Sistemi Operativi

Ivan A. Arena

Marzo 2021

Testo di appunti del corso di *Sistemi Operativi* della Facoltà di Informatica dell'Università degli Studi di Padova.

Indice

1 Introduzione **2**

2 Programmi e processi **2**
2.1 Risorse 2
2.2 Stati di avanzamento dei processi 3
2.3 Gestione dei processi 3

3 Sincronizzazione dei processi **4**
3.1 Race condition e regione critica 4
3.2 Problema produttore-consumatore 5
3.3 Semafori 6
3.4 Monitor 7
3.5 Message Passing 8
3.6 Barriere 8
3.7 Problema dei filosofi 8
3.8 Problema dei lettori e degli scrittori 10
3.9 Problema del barbiere sonnolento 11

4 Ordinamento dei processi **12**
4.1 Politiche e meccanismi 13
4.2 Classificazione di sistemi 13
4.3 Politiche di ordinamento 13
4.3.1 Sistemi a lotti 13
4.3.2 Sistemi interattivi 14
4.3.3 Sistemi in tempo reale 14

5 Gestione della memoria **14**
5.1 Frammentazione 14
5.2 Rilocazione e protezione 15
5.3 Swapping 15
5.4 Strutture di gestione 15
5.5 Memoria virtuale 15
5.6 Paginazione 15
5.6.1 Strutture 16
5.6.2 Rimpiazzo 16
5.7 Segmentazione 17

6 File System **17**
6.1 Struttura logica di file 17
6.1.1 Modalità di accesso 17
6.1.2 Classificazione di file 18
6.2 Struttura di directory 18
6.3 Realizzazione del file system 18
6.3.1 Realizzazione di file 18
6.3.2 Gestione dei blocchi liberi 19
6.4 Integrità del File System 19

1 Introduzione

Un **sistema operativo** (**S/O**) è un insieme di utilità progettate per gestire le risorse fisiche e logiche di un elaboratore fornendo all'utente un'astrazione più semplice e potente, costituita da una **macchina virtuale**, un ambiente in cui la memoria è virtualizzata ed è possibile eseguire applicazioni senza particolari conoscenze.

2 Programmi e processi

Si dice **programma** un codice che esegue una o più istruzioni; un programma in esecuzione è detto, invece, **processo**. In un sistema operativo i processi (*utente* e *di sistema*) avanzano **concorrentemente** (non simultaneamente) secondo diverse politiche di ordinamento, garanti della *fairness*, decise nello **scheduling**.

2.1 Risorse

Una **risorsa** è un qualsiasi elemento fisico o logico necessario alla creazione, esecuzione ed avanzamento di processi e può essere:

- Durevole o consumabile;
- Ad accesso atomico o divisibile;
- Ad accesso individuale o molteplice;

Le risorse del tipo I/O sono gestite da un programma specifico, il **BIOS**.

2.2 Stati di avanzamento dei processi

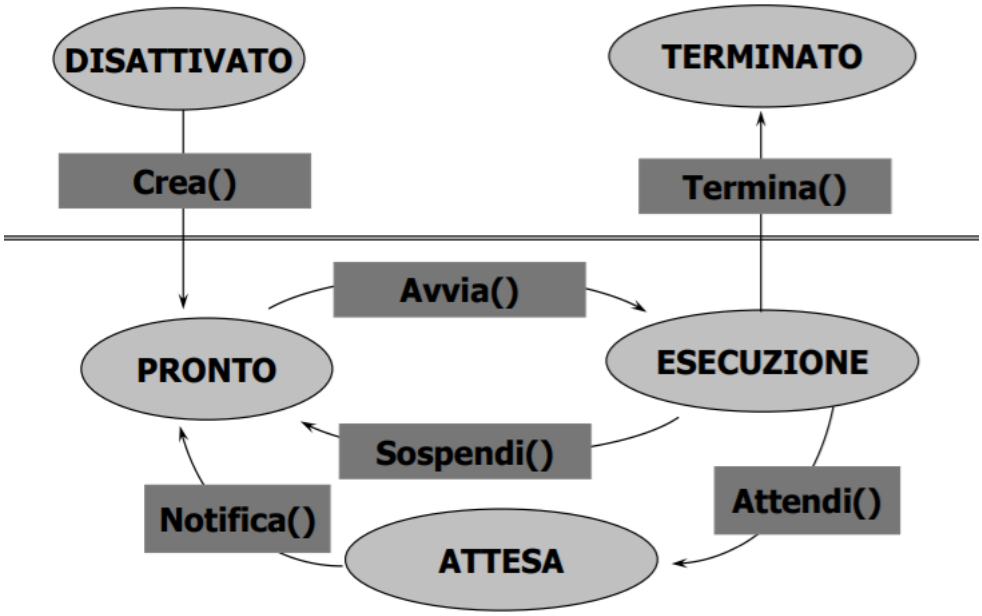

Figura 1: Stati di avanzamento dei processi.

Come illustrato in Fig. 1, un processo **disattivato** viene caricato in memoria tramite una chiamata di sistema che crea una struttura dati detta **Process Control Block** (**PCB**), entra in stato di **pronto**, va in **esecuzione** in base alla priorità attribuitagli, dove può **terminare**, essere **prerilasciato** (o **sospeso**) per dare spazio ad un altro processo, o messo in **attesa**.

2.3 Gestione dei processi

La gestione dei processi è affidata al nucleo del sistema operativo, il **kernel**. Il componente che gestisce gli scambi tra i processi in esecuzione è il **dispatcher**. I processi in stato di pronto sono accodati in una struttura detta **lista dei pronti** (**ready list**), spesso del tipo **First-Come-First-Served** (**FCFS**). I processi possono essere **CPU-bound**, attività dalla durata molto lunga, o **I/O-bound**, comprendenti attività di breve durata sulla CPU ed altre di I/O molto lunghe. Quest'ultimi sono penalizzati dalla tecnica FCFS, motivo per cui vengono messi a disposizione di ogni processo uguali **quanti** di tempo (**round-robin**) e vengono istituite code diverse in base alla priorità o alla categoria dei vari processi.

3 Sincronizzazione dei processi

Spesso, i processi condividono risorse e servono meccanismi di **sincronizzazione di accesso** per gestirne la condivisione.

Esempio: Siano A e B due processi che condividono la variabile $x = 10$; A incrementa x di 2, B decrementa x di 4. A e B leggono x uno dopo l'altro, leggendo, quindi, entrambi 10. Allora A scriverà 12, mentre B scriverà 6. Il risultato desiderato era invece $8(10 + 2 - 4)$.

3.1 Race condition e regione critica

Quando, come nel caso di cui sopra, il risultato finale dell'esecuzione di processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguite le loro istruzioni si verifica una **race condition**, in una regione di codice detta **regione** (o **sezione**) **critica**. Una soluzione al problema della sincronizzazione dei processi è ammissibile se soddisfa le seguenti quattro condizioni:

- Garantire accesso esclusivo (**mutua esclusione**);
- Garantire attesa finita;
- Non fare assunzioni sull'ambiente di esecuzione;
- Non subire condizionamenti dai processi esterni alla sezione critica.

Un altro problema tipico è quello dello stallo (come vedremo nella sottosezione successiva) bloccante (**deadlock**) o non-bloccante (**starvation**). Le condizioni necessarie e sufficienti affinché si verifichi uno stallo sono:

- Accesso esclusivo ad una risorsa condivisa;
- Accumulo di risorse;
- Inibizione di prerilascio;
- Condizione di attesa circolare.

N.B.: *Necessarie* e *sufficienti* significa che basta impedire il verificarsi di anche una sola di queste per impedire lo stallo.

Per affrontare uno stallo possiamo optare, principalmente, per tre strategie: prevenzione (a tempo di esecuzione o prima), riconoscimento e recupero, **indifferenza** (quella più semplice ed efficace).

3.2 Problema produttore-consumatore

Il problema del **produttore-consumatore** nasce in due casi principali:

- Inizia il *consumatore*, legge `count == 0` ma viene prerilasciato prima di eseguire l'istruzione `sleep()`; a quel punto, viene eseguito il *produttore*, che incrementa `count` (portandolo ad `1`) ma non sveglia il *consumatore*, poiché non addormentato, allora, quando il produttore verrà prerilasciato, il *consumatore* si addormenterà e la condizione `count == 1` non sarà mai vera, poiché il *produttore* continuerà ad incrementare `count` fino a `count == N`, quando si addormenterà, causando un deadlock;
- Analogamente, inizia il *produttore* e produce fino a `count == N` ma viene prerilasciato prima di eseguire l'istruzione `sleep()`; allora, il *consumatore* decrementerà `count` fino ad `N-1` e non sveglierà il *produttore*, poiché non addormentato, e consumerà tutti gli elementi del buffer per poi addormentarsi, causando un deadlock.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figura 2: Problema produttore-consumatore.

3.3 Semafori

Una possibile soluzione a problemi come quello del produttore-consumatore coinvolge l'uso di una variabile atomica detta **semaforo**, **binario** (assume solo valori booleani) o **contatore** (consente più accessi condivisi).

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figura 3: Soluzione al problema produttore-consumatore mediante l'uso dei semafori.

3.4 Monitor

L'uso dei semafori da utente è, però, rischioso, perciò viene affidato ad una struttura di variabili e sottoprogrammi detta **monitor**, che definisce la regione critica e agisce da compilatore. Solo i suoi sottoprogrammi possono modificare le variabili interne ad un monitor. Tali strutture, oltre a garantire la mutua esclusione, impiegano due procedure, `wait` e `signal`, rispettivamente per forzare l'attesa del chiamante ed il risveglio del processo in attesa. Inoltre, non sono utilizzabili per lo scambio di informazioni tra elaboratori.

3.5 Message Passing

Un'altra possibilità per sincronizzare i processi è offerta dal **message passing** (**scambio messaggi**), che permette, al contrario dei monitor, la sincronizzazione tra dispositivi distinti ed impiega le funzioni base `send(destination, &msg)` e `receive(source, &msg)`.

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figura 4: Soluzione al problema produttore-consumatore mediante lo scambio messaggi

3.6 Barriere

Le **barriere** sono uno strumento che permette di sincronizzare gruppi di processi bloccandoli finché non la raggiungono tutti.

3.7 Problema dei filosofi

Ad un tavolo circolare sono seduti 5 filosofi, che alternano momenti in cui pensano a momenti in cui mangiano. Per mangiare, ogni filosofo necessita di due posate, quella alla sua destra e quella alla sua sinistra. Se tutti i filosofi cominciano a mangiare prendendo la prima posata ogni filosofo rimarrà senza la seconda, in attesa circolare, causando un deadlock.

```
// Soluzione al problema dei filosofi mediante l'uso di semafori.
```

```
int semaforo f[i] = 1;

// Per evitare il deadlock inseriamo un filosofo mancino, che
    prende prima la posata sinistra e poi quella destra
Filosofo(i) {
    while(1) {
        <pensa>
        if(i == X) { // Filosofo mancino X
            P(f[(i+1) % N]);
            P(f[i]);
        } else {
            P(f[i]);
            P(f[(i+1) % N]);
        }
        <mangia>
        V(f[i]);
        V(f[(i+1) % N]);
    }
}
```

```
Monitor Tavolo{
   boolean fork_used[5] = false; // forchette numerate da 0 a 4
   condition filosofo[5]; // se lo vogliamo fare in java, questa la dobbiamo
                                    togliere

   raccogli(int n){
      while(fork_used[n] || fork_used[(n+1)%5])
         filosofo[n].wait();
      fork_used[n] = true;
      fork_used[(n+1)%5] = true;
   }
   // in java dovevi aggiungere:
   // (synchronized)
   deposita(int n){
      fork_used[n] = false;
      fork_used[(n+1)%5] = false;
      filosofo[n].notify();   // se lo voglio fare in java devo togliere
                                                queste due "filosofo" e sostituire con
                                                notifyall()
      filosofo[(n+1)%5].notify();
   }
}
Filosofo(i){
   while (true){
      <pensa>
      Tavolo.raccogli(i);
      <mangia>
      Tavolo.deposita(i);
   }
}
```

Figura 5: Soluzione al problema dei filosofi mediante l'uso di monitor.

3.8 Problema dei lettori e degli scrittori

Un grande database deve essere letto e scritto. Più lettori possono accedervi contemporaneamente mentre gli scrittori possono solo quando non ci sono lettori.

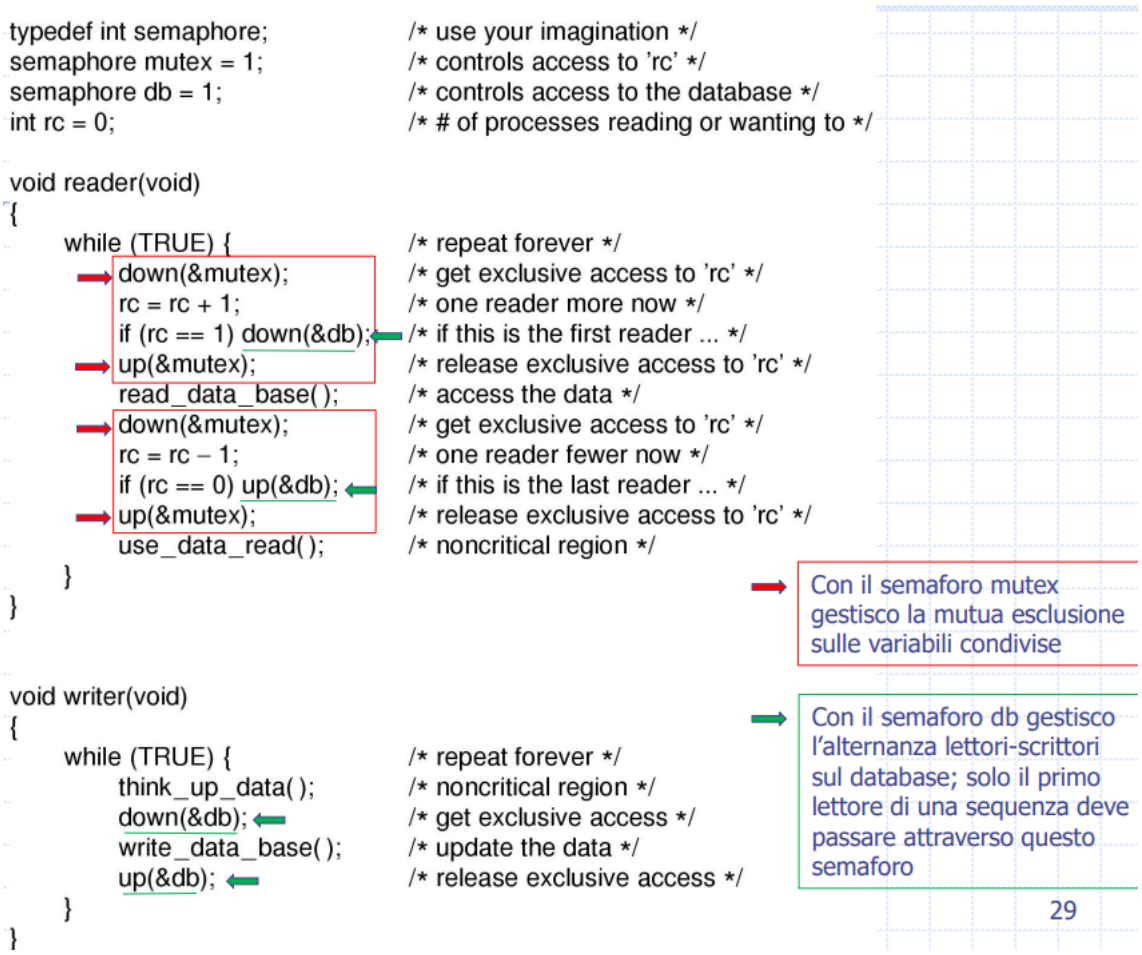

Figura 6: Soluzione al problema dei lettori e degli scrittori mediante l'uso di semafori.

3.9 Problema del barbiere sonnolento

Un barbiere dorme sulla poltrona se non ci sono clienti. Il primo cliente sveglia il barbiere e quelli successivi si siedono, su un numero limitato di sedie, in attesa che il barbiere finisca.

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting - 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Figura 7: Soluzione al problema del barbiere sonnolento mediante l'uso di semafori.

4 Ordinamento dei processi

Le informazioni sulla priorità di un certo processo sono contenute nel suo PCB (Process Control Block). Nella coda dei pronti troviamo solitamente un puntatore al PCB di ogni processo in coda. Una decisione di scheduling è necessaria alla creazione di un processo, alla terminazione, al blocco o all'interruzione I/O. I processi possono essere alternati con **scambio a prerilascio** (necessita di clock), tramite un meccanismo esterno, o con **scambio cooperativo** (il processo decide da solo).

4.1 Politiche e meccanismi

I **meccanismi** sono le sezioni del S/O che mettono in pratica le scelte di ordinamento e di gestione dei processi e risiedono nel nucleo; le **politiche** sono, invece, determinate fuori dal nucleo, nello spazio applicazioni.

4.2 Classificazione di sistemi

I sistemi operativi si dividono in tre classi generali:

- **Sistemi a lotti** (*batch*): ordinamento predeterminato e lavori poco urgenti e di lunga durata; prerilascio non necessario; **obiettivi delle politiche:** massima rapidità per singolo lavoro (***turn-around***);
- **Sistemi interattivi**: grande varietà di attività, prerilascio essenziale; **obiettivi delle politiche:** tempo di risposta (rispetto alla percezione dell'utente);
- **Sistemi a tempo reale**: lavori brevi ma urgenti, prerilascio possibile; **obiettivi delle politiche:** rispetto delle scadenze temporali (***deadline***) e predicibilità.

Le politiche di ordinamento dovrebbero rispettare tre cose: **equità** (*fairness*), **coerenza** (*enforcement*) e **bilanciamento**.

4.3 Politiche di ordinamento

Importante: Lo **scheduler** fissa la politica di ordinamento, il **dispatcher** ne attua le scelte.

- **Tempo di risposta** = Tempo trascorso dall'entrata in coda fino all'inizio della prima esecuzione;
- **Tempo di attesa** = Tempo passato in coda;
- **Tempo di esecuzione** = Durata effettiva del lavoro (senza contare le attese).
- **Tempo di turn-around** = Tempo di attesa + Tempo di esecuzione;

4.3.1 Sistemi a lotti

- **FCFS** (**First Come First Served**): senza prerilascio né priorità, basso utilizzo delle risorse;
- **SJF**: senza prerilascio, richiede conoscenze dei tempi di esecuzione richiesti, non è equo con lavori non presenti dall'inizio;
- **SRTN** (**Shortest Remaining Time Next**): variante di SJF con prerilascio, tiene conto dei nuovi processi quando arrivano ed esegue quello più veloce a completare.

N.B.: In genere, si parla di **lavori** quando operiamo senza prerilascio e di **processi** quando operiamo con prerilascio.

4.3.2 Sistemi interattivi

- **OQ** (**Ordinamento a Quanti**, es.: Round Robin (RR)): con prerilascio ma senza priorità;
- **OQP** (**Ordinamento a Quanti con Priorità**): con prerilascio e priorità, quanti diversi per livello di priorità;
- **GP** (**con Garanzia per Processo**): con prerilascio e con promessa una data quantità di tempo di esecuzione ($1/n - processi$), esegue prima il lavoro maggiormente penalizzato rispetto alla promessa;
- **SG** (**Senza Garanzia**): con prerilascio e priorità a, ogni processo riceve più o meno numeri, segue un'estrazione periodica (più numeri = più probabilità di vittoria);
- **GU** (**con Garanzia per Utente**): Come GP ma con garanzia riferita a ciascun utente ($1/n - utenti$).

4.3.3 Sistemi in tempo reale

- **Modello semplice** (***cyclic executive***): processi periodici con caratteristiche note, un ciclo maggiore (*major cycle*) è suddiviso in N cicli minori (*minor cycles*) di durata fissa che racchiudono sottosequenze di processi;
- **Ordinamento a priorità fissa**: preferibilmente con prerilascio a priorità assegnata secondo il periodo (priorità maggiore per periodo più breve);
- **Calcolo del tempo di risposta**;

5 Gestione della memoria

La componente del S/O che si occupa di soddisfare le esigenze di memoria dei processi è il **gestore della memoria**. Per i sistemi **monoprogrammati** (quelli che eseguono un solo processo alla volta) l'unica scelta progettuale rilevante è decidere dove allocare la memoria del S/O.

5.1 Frammentazione

La **frammentazione** è lo spreco di memoria e può essere **interna**, quando la memoria è divisa in blocchi di uguali dimensioni ed alcuni vengono riempiti solo in parte, o **esterna**, quando la memoria è divisa, invece, in blocchi di dimensione variabile e rimangono delle aree di memoria non copribili da blocchi.

5.2 Rilocazione e protezione

La **rilocazione** è l'interpretazione degli indirizzi emessi da un processo in relazione alla sua collocazione corrente in memoria; la **protezione** assicura che ogni processo operi soltanto nello spazio di memoria ad esso permissibile.

5.3 Swapping

Lo **swapping** è una tecnica per alternare processi in memoria principale senza garantire allocazione fissa, assegnando, infatti, partizioni diverse nel tempo, assegnate ad hoc. Può provocare frammentazione esterna ed occorre ricompattare periodicamente la memoria sprecando non poco tempo.

5.4 Strutture di gestione

È essenziale tenere traccia dello stato d'uso della memoria, e lo si fa o per mezzo di **mappe di bit**, o per mezzo di **liste collegate**, in cui ogni elemento è un segmento e l'allocazione avviene secondo strategie di:

- **First fit**: il primo segmento libero ampio abbastanza;
- **Next fit**: come *first fit* ma cercando solo in avanti;
- **Best fit**: il segmento libero più adatto;
- **Worst fit**: il segmento libero più ampio;
- **Quick fit**: liste diverse per ampiezze tipiche.

5.5 Memoria virtuale

La memoria virtuale nasce per ovviare alle crescenti esigenze di spazio dei singoli processi permettendo di allocare in RAM processi maggiori della stessa, caricandone solo la parte strettamente necessaria e mettendo il resto su disco (come **overlay** (divisione di un processo in parti) ma senza intervento del programmatore). La memoria virtuale è gestita tramite **paginazione** o tramite **segmentazione**. Gli indirizzi virtuali generati dal processo vengono trasformati in fisici dalla **MMU**.

5.6 Paginazione

La memoria virtuale è suddivisa in unità a dimensione fissa dette **pagine**; la RAM è suddivisa in **cornici** (**page frames**), ampie quanto le pagine. I trasferimenti da e verso disco avvengono sempre in pagine intere. Per sapere se una pagina è in RAM le si assegna un **bit di presenza**. Se viene riferita una pagina assente si genera un **page fault**.

5.6.1 Strutture

Ogni processo ha una sua **tabella delle pagine**, contenente per ogni pagina virtuale la corrispondente locazione fisica. La RAM viene scansionata dal **TLB** (**Translation Lookaside Buffer**), realizzata in software o in hardware nell'MMU, per verificare la presenza o meno di una pagina. Data la grandezza delle tabelle delle pagine nelle architetture moderne viene adottata la soluzione della **tabella invertita**, consiste nell'indirizzare in ogni riga un page frame invece edi una pagina (implicando una traduzione più complessa), realizzata come una **tabella hash** (**hash table**).

5.6.2 Rimpiazzo

Quando si produce un *page fault* il S/O deve rimpiazzare una pagina (salvando su disco la pagina rimossa se è stata modificata (*nda* come quando si lavora con la cache)). Il **rimpiazzo ottimale** (**optimal replacement**) rimpiazza la pagina che non sarà usata per maggior tempo, ma non è realizzabile in quanto il S/O non può prevedere a quali pagine il progesso accederà in futuro. Le sue approssimazioni sono:

- **NRU** (**Not Recently Used**): ad ogni page frame vengono assegnati un **bit M** (**modified**) ed un **bit R** (**referenced**) come contatori e viene rimpiazzata una pagina casuale nella classe non vuota ad indice più basso tra classe 0 (non riferita, non modificata), 1 (non riferita, modificata), 2 (riferita, non modificata) e 3 (riferita, modificata);
- **FIFO**: rimpiazza la pagina entrata meno recentemente in RAM;
- **Second chance**: corregge FIFO rimpiazzando solo le pagine con bit R uguale a 0 (può degenerare in FIFO);
- **Orologio**: come SC ma i page frame sono mantenuti in una lista circolare;
- **LRU** (**Least Recently Used**): approssima l'algoritmo ottimale ma necessita di un hardware dedicato e della lista aggiornata ogni volta che si fa un riferimento;
- **NFU** (**Not Frequently Used**): Realizzabile a software, aggiorna periodicamente un contatore per ogni page frame ma non dimentica nulla;
- **Aging**: come NFU ma valuta solo periodicamente;
- **WS approssimato** (vedi sotto): simile all'aging;
- **WS approssimato con orologio** (vedi sotto): come WS approssimato ma con i page frame dipsosti in una lista circolare.

WS: Il **Working Set** (**WS**) è l'insieme delle pagine che un processo ha in uso ad un dato istante. Se esso viene caricato prima dell'esecuzione si ha **prepaging** e si evita il page fault, mentre se la memoria non è sufficiente a contenerlo si ha il fenomeno di **thrashing** (rimpiazzi indesiderati).

N.B.: Aumentare il numero delle page frame non implica necessariamente una diminuzione dei page fault (anomalia di Belady) .

5.7 Segmentazione

La memoria può anche essere divisa in **segmenti** o in **segmenti paginati**. Per accedervi, un programma carica il **selettore** del segmento, che punta ad un **descrittore** contenente una base di indirizzo alla quale bisogna sommare un offset per trovare l'indirizzo fisico (o logico nel caso di segmentazione paginata) cercato.

6 File System

Il File System è il serivizio di S/O progettato per realizzare, conservare, mantenere dati di diverso tipo e dimensione e fornirne l'accesso alle diverse applicazioni. Il **file** è un insieme di dati trattati unitariamente e strutturati in tre livelli:

- Livello **utente**: l'applicativo associa autonomamente significato al contenuto del file;
- Livello di **struttura logica**: il S/O organizza i dati in strutture logiche per facilitarne il trattamento;
- Livello di **struttura fisica**: il S/O mappa le strutture logiche su quelle fisiche della memoria secondaria disponibile.

6.1 Struttura logica di file

Le possibili strutture logiche di un file sono a **sequenza** di byte (puntatore all'inizio del file), a **record** di lunghezza e struttura interna fissa (il S/O deve conoscerne la struttura, obsoleto) o variabile (una tabella contiene i puntatori alle chiavi / ***keys*** che descrivono ogni record).

6.1.1 Modalità di accesso

- **Accesso sequenziale:** un puntatore indirizza il record corrente ed avanza ad ogni lettura o scrittura: si può scrivere solo sulla coda e leggere sequenzialmente (se il file da leggere è stato oltrepassato è necessario tornare al primo indirizzo e ricominciare la ricerca);
- **Accesso diretto:** i record sono posti in posizione arbitraria, determinata rispetto dalla base;
- **Accesso indicizzato:** per ogni file una tabella contenente gli **offset** dei rispettivi record; consente anche accesso sequenziale.

6.1.2 Classificazione di file

- **File regolari:** contenuto ASCII o binario su cui l'utente può operare normalmente;
- **File catalogo (directory):** descrivono l'organizzazione di gruppi di file (file e directory risiedono in aree logiche distinte);
- **File speciali:** rappresentano logicamente dispositivi orientati a carattere o a blocco (*solo UNIX / GNU / LINUX*).

6.2 Struttura di directory

A livello utente, una struttura di directory deve essere efficiente (realizzare e trovare un file deve essere semplice e veloce) e fornire libertà di denominazione e raggruppamento (logico). Rispetto all'organizzazione, le directory possono essere:

- **A livello singolo:** tutti i file sono elencati su una lista lineare, ciascuno con il proprio nome (gestione onerosa all'aumentare dei file);
- **A due livelli:** una **root directory** contiene una **User File Directory** (**UFD**) per ciascun utente di sistema, il quale può vedere solo la propria UFD; i file sono localizzati tramite **percorso** (**path**) e i programmi di sistema possono essere copiati su tutte le UFD o (meglio) posti in una directory di sistema condivisa;
- **Ad albero**: Numero arbitrario di livelli sottostanti alla root directory;
- **A grafo (aciclico o generalizzato):** un file può appartenere contemporaneamente a più directory; UNIX e GNU/Linux utilizzano **link** tra il nome reale del file e la sua presenza virtuale, **hard link** quando un puntatore diretto ad un file viene inserito in un'altra directory remota dello stesso file system (due vie d'accesso dirette), o **symbolic (soft) link**, quando, invece, viene creato un file speciale contenente il cammino del file originario di qualunque FS (una sola via d'accesso);

6.3 Realizzazione del file system

I file system sono memorizzati su dischi il cui settore 0 contiene le informazioni di inizializzazione del sistema (**Master Boot Record**), eseguita dal BIOS.

6.3.1 Realizzazione di file

A livello fisico un file è un insieme di blocchi di disco, allocati in tre modi possibili:

- **Allocazione contigua:** la memorizzazione avviene su blocchi **consecutivi**, ogni file è descritto dall'indirizzo del suo primo blocco ed il numero di blocchi utilizzati; ogni modifica di file comporta il rischio di frammentazione esterna;
- **Allocazione a lista concatenata:** il file è identificato dal puntatore al suo primo blocco, che è anche il puntatore ad una lista concatenata di blocchi, contenenti ognuna parte del file (un solo blocco guasto corrompe l'intero file);
- **Allocazione a lista indicizzata:** i blocchi contengono solo dati e i puntatori si trovano in strutture apposite, di forma tabulare (**FAT**, **File Allocation Table**) o indicizzata (**i-node**, nodo indice), consentendo accesso sequenziale e diretto senza causare frammentazione esterna.

  FAT: tabella ordinata di puntatori a blocchi (**cluster**); un file è una catena di indici e la FAT relativa ai file in uso deve risiedere interamente in RAM; la sua dimensione non dipende dalla grandezza dei file al suo interno bensì dalla dimensione della partizione di disco e dall'ampiezza dei suoi indirizzi;

  i-node: una struttura indice ad ampiezza limitata contenente gli attributi di ogni file ed i puntatori ai suoi blocchi, quindi non è influenzata dalla contiguità (o meno) dei blocchi; in RAM una tabella per i soli file in uso. Per file medi e grandi (troppi puntatori per un singolo i-node) gli i-node possono puntare rispettivamente uno o più blocchi i-node intermedi;

6.3.2 Gestione dei blocchi liberi

Si può optare per una lista concatenata dei blocchi liberi (come in FAT) o per un vettore di bit (**bitmap**), dove ogni bit indica lo stato del corrispondente blocco (0 libero, 1 occupato).

6.4 Integrità del File System

I blocchi danneggiati vengono gestiti **via software**, occupandoli con un falso file, o **via hardware**, creando e mantenendo in un settore del disco un elenco di blocchi danneggiati e dei loro sostituti. Il FS viene salvato su nastro (tempi lunghi) o su disco, con partizione di **back-up** o mediante **RAID**. Ogni blocco può appartenere ad una delle due liste dei blocchi in uso e liberi, se appartiene a due si ha una duplicazione, se a nessuna si ha una perdita. Una parte della memoria principale viene usata come cache per alcuni blocchi e le modifiche vengono ricopiate immediatamente su disco (**write-through**, MS-DOS) o con un refresh periodico (**sync**, UNIX / GNU / Linux).