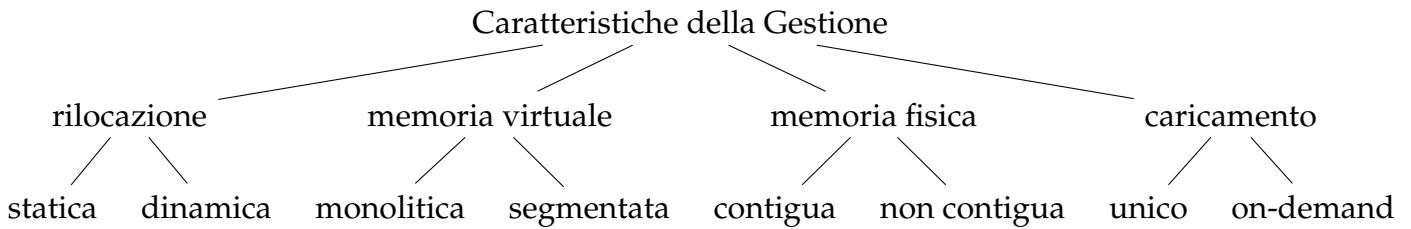
- contesto del processo *vs* area di swap
- la memoria può ospitare contemporaneamente più processi (richiede accortezze di protezione)
- opportunità di condivisione di tutta o parte della risorsa

7.1.1 Terminologia utilizzata

- **funzione di rilocazione**, funzione $y = f(x)$ che traduce l'indirizzo virtuale x nell'indirizzo fisico y ;
- **MMU**, Memory Management Unit, dispositivo hardware atto alla traduzione di indirizzi, da virtuali a fisici, per mezzo di una apposita funzione di rilocazione;
- **swap-in, swap-out**, operazione di caricamento di un programma (o parte di esso) dalla memoria di massa verso la memoria centrale, e viceversa;
- **TLB**, Translation Lookaside Buffer, dispositivo hardware di caching associativo per velocizzare le operazioni più frequenti di traduzione di indirizzi, sfruttando i principi di località spaziale e temporale³

³Vedi [2]

7.1.2 Caratteristiche della Gestione



- Rilocazione

- Rilocazione statica: la traduzione degli indirizzi viene effettuata una sola volta all'atto del caricamento del programma da parte del caricatore (rilocante) nella memoria fisica; in caso di *swap-out* e successivo *swap-in*, il programma deve essere ricaricato nella stessa locazione fisica;
- Rilocazione dinamica: la traduzione degli indirizzi viene effettuata ogni volta che viene generato un indirizzo (virtuale), per mezzo di un apposito supporto hardware (MMU);

- Organizzazione della memoria virtuale

- Memoria Virtuale Monolitica: lo spazio di indirizzamento virtuale di un programma viene organizzato come un unico blocco, in cui è possibile trovare le varie sezioni (logiche) di codice, dati e stack una di seguito all'altra (*contigui*);
- Memoria Virtuale Segmentata: lo spazio di indirizzamento virtuale di un programma viene logicamente diviso in più *segmenti*, che rispecchiano la struttura immaginata dal programmatore (funzioni, strutture dati), alle quali è possibile assegnare diversi permessi (lettura, scrittura); più segmenti diversi possono così essere allocati in più regioni fisiche non contigue in qualunque ordine;

- Organizzazione della memoria fisica

- Memoria Fisica Contigua: la sezione virtuale (o le singole sezioni virtuali) di un programma vengono caricate in memoria in modo *contiguo*;
- Memoria Fisica Non Contigua: blocchi virtuali di un programma, anche se *contigui*, possono essere caricati nella memoria fisica in pagine diverse in modo non *contiguo*;

- Caricamento dei blocchi del programma

- Caricamento Unico: tutti i blocchi di un programma vengono caricati in memoria all'atto della creazione del processo, anche se alcuni potrebbero non essere mai utilizzati;
- Caricamento On-Demand: i blocchi di un programma vengono caricati in memoria a tempo di esecuzione di un processo, e solo se sono richiesti (in seguito a generazione di *fault*);

Diverse combinazioni di queste modalità di rilocazione, caricamento e organizzazione dello spazio fisico e virtuale danno luogo a diversi schemi di gestione della memoria virtuale. Delle $2^4 = 16$ diverse potenziali combinazioni, alcune sono impossibili, altre inutili o non significative, e sono riassunte nella Figura 2 e nella Figura 3.



Figure 2: Combinazioni significative delle varie tecniche di gestione della memoria

		Spazio virtuale monolitico		Spazio virtuale segmentato				
		Allocazione fisica contigua	Allocazione fisica non contigua	Allocazione fisica contigua	Allocazione fisica non contigua			
Rilocazione statica	Caricam. Unico	Partizioni statiche o variabili		Partizioni multiple		Caricam. Unico	Rilocazione statica	
	Caricam. On-demand					Caricam. On-demand		
Rilocazione dinamica	Caricam. Unico		Paginazione	Segmentazione		Caricam. Unico	Rilocazione dinamica	
	Caricam. On-demand		Paginazione on-demand	Segmentazione on-demand	Segmentazione con paginazione	Caricam. On-demand		
		Allocazione fisica contigua	Allocazione fisica non contigua	Allocazione fisica contigua	Allocazione fisica non contigua			
		Spazio virtuale monolitico		Spazio virtuale segmentato				

Figure 3: Combinazioni delle varie tecniche di gestione della memoria

7.2 Tecniche di gestione

Tecnica	Rilocazione	Allocazione	Spazio virtuale	Caricamento
Partizioni fisse e variabili	statica	contigua	monolitico	unico
Partizioni multiple	statica	contigua	segmentato	unico
Segmentazione	dinamica	contigua	segmentato	unico
Segmentazione on-demand	dinamica	contigua	segmentato	on-demand
Paginazione	dinamica	non contigua	monolitico	unico
Paginazione on-demand	dinamica	non contigua	monolitico	on-demand
Segmentazione con paginazione	dinamica	non contigua	segmentato	on-demand

7.3 Punti chiave delle varie tecniche

7.3.1 Partizioni fisse

- Frammentazione interna: per poter essere rilocato in una partizione, la dimensione di un programma in memoria deve essere minore (o al più uguale) alla dimensione della partizione in cui si intende rilocarlo. Ne consegue che spesso rimane dello spazio inutilizzato al termine della partizione, fenomeno che prende il nome di *frammentazione interna*.
- Rotazione round robin: periodicamente il sistema operativo revoca la partizione della memoria di un processo e la assegna ad un altro, secondo la politica round robin.

7.3.2 Partizioni variabili

- Frammentazione esterna: le partizioni variabili eliminano completamente il problema della frammentazione interna, ma introducono quello della *frammentazione esterna*: quando un processo termina, infatti, la sua partizione di memoria torna disponibile, e lascia un "vuoto" tra partizioni occupate. Pur essendo la memoria totale "vuota" sufficiente a contenere l'immagine di un nuovo processo, la sua non contiguità rende praticamente impossibile il suo sfruttamento.
- Partizioni libere adiacenti: possono essere compattate in un'unica partizione libera più grande.
- Ricompattamento: lo spostamento fisico di tutte le partizioni al fine di ricompattare lo spazio occupato e quello libero è molto oneroso e, in questo caso particolare, non è applicabile in quanto trattasi di una tecnica con rilocazione statica.
- Lista: il sistema operativo mantiene una lista ordinata delle partizioni:
 - schema *best-fit*: lista ordinata per dimensione crescente delle partizioni: in fase di caricamento di un programma, la prima partizione libera di dimensione sufficiente a contenere il programma è anche quella che "calza meglio" (lo spazio che avanza prima della successiva partizione adiacente è minimo). In fase di scaricamento, la ricerca di partizioni adiacenti per il loro ricompattamento richiede però lo scorrimento di tutta la lista.
 - schema *first-fit*: lista ordinata per indirizzi crescenti delle partizioni: in fase di caricamento di un programma, la prima partizione libera di dimensione sufficiente a contenere il programma è quella che viene scelta, anche se non è quella che calza meglio. In fase di scaricamento, la ricerca di partizioni adiacenti è banale.

7.3.3 Segmentazione

- Segmenti: almeno tre segmenti, per codice, dati e stack, banalmente discernibili in hardware in stato di *fetch*, *ld/st/mov* e *push/pop*. Architetture più evolute prevedono la possibilità di utilizzare più segmenti, tramite la generazione di indirizzi del tipo $x = \langle\text{segment}\rangle, \langle\text{offset}\rangle$ e la consultazione di una *tavella dei segmenti*.
- Eccezioni:
 - Protezione: segmento non appartenente al processo (errore);

- Protezione: offset al di fuori del segmento (errore o richiesta di allocazione);
- Protezione: flag R, W, X;
- TLB: Translation Lookaside Buffer, cache associativa diventa necessaria per sopperire all'inefficienza del meccanismo.
- Condivisione: possibile, a condizione che segmenti semanticamente affini abbiano lo stesso numero;

7.3.4 Segmentazione a domanda

- Presenza: flag P. Un segmento non presente genera un'eccezione di tipo *segment fault*.

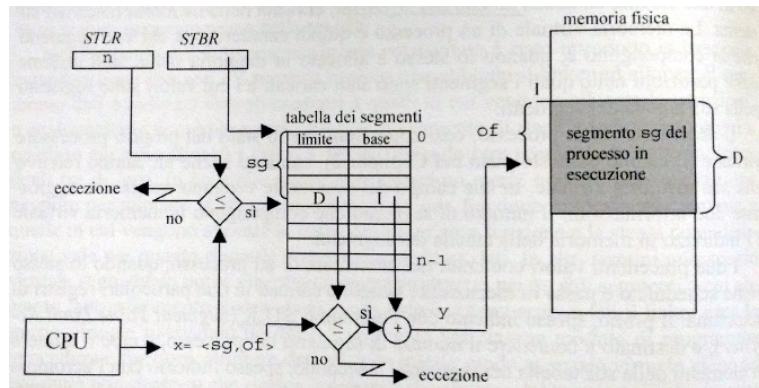


Figure 4: Rilocazione dinamica nella segmentazione

7.3.5 Paginazione

- Terminologia: *pagina* o *pagina virtuale*: regione in cui è diviso lo spazio virtuale; *frame* o *pagina fisica*: regione fisica in cui è divisa la memoria fisica; Le pagine sono mappate nei frame. Pagine consecutive non devono necessariamente essere mappate in frame consecutivi.
- Protezione: possibile ma meno significativa, in quanto la pagina, a differenza del segmento, riflette solo la struttura fisica della memoria e non la struttura semantica del programma;
- Pagine residenti: necessarie per trasferimenti DMA dall'IO o per funzioni vitali del sistema operativo;
- Trashing: situazione di funzionamento inefficiente in cui le stesse pagine vengono continuamente swapped *in* e *out*.
- Condivisione: possibile, a condizione che gli indirizzi virtuali utilizzati da processi diversi uguali (puntatori).

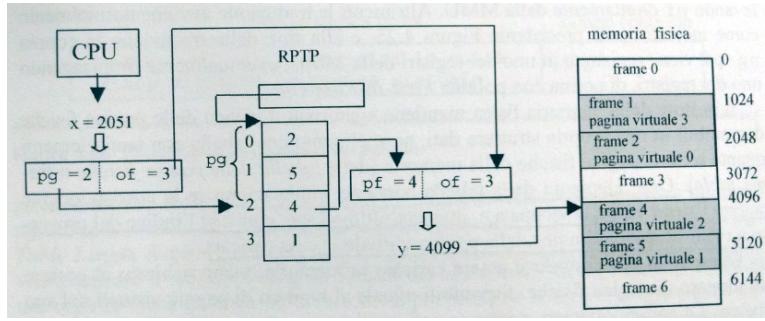


Figure 5: Rilocazione dinamica nella paginazione

7.3.5.1 Politiche di caricamento Alla creazione di un processo possono essere caricate le sue pagine:

- tutte le pagine: ammesso di avere sufficiente memoria fisica a disposizione, in questo modo un fault può essere generato solo se il processo richiede l'allocazione di nuova memoria in maniera dinamica durante la sua esecuzione;
- alcune pagine: possono essere caricate solo le pagine indispensabili per l'"avvio" del processo, così le altre saranno caricate solo se questo ne fa richiesta;
- nessuna pagina: si fa affidamento solo sui fault, che verranno generati frequentemente nei primi istanti in cui il processo inizia ad essere eseguito;

7.3.5.2 Algoritmi di rimpiazzamento

- Ottimo: tengo un puntatore alla pagina che verrà utilizzata tra più tempo nel futuro (requisiti hardware: Mago Merlino vegente⁴);
- FIFO: i descrittori delle pagine fisiche frame sono organizzati in una lista circolare. Un puntatore punta sempre alla pagina fisica successiva all'ultima rimpiazzata, cioè alla pagina fisica successiva a quella più recente più recente, cioè alla pagina che è in memoria da più tempo.
- LRU, Least Recently Used: contrassegno ogni pagina fisica con il timestamp del suo ultimo accesso, e rimpiazzo la pagina che non viene usata da più tempo. Svantaggi: costoso in termini di tempo macchina e memoria, complesso da realizzare in hardware.
- Second Chance (orologio): l'MMU setta i bit di accesso (A, access) e modifica (D, dirty) alle pagine fisiche. Queste vengono organizzate in una lista circolare. Un puntatore punta sempre alla pagina successiva all'ultima rimpiazzata, cioè alla pagina che si trova in memoria da più tempo. In caso di rimpiazzamento, tuttavia, scorro la lista finché non giungo alla prima pagina fisica con A = 0, ponendo il bit A = 0 a tutte le pagine che incontro via via che scorro. Se tutte le pagine hanno il bit A = 0, l'algoritmo rispecchia automaticamente l'algoritmo FIFO.

⁴<https://www.youtube.com/watch?v=Icg6R5QWYwk>

7.3.5.3 Tipi di rimpiazzamento

- Locale: le pagine vittime vengono scelte solo tra le pagine del processo che ha generato il page fault; ad ogni processo viene assegnato un *budget* alla sua creazione, e le pagine gli vengono revocate se lo supera;
- Globale: le pagine vittime vengono scelte tra le pagine di tutti i processi;

7.3.6 Segmentazione paginata

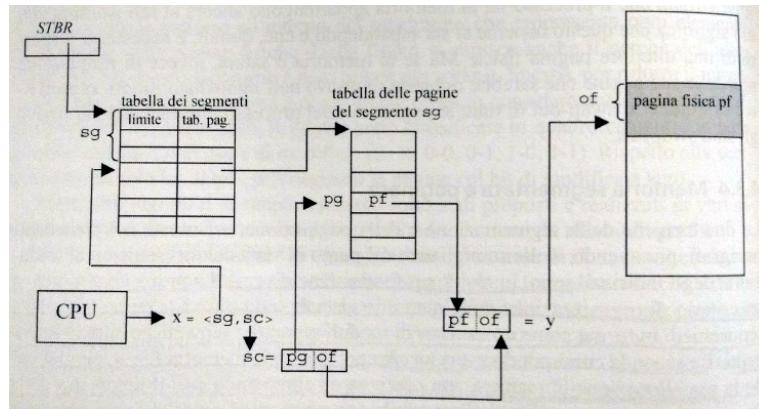


Figure 6: Rilocazione dinamica nella segmentazione paginata

7.4 Stati di un processo swapped

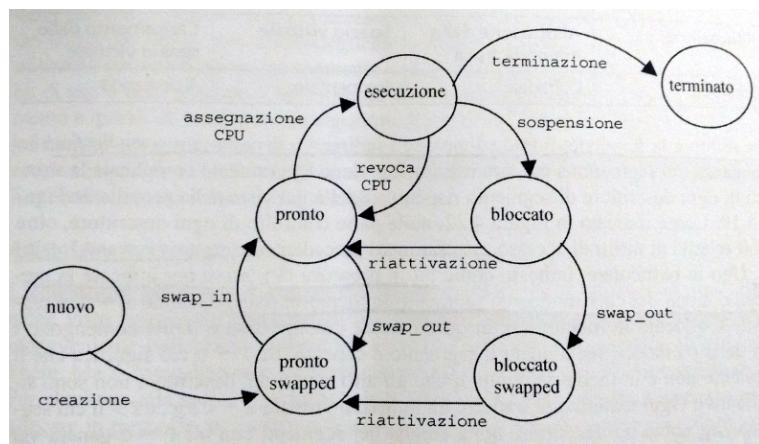


Figure 7: Stati di un processo swapped con caricamento unico

8 Gestione I/O

Questa sezione è fatta male. Studia il materiale dell'esame di Calcolatori Elettronici.

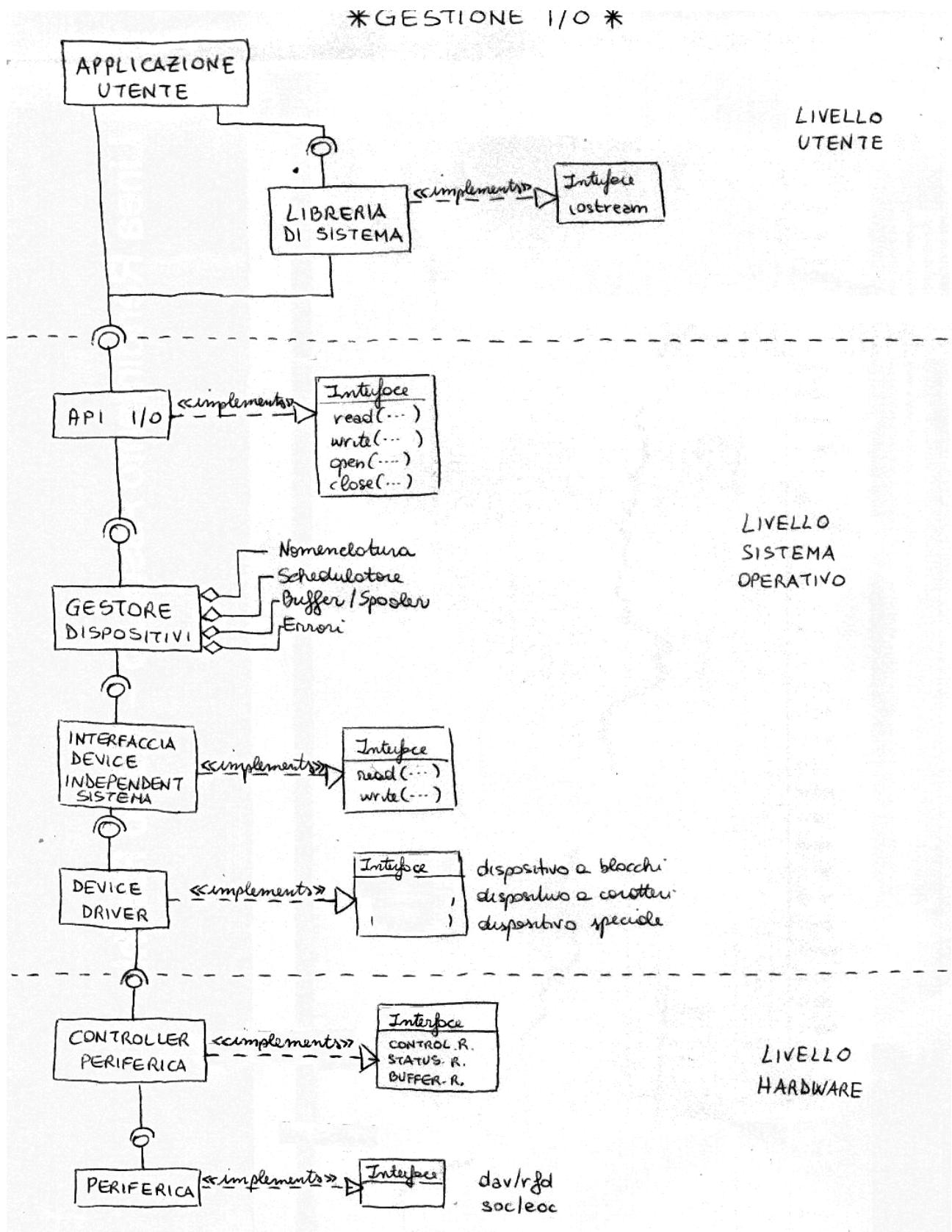


Figure 8: Organizzazione logica del sottosistema di I/O tramite notazione approssimativa e inappropriata

Descrittore di un dispositivo:

```

struct DescrittoreDispositivo {
    void* puntStatus;
    void* puntControl;
    void* puntData;
    semaforo datoDisponibile;
    char* buffer;
    int contatore;
    int esito;
};

}

```

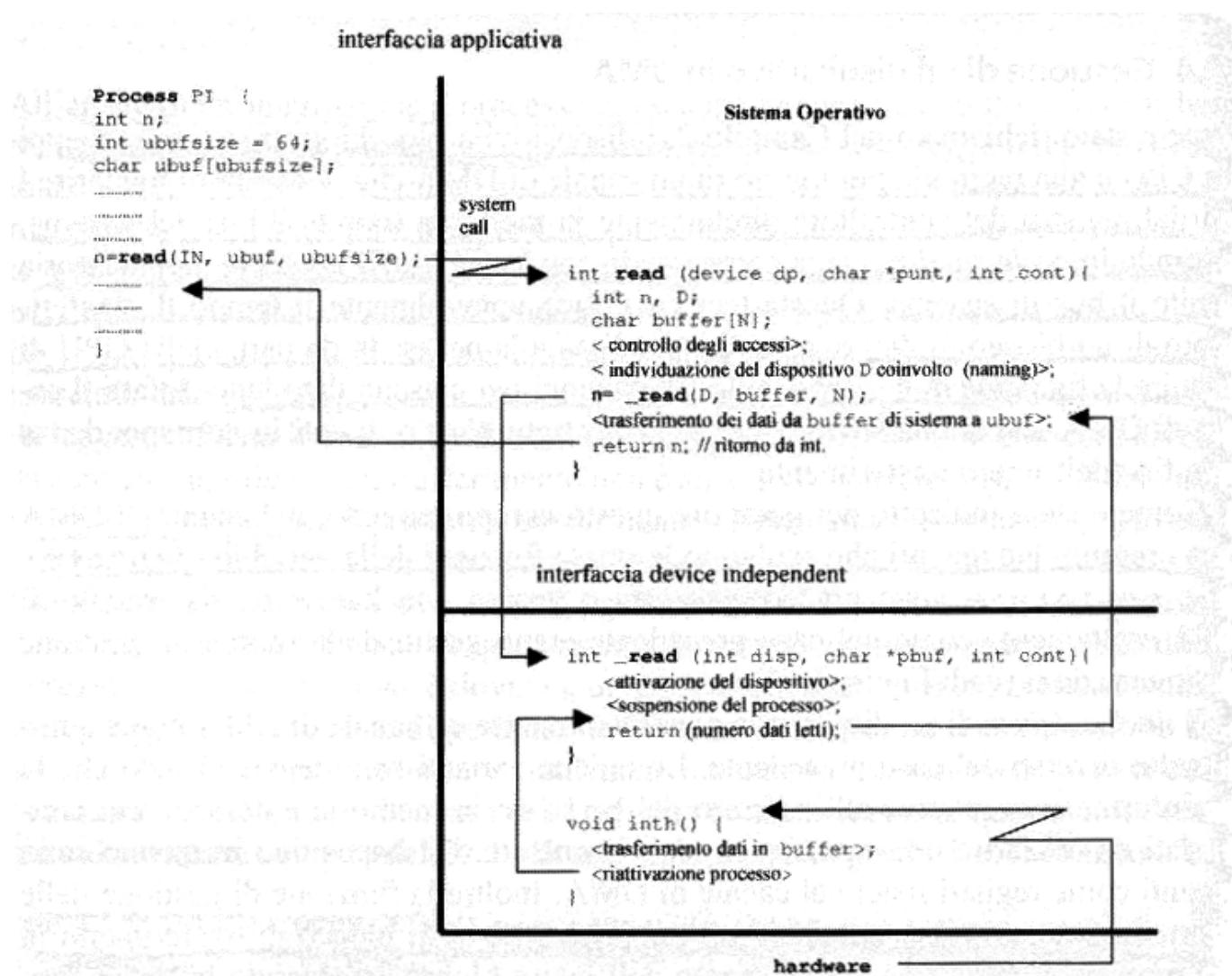


Figure 9: Flusso di controllo durante l'esecuzione di una primitiva di I/O con interruzione

- Device independent
 - Nomenclatura
 - Protezione
 - Buffering
 - Buffer utente / sistema

- Sincronizzare produttore / consumatore
- Leggere meno dati del blocco
- Realizzare cache software
- Buffer di sistema residenti
- o Errori
 - Errori recuperabili / Irrecuperabili
- o Allocazione dinamica dispositivi
- o Spooling

Seguono delle versioni molto approssimative di funzioni di gestione dell'I/O: non c'è scritto quasi nulla, perché dipendono dal particolare tipo di hardware, dal tipo di trasferimento (a interruzione o in DMA), e da numerosi altri fattori.

System call read(), chiamata dall'applicazione:

```
int read(file devicename, char* ubuffer, int n) {
    // Codice eseguito con privilegi di sistema
    ... Controllo permessi di accesso al dispositivo 'devicename' ...
    D = f(devicename); // Traduzione del nome
    ... Garantisco mutua esclusione ...
    r = _read(D, sbuffer, n);
    memcpy(ubuffer, sbuffer, n);
    return r;
}
```

System call _read(), chiamata dal sistema:

```
int _read(int deviceid, char* sbuffer, int n) {
    descriptor[deviceid].buffer = sbuffer;
    descriptor[deviceid].count = n;
    ... Abilito dispositivo a mandare interruzioni ...
    ... Avvio il trasferimento dal dispositivo (bit start = 1) ...
    wait(descriptor[deviceid].eot);
    if (descriptor[deviceid].result == <tutto ok>) {
        return n - descriptor[deviceid].count;
    }
    else
        return -1;
}
```

Interrupt handler inth(), chiamato dal processo esterno:

```
void inth() {
    ... Disabilito dispositivo a mandare interruzioni ...
    c = get(descriptor[deviceid].addr_data); // Leggo dato dal registro dati
    *descriptor[deviceid].buffer = c;
    ++descriptor[deviceid].buffer;
    if (--descriptor[deviceid].count == 0) {
        ... Imposto descriptor[deviceid].result ...
    }
}
```

```

    signal(descriptor[deviceid].eot);
}
else {
    ... Riabilito dispositivo ...
}
}

```

8.1 Gestione timer

Versione ibrida Calcolatori/Sistemi.

System call delay(), chiamata dall'applicazione:

```

struct delay_s { // Lista di delay_s
    int n;
    condition awake;
    struct delay_s* next;
};

int delay(int n) {
    +---+      +---+      +---+
    |   | --> |   | --> |   |
    +---+      +---+      +---+
    ... ricerca di delay_s D tale che n == SUM_precedenti(delay_s.n) ...
    ... se non esiste, creazione di D e inserimento ...

    D.awake.wait();
}

```

Interrupt handler inth(), chiamato dal processo esterno:

```

void inth() {
    D = <lista>; // primo elemento
    if (--D.n == 0) {
        D.awake.signal_all();
        <lista> = D.next;
        free(D);
    }
}

```

8.2 Gestione dischi

8.2.1 Terminologia utilizzata

- TF = TA + TT; tempo medio
- TA = TS + RL; tempo di accesso
- TS; tempo di seek
- RL; rotation latency
- TT; tempo di trasferimento del settore

8.2.2 Schedulazione

- FCFS, First Come First Served, altamente inefficiente;
- SSTF, Shortest Seek Time First, molto efficiente, ma soffre di starvation;
- SCAN, scansione continua del disco dalla prima all'ultima traccia, miglior compromesso tra FCFS e SSTF, ha buone prestazioni e non soffre di starvation;

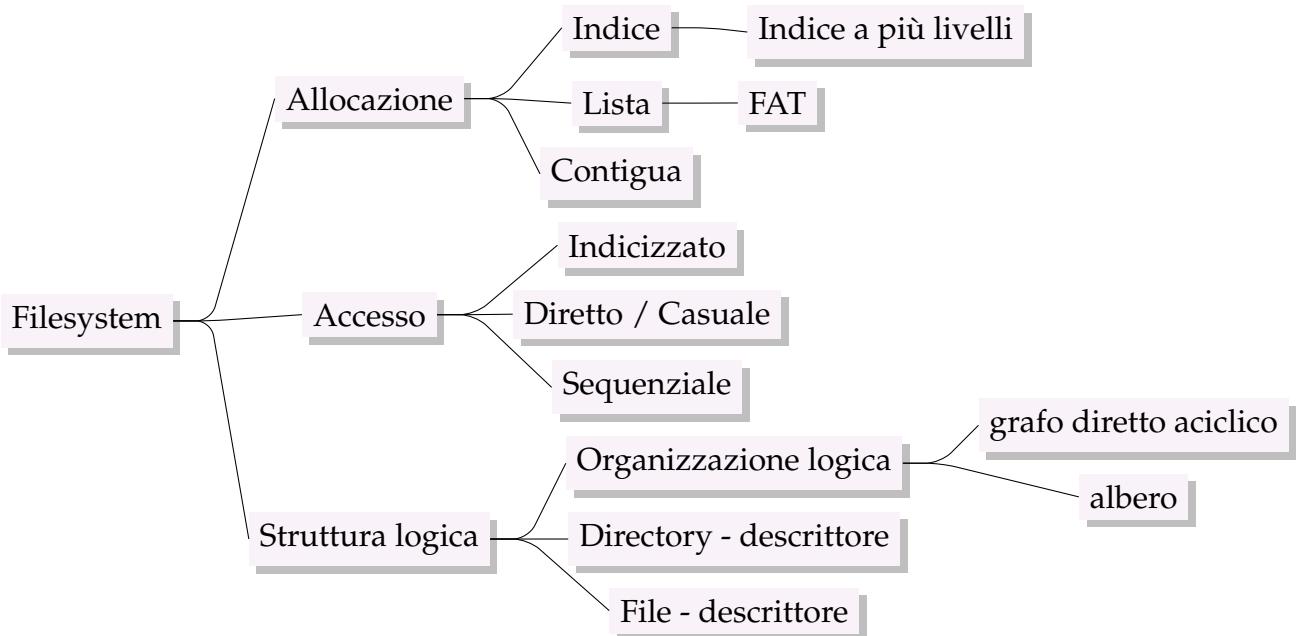
9 File System

9.1 Stack di astrazione



Table 3: Stack di astrazione

9.2 Visione d'insieme



9.2.1 Struttura logica

Terminologia utilizzata:

- **file**, sequenza di informazioni organizzate in record;
- **record**, unità logica minima di informazione all'interno di un file (es. *linea, byte, ...*);

Organizzazione del filesystem:

- albero;
- grafo diretto aciclico;

Descrittore di file:

```
struct file_descriptor_s {
    string nome;
    uint dimensione;
    sector indirizzo;
};
```

Il descrittore può contenere anche altre informazioni di utilità, quali il tipo, la data di creazione o di ultima modifica del file, e informazioni di protezione, quali utente e gruppo proprietario e relativi permessi di accesso.

9.2.2 Accesso

I descrittori possono essere memorizzati:

- nel file rappresentante la directory di appartenenza:
 - il sistema è distribuito;
 - il linking necessita di costosi barbatrucchi;⁵
- in un'area dedicata del disco; la directory contiene i puntatori ai descrittori di file:
 - il sistema è centralizzato in un'area dedicata del disco;
 - il linking è banale;

Il sistema può fornire una (o più) delle seguenti interfacce:

- Accesso sequenziale: l'accesso al record i -esimo richiede l'accesso a tutti i precedenti $i - 1$ record.
- Accesso diretto: l'accesso ai record avviene in modo indicizzato, numericamente, in modo analogo ad un vettore.
- Accesso indicizzato: l'accesso ai record avviene tramite l'associazione di un'etichetta al record; è necessaria una tabella associativa all'interno del file; questo approccio è tipico dei DBMS⁶.

9.2.3 Organizzazione fisica

Un dispositivo hardware viene astratto con un dispositivo virtuale, ad es. un hard disk viene astratto come un insieme di blocchi da 512 byte ciascuno, indicizzati dalla tupla CHS⁷.

⁵workaround, espedienti creativi

⁶Database Management System

⁷Cylinder Head Sector

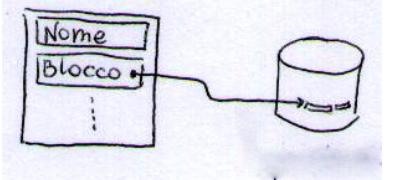
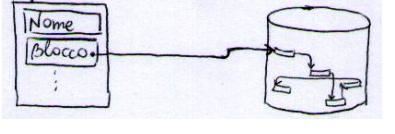
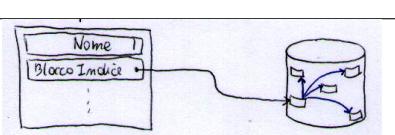
Allocazione contigua	
Allocazione a lista, eventualmente ridondata tramite FAT	
Allocazione a indice, eventualmente a più livelli	

Table 4: Tipi di allocazione fisica

9.2.3.1 FAT, File Allocation Table Struttura ridondante per filesystem con allocazione fisica a lista: la tabella contiene un'entrata per ogni settore del disco, che associa, ad ogni settore, il suo successivo.

10 Protezione

- **soggetto:** parte attiva di un sistema che accede a degli oggetti per conto degli utenti, es. processi;
 - soggetto = (processo, dominio di protezione)
- **oggetto:** parte passiva di un sistema, es. risorse fisiche o logiche;

Politiche di protezione:

- **DAC**, Discretionary Access Control, i diritti di accesso agli oggetti vengono definiti dai loro proprietari;
- **RBAC**, Role-based Access Control, i diritti di accesso agli oggetti vengono definiti da un'entità amministratrice e vengono assegnati a gruppi di utenti che rappresentano degli specifici ruoli all'interno di una organizzazione;
- **MAC**, Mandatory Access Control, i diritti di accesso agli oggetti vengono definiti da un'entità amministratrice centrale, per esigenze di elevata sicurezza (militare, sanitaria);

Principio del minimo privilegio: a un soggetto vengono garantiti i privilegi minimi indispensabili per lo svolgimento del suo compito.

11 Unix

11.1 Unix puro

11.1.1 Generalità

Sistema monolitico.

11.1.2 Processi

- processi pesanti, no thread;
- spazio di indirizzamento dati privato;
- comunicazione tramite segnali;
- spazio di indirizzamento codice condiviso: il codice viene detto *rientrante*;

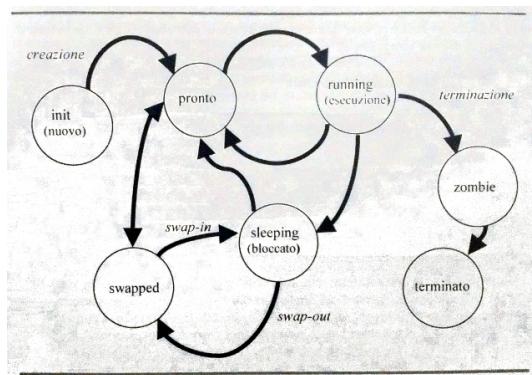


Figure 10: Stati di un processo su Unix

Un processo diventa *zombie* dopo che ha chiamato la *system call* di terminazione `exit()`, ma le sue strutture dati non possono essere distrutte in quanto la sua immagine è ancora necessaria: questo avviene sempre quando il processo figlio intende terminare prima del processo padre, quest'ultimo non è ancora terminato, e potrebbe quindi richiedere di conoscerne lo stato di terminazione del figlio.

11.1.2.1 PCB, Process Control Block

- **Process Structure**, residente in memoria centrale, contiene informazioni vitali di cui il sistema deve sempre essere a conoscenza;
- **User Structure**, swappabile, contiene informazioni specifiche del processo che sono necessarie solo quando il processo è in esecuzione; può essere swapata quando il processo si trova nello stato swapped;

Process Structure:

- PID
- Stato
- Riferimento a area dati e stack
- Riferimento alla *text structure*
- Parent PID
- Priorità
- Next process in coda
- Riferimento alla *user structure*

User Structure:

- Contesto registri generali CPU
- File allocati
- Segnali non ancora gestiti
- Proprietario, Gruppo
- Ambiente: cwd^a, argc, argv

^aCurrent Working Directory

11.1.2.2 System call per la gestione dei processi

- int fork(void);
- int execl(char* path, char* arg0, char* arg1, ..., char* argN, (char*)0);

11.1.2.3 Schedulazione

- Priorità
 - priorità ∈ [−20, +19];
 - −20 è la priorità più alta, 19 quella più bassa;
 - i processi utente hanno priorità positiva, i processi sistema hanno priorità negativa;
 - un processo può diminuire volontariamente la sua priorità, ma non aumentarla; solo root può aumentare la priorità dei processi;⁸
- Esiste una coda di processi per ogni livello di priorità;
- I processi dentro una singola coda sono gestiti *round robin*;

11.1.3 Memoria

Segmentazione paginata, on demand.

11.1.4 Filesystem

- il descrittore di file viene detto *inode*;
- i descrittori sono memorizzati in una regione dedicata del disco detta *inode-list*;
- ogni descrittore è identificato da un *inode-number*;
- un record ha la dimensione di *un* byte;

⁸perché root è root, root è onnipotente, root può fare tutto, anche uccidere init⁹

- le directory contengono puntatori agli inodes;
- allocazione dei file contigua/indirizzata 13-15 ?? TODO

11.2 Linux

11.2.1 Processi

- processi pesanti e threads;

References

- [1] Paolo Ancilotti et al. *Sistemi operativi*. 2nd ed. McGraw-Hill, 2008.
- [2] Graziano Frosini and Giuseppe Lettieri. *Architettura dei Calcolatori*. Vol. 2. Pisa University Press, 2013.
- [3] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. 2013.