Contents

Sistemi operativi

kocierik

September 2021

1 Introduzione alla concorrenza

Un **Processo** è un'attività controllata da un programma che si svolge su un processore. Un **programma** è un entità **statica**, un processo è **dinamico**.
Assioma di finite progress: Ogni processo viene eseguito ad una velocità finita, non nulla ma sconosciuta.

1.0.1 Stati dei processi

- **Running** il processo è in esecuzione
- **Waiting:** il processo è in attesa di qualche evento esterno, non può essere eseguito
- **Ready** il processo può essere eseguito, ma attualemtne il processore è impegnato in altre attività.

Il tema principale nella progettazione dei sistemi operativi riguarda la gestione di processi **multipli:**

- **Multiprogramming**
  - Più processi su un solo processore
  - Parallelismo apparente
- **Multiprocessing**
  - Più processi su una macchina con processori multipli
  - Parallelismo reale
- **Distributed processing**
  - Più processi su un insieme di computer distribuiti e indipendenti
  - Parallelismo reale

Due programmi si dicono in **esecuzione concorrente** se vengono eseguiti in parallelo. La concorrenza è l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali **comunicazione** e **sincronizzazione**.
Non vi è sostanziale differenza tra i problemi relativi a **multiprogramming** e **multiprocessing**. I problemi derivano dal fatto che:

- Non è possibile predire gli instanti temporali in cui vengono eseguite le istruzioni
- I due processi accedono ad una o più risorse condivise

Si dice che un sistema di processi multipli presenta una **race condition** qualora il risultato finiale dell'esecuzione dipenda dalla temporizzazione con cui vengono eseguiti i processi. Per scrivere un programma concorrente è necessario **eliminare** la race condition.

2 Interazione tra processi

E' possibile classificare le modalità di interazione tra processi in base a quanto sono **consapevoli** uno dell'altro. Processi separati devono sincronizzarsi nella loro utilizzazione. Il sistema operativo deve **arbitrare** questa competizione, fornendo meccanismi di **sincronizzazione**.
Una **proprietà** di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso. Sono presenti due tipi di proprietà:

- **Safety**
  - Mostrano che il programma va nella direzione voluta, cioè non esegue azioni scorrette
- **Liveness**
  - Il programma avanza, non si ferma, è **vitale**

Dalla teoria dei sistemi distribuiti deriva il problema del **consensus**. Si consideri un sistema con `N` processi:

- Ogni processo propone un valore
- Alla fine tutti si devono accordare su uno dei valori proposti
- Proprietà di safety
  - Se un processo decide, deve decidere uno dei valori proposti
  - Se due processi decidono, devono decidere lo stesso valore
- Proprietà di liveness
  - Prima o poi ogni processo corretto prenderà una decisione

Nei programmi sequenziali, le proprietò di **safety** esprimono la correttezza dello stato finale. La principale proprietà di **liveness** è la terminazione.

- Proprietò di safety:
  - I processi non devono **interferire** fra di loro nell'accesso alle risorse condivise
- Proprietò di liveness:
  - I meccanismi di sincronizzazione utilizzati non devono prevenire l'avanzamento del programma. Non è possibile che un processo debba **attendere indefinitamente** prima di poter accedere ad una risorsa condivisa

L'accesso ad una risorsa si dice **mutualmente esclusivo** se ad ogni istante, al massimo un processo può accedere a quella risorsa. La mutua esclusione permette di risolvere il problema della non interferenza, ma può causare il blocco permanente dei processi. La assenza di deadlock è una proprietà di safety (Esempio: incrocio stradale). Nei sistemi reali, se ne può uscire solo con metodi distruttivi, ovvero uccidendo i processi, riavviando la macchina.

2.0.1 Starvation

Il **deadlock** è un problema che coinvolge tutti i processi che utilizzano un certo insieme di risorse. Esiste anche la possibilità che un processo non possa accedere ad una risorsa perché sempre occupata. L'assenza di **starvation** è una proprietà di liveness. Se durante l'esecuzione di un programma arrivano processi di maggiore importanza il processo in coda con minor importanza dovrà aspettare.
A differenza del deadlock la **starvation** non è una condizione definitiva, è possibile uscirne, basta adottare un'opportuna politica di assegnamento. E' comunque una situazione da evitare.

2.0.2 Azioni atomiche

Le azioni atomiche vengono compiute in modo indivisibile e soddisfano la condizione: o tutto o niente. Nel caso di **parallelismo reale** si garantisce che l'azione non interferisca con altri processi durante la sua esecuzione. Nel caso di **parallelismo apparente** l'avvicendamento (**context switch**) fra processi avviene prima o dopo l'azione, che quindi può interferire.
Le singole istruzioni del linguaggio macchina sono **atomiche**. Nel caso del **parallelismo apparente** gli interrupt garantiscono un hc ebenga eseguito prima o dopo un'instruzione, mai durante. Nel caso di **parallelismo reale** si garantisce che uno dei due processi venga servito prima dell'altro usando una **politica di arbitraggio del bus**. In generale, sequenze di istruzioni in linguaggio macchina **non sono azioni atomiche**. Le singole istruzioni invece si.

3 Sezioni critiche

3.1 Problemi delle critical section

La parte di un programma che utilizza una o più risorse condivise viene detta **sezione critica (critical section, o CS)**

```
process P1
  a1 = read();
  totale = totale + a1;
}
```

```
process P2 {
  a2 = read();
  totale = totale + a2;
}
```

I requisiti per le **CS** (critical section) sono:

- **Mutua esclusione**
  - Solo un processo alla volta deve essere all'interno della CS, fra tutti quelli che hanno una CS per la stessa risorsa condivisa
- **Assenza di deadlock**
  - Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile
- **Assenza di delay non necessari**
  - Un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo
- **Eventual entry (assenza di starvation)**
  - Ogni processo che lo richiede, prima o poi entra nella CS

Dei possibili approcci sono:

- **Approcci software**, dove la responsabilità cade sui processi che vogliono accedere ad un oggetto distribuito (costoso in termini di esecuzione)
- **Approcci hardware**, utilizza istruzioni speciali del linguaggio macchina, efficient, ma non sono adatti come soluzione general-porpose

Nel 1965 **Dijkstra** riporta un algoritmi basato sulla mutua esclusione, progettato dal matematico **Dekker**.s

```
shared int turn = P;
shared boolean needp = false; shared boolean needq = false;
cobegin P // Q coend

process P {                          process Q {
  while (true) {                       while (true) {
    /* entry protocol */                 /* entry protocol */
    needp = true;                        needq = true;
    while (needq)                        while (needp)
      if (turn == Q) {                     if (turn == P) {
        needp = false;                       needq = false;
        while (turn == Q)                    while (turn == P)
          ; /* do nothing */                   ; /* do nothing */
        needp = true;                        needq = true;
      }                                    }
    critical section                     critical section
    needp = false; turn = Q;             needq = false; turn = P;
    non-critical section                 non-critical section
  }                                    }
}                                    }
```

Nel 1981 **Peterson** ne ideò uno più semplice e generalizzabile al caso di processi multipli

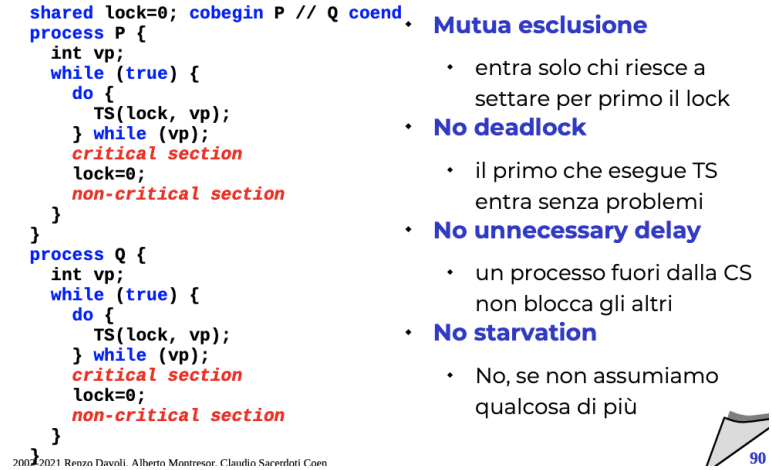
```
shared boolean needp = false;
shared boolean needq = false;
shared int turn;

cobegin P // Q coend
process P {                          process Q {
  while (true) {                       while (true) {
    /* entry protocol */                 /* entry protocol */
    needp = true;                        needq = true;
    turn = Q;                            turn = P;
    while (needq && turn != P)           while (needp && turn != Q)
      ; /* do nothing */                    ; /* do nothing */
    critical section                     critical section
    needp = false;                       needq = false;
    non-critical section                 non-critical section
  }                                    }
}                                    }
```

Riassumendo le soluzioni software permettono di risolvere il problema delle critical section, ma sono tutte basate su **busy waiting**, che spreca il tempo del processore, ed è una tecnica che non dovrebbe essere utilizzata!
Per risolvere questo problema si puo' pensare di fornire alcune istruzioni hardware speciali per semplificare la realizzazione di sezioni critiche. Il S.O deve lasciare ai processi la resposabilita' di riattiare gli interrupt, ma questo e' altamente pericoloso. Le istruzioni speciali sono:

- Istruzioni che realizzano due azioni in modo atomico
  - Lettura e scrittura
  - Test e scrittura
- Le sezioni critiche realizzare con istruzioni speciali vengono chiamate **spinlock**

$$TS(x, y) :=< y = x; x = 1 >$$

Ritorna in **y** il valore precedente di **x** e assegna 1 ad **x**



Altri istruzioni possibili possono essere

- fetchset
- compareswap
- etc.

3.1.1 vantaggi e svantaggi delle istruzioni speciali hardware

vantaggi:

- Sono applicabili a qualsiasi numero di processi, sia su sistemi monoprocessore che in sistemi multiprocessori
- Semplici e facili da verificare
- Puo' essere utilizzato per supportare sezioni critiche multiple, ogni sezione critica puo' essere definita dalla propria variabile

svantaggi:

- Si utilizzza ancora busy-waiting
- I problemi di starvation non sono eliminati
- Sono comunque complesse da programmare

4 Semafori

Il nome indica chiaramente che si tratta di un paradigma per la **sincronizzazione**. Termine utilizzato da Dijkstra nel 1965, con l'obiettivo di descrivere un S.O come una collezioni di processi sequenziali ed anche per facilitare la **cooperazione**.

4.1 Definizione

Due o piu' processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finche' non riceve un segnale da un altro processo.
E' un tipo di dato astratto per il quale sono definite due operazioni:

- **V**: viene invocata per inviare un segnale, quale il verificars idi un evento o il rilascio di una risorsa
- **P**: viene invocata per attendere il segnale (ovvero per attendere un evento)

Un semaforo puo' essere visto come una **variabile intera**. Questa variabile viene inizializzata ad un valore **non negativo**

- **P** attende che il semaforo sia positivo e lo decrementa
- **V** incrementa il valore del semaforo

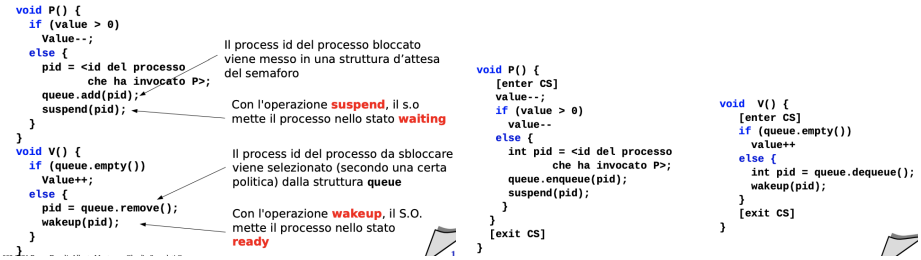
Queste due azioni sono atomiche. Il valore del semaforo deve sempre essere non negativo ($>= 0$)

4.1.1 Due casi d'uso

- Il numero di eventi consegnati deve essere non superiore al numero di volte che l'evento si e' verificato
- Il numero di richieste soddisfatte non deve essere superiore al numero iniziale di risorse + il numero di risorse restituite
- Il numero di richieste soddisfatte non deve essere superiore al numero iniziale di risorse + il numero di risorse restituite

```
Semaphore s = new Semaphore(1);
process P {
  while (true) {
    s.P();
    critical section
    s.V();
    non-critical section
  }
}
```

Mutua esclusione, assenza di deadlock, assenza di starvation, assenza di ritardi non necessari. Per ogni semaforo occorre mantere una struttura dati contenente l'insieme dei processi sospesi, quando un processo deve essere sveglioato, e' necessario selezionare uno dei processi sospesi Viene utilizzata una politica **FIFO** (First-In-First-Out)



E' possibile disabilitare/riabilitare gli interrupt all'inizio-fine di **P** e V. E' possibile farlo perchè sono implementate direttamente dal sistema operativo.
Esistono anche semafori binari in cui il valore puo' assumere solo i valori 0 e 1. Servono a garantire mutua esclusione, semplificando il lavoro del programmatore.

4.1.2 Semafori binari

Variante dei semafori in cui il valore può assumere solo i valori 0 e 1.

```
class BinarySemaphore {

  private Semaphore s0, s1;
  int value;

  BinarySemaphore(int v) {   // +fail if v not in {0,1}
    S0 = new semaphore(v)
    S1 = new semaphore(1-v)
  }
  void P9void) {
    S0.P();
    S1.V();
  }
  void V(void) {
    S1.P();
    S0.V();
  }
}
```

Esistono un certo numero di problemi della programmazione concorrente:

- **Produttore/consumatore**, **producer** genera i valori e vuole trasferirli a un processo Consumer. Producer non deve scrivere nuovamente l'area di memoria condivisa prima che Consumer abbia utilizzato il valore precedente. Consumer non deve leggere due volte lo stesso valore.

- **Buffer limitato**, lo scambio tra produttore e consumatore avviene tramite un buffer. **Producer** non deve sovrascrivere elementi del buffer prima che **consumer** abbia utilizzato i valori. **Consumer** non deve leggere due volte lo stesso valore.

- **Filosofi a cena**, mostra come gestire situazioni in cui i processi entrano in competizione per accedere ad insiemi di risorse a intersezione non nulla.

- **Lettori e scrittori**, un database è condiviso tra un certo numero di processi; esistono due tipi di processi:

– I **lettori** accedono al database per leggerne il contenuto
– Gli **scrittori** accedono al database per aggiornare il contenuto

se uno scrittore accede a un database per aggiornarlo, esso opera in mutua esclusione; nessun altro lettore o scrittore può accedere al database.
Se nessuno scrittore sta accedendo al database, un numero arbitrario di lettori può accedere al database in lettura

rappresentano le interazioni tipiche dei processi concorrenti.

4.2 Lettori e scrittori

I **lettori** accedono al database per leggerne il contenuto. Gli **scrittori** accedono al database per aggiornarne il contenuto. Se uno scrittore accede a un database per aggiornarlo, esso opera in mutua esclusione; nessun altro lettore o scrittore può accedere al database. Se nessuno scrittore sta accedendo al database, un numer oarbitrario di lettori può accedere al database in lettura.

4.3 Come derivare una soluzione basata su semafori

1. **Definire il problema con precisione**
   - Identificare i processi, specificare i problemi di sincronizzazione, introdurre le variabili necessarie e definire un'invariante
2. **abbozzare una soluzione**
   - Produrre un primo schema di soluzione, e identificare le regioni che richiedono accesso atomico o mutualmente esclusivo
3. **Garantire l'invariante**
   - Verificare che l0invariante sia sempre verificato
4. **Implementare le azioni atomiche**
   - Esprimere le azioni atomiche e gli statement **await** utilizzando le primitive di sincronizzazione disponibili

4.4 Difetti dei semafori

I semafori sono costrutti di basso livello. E' responsabilità del programmatore non commettere alcuni possibili errori banali

- Omettere **P** e **V**
- Scambiare l'ordine delle operazioni **P** e **V**
- Fare operazioni **P** e **V** su semafori sbagliati

E' responsabilità del programmatore accedere ai dati condivisi in modo corretto

- Più processi possono accedere ai dati condivisi

Vi sono forti problemi di leggibilità

5 Monitor

I monito sono un paradigma di programmazione concorrente che fornisce un approccio più strutturato alla programmazione concorrente. Sono stati introdotti nel 1974 da Hoare.

5.1 Introduzione

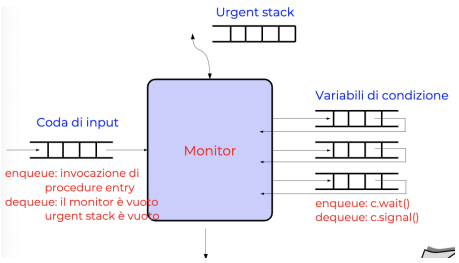
Un monitor è un **modulo software** che consiste di:

- Dati locali
- Una sequenza di inizializzazione
- una o più procedure

Le caratteristiche principali sono:

- I dati locali sono accessibili solo alle procedure del modulo stesso
- Un processo entra in un monitor invocando una delle sue procedure
- Solo un processo alla volta può essere all'interno del monitor, gli altri processi che invocano il monitor sono sospesi, in attesa che il monitor diventi disponibile

Sintassi:



Assomiglia ad un **oggetto** nella programmazione.

- Il codice di inizializzazione corrisponde al costruttore
- Le procedure entry sono richiamabili dall'esterno e corrispondono ai metodi pubblici di un oggetto
- Le procedure normali corrispondono ai metodi privati
- Le variabili locali corrispondono alle variabili private

Solo un processo alla volta può essere all'interno del monitor

- Il monitor fornisce un semplice meccanismo di mutua esclusione
- Strutture dati condivise possono essere messe all'interno del monitor

E' necessario un meccanismo di **sincronizzazione**. Abbiamo bisogno di:

- Poter sospendere i processi in attesi di qualche condizione
- Far uscire i processi dalla mutua esclusione mentre sono in attesa
- Permettergli di rientrare quando la condizione è verificata

5.1.1 Meccanismi di sincronizzazione

Dichiarazione di variabili di condizione (CV)

- `condition c;`

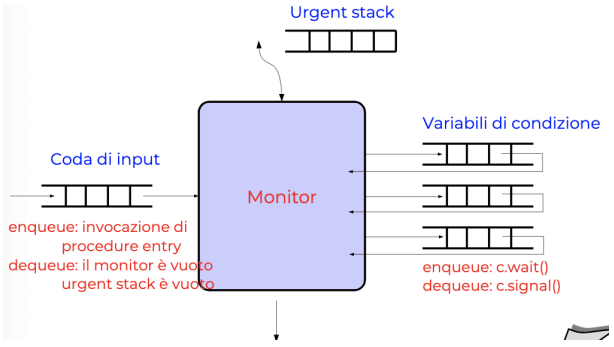
Le operazioni definite sulle CV sono:

- `c.wait()` attende il verificarsi della condizione
- `c.signal()` segnala che la condizione è vera

5.2 Politica signal urgent

- `c.wait()` viene rilasciata la mutua esclusione e il processo che chiama **c.wait()** viene sospeso in una coda di attesa della condizione **c**.
- **c.signal()** causa la riattivazione immediata di un processo (FIFO).

Il chiamante viene posto in attesa e verrà riattivato quando il processo risvegliato avrò rilascitato la mutua esclusione **(urgent stack)**. Se nessun processo sta attendendo **c** la chiamata non avrà nessun effetto.



signal non ha alcun effetto se nessun processo sta attendendo la condizione **V** (semaforo).
wait è sempre bloccante **P** (se il semaforo ha valore positivo) no. Il processo risvegliato della **signal** viene eseguito per primo.
Signal urgent è la politica classica di signaling. Ne esistono altre:

- SW-**signal wait** no urgent stack, signaling process viene messo nella entry queue
- SR-**signal and return** dopo la **signal** si esce subito dal monitor
- SC-**signal and continue** la **signal** segnala solamente che un processo può continuare, il chiamante prosegue l'esecuzione. Quando lascia il monitor viene riattivato il processo segnalato

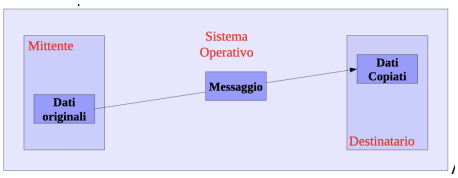
I semafori hanno una memoria in sè a differenza dei segnali, nei quali non appena conclude la sua operazione cancella il dato. Nei semafori non viene utilizzata la **busy-wait**.

6 Message passing

Semafori, monitor sono paradigmi di sincronizzazione tra processi. In questi paradigmi, la comunicazione avviene tramite memoria condivisa. Il meccanismo detto **message massing** è un paradigma di comunicazione tra processi. La sincronizzazione avviene tramite lo scambio di messaggi, e non più semplici segnali.

6.1 Messaggio

Un **messaggio** è un insieme di informazioni formattate da un processo mittente e interpretate da un processo destinatario. Un meccanisco di **scambio di messaggi** copia le informazioni di un messaggio da uno spazio di indirizzamento di un processo allo spazio di indirizzamento di un altro processo.



Send:

- Utilizzata dal processo mittente per spedire un messaggio ad un processo destinatario
- Il processo destinatario deve essere specificato

Receive:

- Utilizzata dal processo destinatario per ricevere un messaggio da un processo mittente
- Il processo mittente può essere specificato, o può essere qualsiasi

Il passaggio dallo spazio di indirizzamento dal mittente a quello del destinatario è mediato dal sistema operativo. Il processo destinatario deve eseguire un'operazione **receive** per ricevere qualcosa.

MP sincrono:

- Send sincrono
- Receive bloccante

MP asincrono:

- Send asincrono
- Receive bloccante

MP completamente asincrono:

- Send asincrono
- Receive non bloccante

$$ssend(m, q)$$

Il mittente **p** spedisce il messaggio **m** al processo **q**, restando bloccato fino a quanto **q** non esegue l'operazione **sreceive(m,q)**.

$$m = sreceive(p)$$

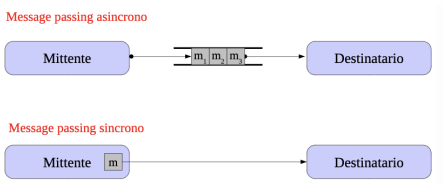
Il destinatario **q** riceve il messaggio **m** dal processo **p** se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio. E' possibile lasciare il mittente non specificato (utilizzando la costante **ANY** per il parametro **p**)

$$asend(m, q)$$

Il mittente **p** spedisce il messaggio **m** al processo **q**, senza bloccarsi in attesa che il destinatario esegua l'operazione **nb-receive(p)**.

$$m = nb - receive(p)$$

Il destinatario **q** riceve il messaggio **m** dal processo **p**, se il mittente non ha ancora spedito alcun messaggio, la **nb-receive** termina ritornando un messaggio nullo E' possibile lasciare il mittente non specificato (utilizzando la costante **ANY** per il parametro **p**)



7 Riassunto

- **Sezioni critiche:**
  - Meccanismi fondamentali per realizzare mutua esclusione in sistemi mono e multiprocessore all'interno del sistema operativo stesso
  - Ovviamente livello troppo basso
- **Semafori:**
  - Fondamentale primitiva di sincronizzazione, effettivamente offerte dai S.O
  - Livello troppo basso; facile commettere errori
- **Monitor:**
  - Meccanismi integrati nei linguaggi di programmazione
  - Pochi linguaggi di larga diffusione sono dotati di monitor
  - Unica eccezzione Java, con qualche distinguo
- **Message passing:**
  - Da un cert opunto d ivista, il meccanismo più diffuso
  - Può essere poco efficiente (copiad ati tra spazi di indirizzamento)

7.1 Potere espressivo

Si dice che il paradigma di programamzione **A** è espressivo almeno quanto il paradigma di programmazione **B** quando è possibile esprimere ogni programma scritto con **B** mediante **A** ($A \geq B$).

Si possono tracciare le seguenti classi di paradigmi:

- **Metodi a memoria condivisa:**
  - Semafori, semafori binari, monitor hanno tutti lo stesso potere espressivo
  - Dekker e peterson, TestSet necessitano di busy waiting

  Metodi a memoria privata
  - Message passing asincrono ha maggiore potere espressivo
  - Message passing sincrono (abbiamo dovuto aggiungere un processo, non solo una libreria)

8 Introduzione ai sistemi operativi

8.1 Cos'è un sistema operativo

Un sistema operativo è livello di astrazione e:

- Realizza il concetto di processo
- Il "linguaggio" fornito dal **S.O** è definito dalle **system call**
- E' implementato tramite un programma che **controlla** l'esecuzione di programmi applicativi e agisce come **interfaccia** tra le applicazioni e l'hardware

Un S.O deve utilizzare in modo efficiente le risorse della macchina e semplificare l'utilizzo

8.2 S.O come macchina estesa

Servizi estesi offerti da un S.O:

- Esecuzione di programmi
- Accesso semplificato/unificato ai dispositivi di I/O
- Accesso a file system
- Accesso a networking
- Accesso al sistema
- Rilevazione e risposta agli errori
- Accounting

Un **job** è un programma o un'insieme di programmi la cui essecuzione veniva richiesta da uno degli utilizzatori del computer. Il ciclo di esecuzione di un job è rappresentato da:

- Il **programmatore**
  - scrive su carta un programm in linguaggio ad alto livello
  - Perfora una serie di schede con il programma e il suo input
- L'**operatore**
  - Inserisce schede di controllo scritte **JCL** (Job Control Language)
  - Inserisce schede di controllo
  - Attende il risultato

Nella **multiprogrammazione** vengono sviluppati i primi algoritmi di **scheduling** dove vengono gestiti in parallelo più job.
Time sharing è l'estensione logica della multiprogrammazione dove l'esecuzione della CPU viene suddivisa in un certo numero di quanti temporali. Questi passaggi di **context switch** permettono che più utenti possano interagire con i programmi in esecuzione.
Un **sistema parallelo** è un elaboratore che possiede più unità di elaborazione. I tipi di elaborazioni possono essere di:

- **SIMD** - Single Instruction, Multiple Data
- **MIMD** - Multiple Instruction, Multiple Data

A seconda del numero dei procecssori si suddividono in **sistemi a basso parallelismo** dove abbiamo pochi processori in genere molto potenti oppure **sistemi massicciamente paralleli**, con un gran numero di processori.
Altre tipologie di sistemi paralleli possono essere **tightly** coupled dove i core condividono i bus e la memoria. E sistemi **loosely** compled, dove i processori lavorano separatamente con memorie private.
Un'altra caratteristica dei sistemi paralleli è la **Symmetric multiprocessing** dove ogni processore esegue una copia identica del sistema operativi e processi diversi possono essere eseguiti contemporaneamente.
I processi **Asymmetric multiprocessing** invece procedono nel modo inversi e quindi assegnando ad ogni processore un compito specifico.
Un **sistema distrubuito** è un insieme di elaboratori indipendenti, collegati da una rete, apparendo come un unico sistema.
Un **sistema real-time** sono sistemi per i quali la correttezza del risultato dipenda dal suo valore e anche dall'instante nel quale il risultato viene prodotti (viene aggiunto ad esempio il vincolo temporale).
I sistemi real-time si dividono in:

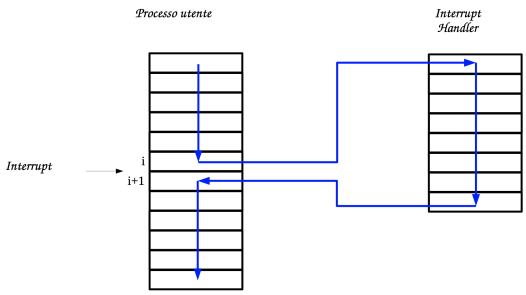
- **hard real-time**, se il mancato rispetto dei vincoli produce effetti catastrofici (controlli centrali nucleari, controlli velivoli)
- **soft real-time**, se si hanno solo disservizi (programmi interattivi)

8.3 architettura hardware

Un **interrupt** è un meccanismo che permetto l'interruzione del normale ciclo di esecuzione della CPU. Esso permette di intervenire durante l'esecuzione di un programma, allo scopo di gestire in modo efficace le risorse del sistema, e possono essere sia hardware che software. Nel caso non avessimo a disposizione questo tipo di meccanismo, potremmo replicarlo utilizzando la system call relativa **poll** il quale produrrebbe un risultato con pessime performance.
Gli interrupt Software (**trap**) sono causati dal programma e indicano gli errori relativi (segmentation fault, ecc..)
In seguito ad un interrupt viene spedito un segnale di **request** al processore, che sospenderà le operazioni del processo corrente e salterà ad un particolare

indirizzo di memoria contentente la routine di gestione dell'interrupt (**interrupt handler**). L'interrupt handler gestisce gli interrupt e ritorna li controllo al processo interrotto. Analizzando nel dettaglio gli interrupt:

1. Un segnale di interrupt request viene mandato alla CPU
2. La CPU finisce l'esecuzione dell'istruzione corrente
3. La CPU verifica la presenza di un segnale interrupt
4. Preparazione al trasferimento di controllo del programma all'interrupt handler (salvando i registri)
5. Selezione dell'interrupt handler appropriato
6. Caricamenteo del PC con l'indirizzo iniziale dell'interrupt handler assegnato
7. Salvataggio dello stato del processore
8. Gestione dell'interrrupt
9. Ripristino dello stato del processore
10. Ritorno del controllo al processo in esecuzione (o ad un altro processo)



La comunicazione tra processore e dispositivo I/O può avvenire in due modalità possibili:

- **Programmed I/O**, oramai obsoleta, dove la CPU attraverso i bus carica i parametri delle richieste di input nei registri, il dispositivo attenderà il risultato (**busy-waiting-polling**)
- **Interrupt-Driver I/O**, la CPU carica attraverso i basi i parametri delle richieste negli appositi registri, il S.O sospenderà l'esecuzione del processo che ha eseguito ed esegue un altro processo, quando l'operazione viene completata viene mandato un segnale di interrupt.

Nel caso di operazioni in output il procedimento è molto simile.
Un controller **DMA** (Direct Memory Access) evita l'uso del buffer nel controller. Assieme ai registri, l'unico spazio di memorizzazione che può essere acceduto direttamente dal processore, l'acesso avviene mediante istruzioni **LOAD/STORE**. Nei sistemi moderni l'accesso avviene mediate **MMU** (trasforma indirizzi logici in fisici).
Un dispositivo è **completamente** indirizzabile tramite bus. Un **disco** consente la memorizzazione non volatile dei dati, tramite accesso diretto. Le operazioni gestite dal controller di disco sono:

- **READ**
- **WRITE**
- **SEEK**

Le operazioni di seek che corrisponde allo spostamento del pettine di testa devono essere limitati perché è la più costosa. Essi inolte si leggono a blocchi e si scrivano a banchi.
Possiamo dividere i meccanismi di **caching** in due tipi:

- Hardware, cache, cpu (politiche **non modificabili** dal S.O
- Software, cache, disco (politiche **modificabili** dal S.O

Alcuni problemi relativi a questi meccanismi sono problemi di:

- Algoritmo di replacement, che deve garantire il maggior numero di accessi in cache
- Coerenza, dove gli stessi dati possono apparire in diversi livelli della struttura di memoria

I sistemi multiprogrammati e multiutente richiedono la presenza di meccanismi di protezione; bisogno evitare che processi concorrenti degli utenti generino interferenze con il sistema operativo. La protezione HW è data dalla modalità kernel (ring 0) che garantisce l'accesso a tutte le istruzioni, incluse quelle **privilegiate**, che permettono di gestire totalmente il sistema. Per distinguere fra le modalità si utilizza il **Mode bit** del processore. Le istruzioni di I/O devono essere considerate privilegiate.

9 Architettura dei sistemi operativi

L'architettura di un sistema operativo descrive quali sono le varie componenti del S.O. Essa possiamo identificarla da diversi punti di vista:

- (servizi forniti)
- Interfaccia di sistema

- Componenti del sistema

Le componenti di un sistema operativo sono:

1. Gestione dei processi, un processo utilizza le risorse fornite dal computer per assolvere i propri compiti. Il sistema operativo è resposabile di:
   - Creazione e terminazione di processi
   - Sospensione e riattivazione dei processi
   - Gestione dei deadlock
   - Comunicazione tra processi
   - Sincronizzazione tra processi
2. Gestione della memoria principale, è un "array" di byte indirizzabile singolarmente. Il S.O è responsabile delle seguenti attività
   - Tenere tracciad i quali parti della memoria sono usate
   - Decidere quali processi caricare quando diventa disponibile della memoria
   - Allocare e deallocare spazio di memoria
   - Usare memoria secondaria per ampliare la memoria principale
3. Gestione della memoria secondaria, in genere la memoria secondaria è data dall'hard disk e dischi ottici. Il S.O è resposabile delle seguenti attività:
   - Gestione del partizionamento
   - Gestione dell'accesso efficiente e affidabile (RAID)
   - Ordinamento efficiente delle richieste (disk scheduling)
4. Gestione file system, il concetto di file è indipendente dal media sul quale viene memorizzato. Un file system è composto da un insieme di file. Il S.O è responsabile delle seguenti attività:
   - Creazione e cancellazione di file e directory
   - Manipolazione di file e directory
   - Codifica del file system su una sequenza di blocchi
5. Gestione dei dispositivi I/O, gestisce una interfaccia comune per la gestione dei device driver
6. Supporto multiuser, il termine protezione si riferisce al meccanismo per controllare gli accessi dei processi alle risorse del sistema e degli utenti. Esso deve:
   - Gestire identità del proprietario (uid gid)
   - Gestire chi può fare cosa

    - Fornire un meccanismo di attuazione della protezione

7. Networking, consente di far comunicare e condividire processi e risorse in esecuzione su più elaboratori. Utilizzando protocolli di basso livello:
    - TCP/IP
    - UDP
    - Servizi di comunicazione ad alto livello, file system distribuiti (NFS, SMB)
8. Inter Process Communication (IPC)

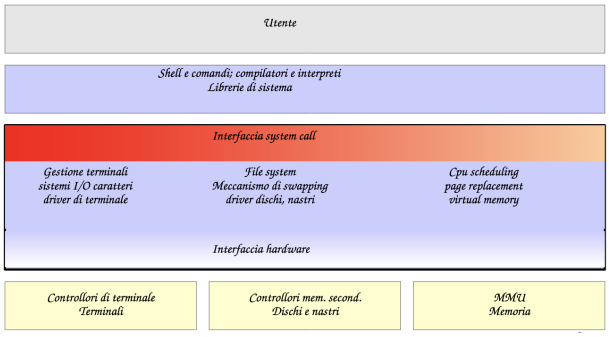
La progettazione di un S.O deve tener conto di diverse caratteristiche:

- Efficienza
- manutenibilità
- Espansibilità
- Modularità
- Efficienza
- manutenibilità
- Espansibilità
- Modularità
- Efficienza
- manutenibilità
- Espansibilità
- Modularità

Spesso queste caratteristiche presentano un trade-off, sistemi molto efficienti sono poco modulari e viceversa.
E' possibile suddividere i S.O in due grandi famiglie, a seconda della lor struttura:

- Sistemi con struttura semplice, che sono sistemi che non hanno una struttura progettata a priori, possono essere descritti come una collezione di procedure. Tipicamente sono S.O semplici e limitati (**MS-DOS**)
- Sistemi con struttura a strati, che tendono a essere meno efficienti e occorre studiare accuratamente al struttura dei layer. I moderni sistemi tendono ad avere meno strati

9.1 Architettura Unix



9.1.1 Politiche e meccanismi

E' presente una separazione tra la politiche e i meccanismi. La politica decide cosa deve essere **fatto** e i meccanismi attuano la **decisione**. La componente che prende le decisioni politiche può essere completamente diversa da quella che implementa i meccanismi. Questo concetto rende possibile cambiare politiche e meccanismi in modo indipendente.
Nei sistemi a **microkernel** si implementano solo meccanismi, delegando la gestioneb della politica a processi fuori dal kernel, esempio **MINIX**, dove il gestore della memoria è un processo esterno al kernel. Quando deve attuare delle operazioni per implementare la politica decisa lo fa tramite chiamate specifiche al kernel (**system task**).

9.2 Organizzazione del kernel

Esistono vari tipi di kernel:

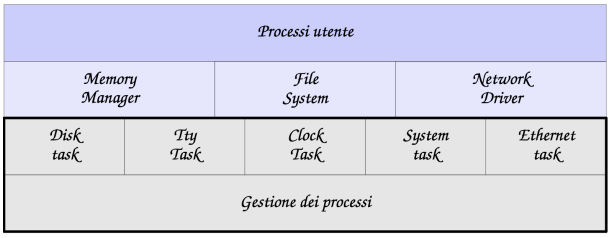
- Kernel monolitici, un cui abbiamo un aggregato unico di procedure di gestione mutualmente coordiante e astrazione dell'HW (efficienza e modularità).
- MicroKernel, semplici astrazione dell'HW gestite e coordinate da un kernel minimale basate su un paradigma client/server, e primitive di message passing. Le sistem Call di un S.O basato su microkernel sono **send** e **receive** (tramite queste due SC è possibile implementare le API standard della gran parte dei S.O. Questa tipologia risulta:
  - **Semplice** e facile da realizzare
  - **Espandibile** e modificabile, è possibile aggiungere servizi
  - **Portabile** ad altre architetture, grazie alla ricompilazione
  - **Robusto**, se un processo di un servizio cade il sistema funziona ancora
  - **Sicurezza**, è possibile assegnare livelli di sicurezza

– **Adattabilità** del modello ai sistemi distribuiti

Come svantaggio abbiamo una maggiore **inefficienza**, dovuta all'**overhead**

- Kernel ibridi, simili a MicroKernel, ma hanno componenti eseguite in kernel space (spazio utente), per ragioni di efficienza e adottano **message passing** (memoria condivisa)
- Esokernel, che non forniscono livelli di astrazione dell'HW, ma forniscono librerie che mettono a contatto diretto le applicazioni con l'HW. Creato per scopo didattico

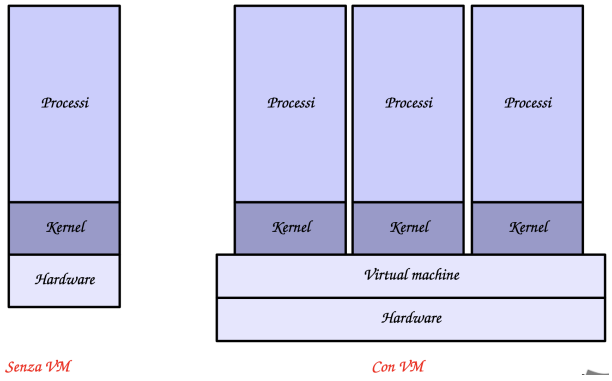
Il **kernel** è dato dal gestore dei processi e dai task. I task sono i **thread** del kernel



Nel 1990 i kernel monolitici sono diventati obsoleti. I microkernel invece sono maggiormente utilizzati dove non sono ammessi **failure**.

9.3 Macchine virtuali

Nelle macchine virtuali abbiamo un approccio diverso al **multitasking**. Invece di creare l'illusione di molteplici processi che posseggono la propria CPU e la propria memoria, si crea l'astrazione di una macchina virtuale. Le macchine virtuali emulano il funzionamento dell'hardware.



I vantaggi delle macchine virtuali sono i seguenti:

- Consentono di far coesistere S.O differenti

- Possono fare funzionare S.O monotask in un sistema multitask e "sicuro"
- Possono essere emulate architetture hardware differenti (intel, arm, mips)

Gli svantaggi sono:

- Soluzione inefficiente, con istruzioni HW di virtualizzazione
- Difficile condividere risorse

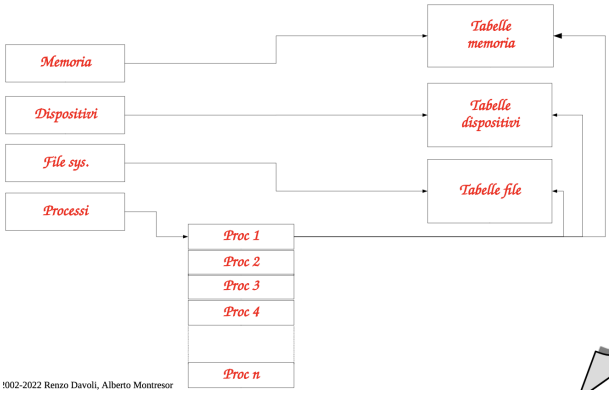
Alcuni esempi di virtual machine sono:

- **Qemu** PVM or SVM, libre sw, utilizza una cross emulation platform (ia32, ia64, sparc, arm...) e una traduzione dinamica a run-time traducendo e ricompilando per la macchina da utilizzare
- **KVM**: SVM software libero, che utilizza I/O virtuali di QEMU, e istruzioni per la virtualizzazione intel VT-x (processori che supportano virtualizzazione), AMD SVM
- **XEN**: SVM, Native, utilizza un device di base chiamato **domain0** il quale virtualizza le possibili macchine
- **VMWare**: SVM, proprietario e dual
- **VirtualBox**: SVM, doppia licenza e dual

10 Scheduler, processi e thread

Un Sistema Operativo è un gestore di risorse. Per svolgere i suoi compiti, un sistema operativo ha bisogno di strutture dati per mantenere informazioni sulle risorse gestite, queste strutture dati comprendono:

- Tabelle di memoria, gestione della memoria e allocazioni
- Tabelle di I/O, informazioni sullo stato di assegnazione dei dispostivi e gestione di code di richieste
- Tabelle del file system, elenco dei dispositivi utilizzati per mantenere il file system, e inoltre elenco dei file aperti e loro stato
- Tabelle dei processi



Un **Process Control Block** PCB è un descrittore di un processo ed è responsabile di:

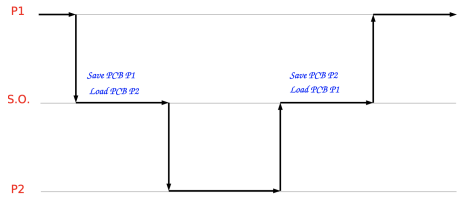
- Codice da eseguire
- Dati su cui operare
- Uno stack di lavoro per la gestione di chiamate di funzione, passagio di parametri e variabili locali
- Attributi contenenti le informazioni necessarie per la gestione del processo stesso

La tabella per la gestione dei processi contiene, i descrittori dei processi. Ogni processo ha un **PCB** assegnato. E' possibile suddividere le informazioni contenute nel descrittore in 3 aree:

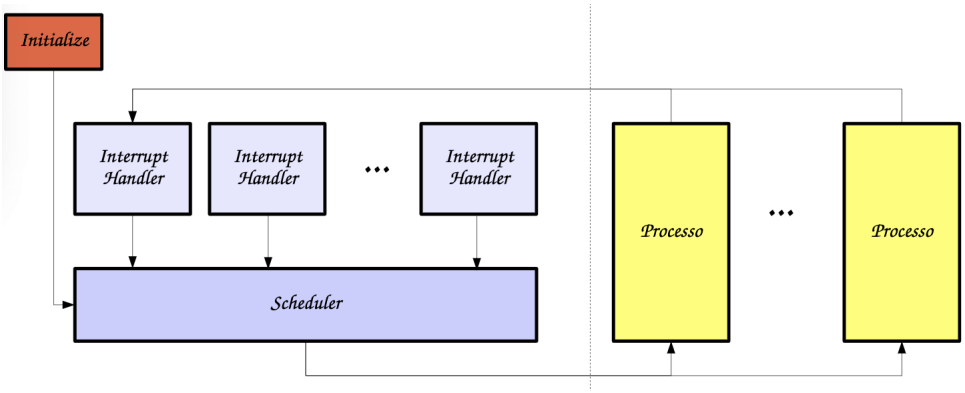
- Informazioni di identificazione di processo (id, pid) può essere semplicemente un indice all'interno di una tabella di processi
- Informazioni di stato di un processo, che possiede informazioni riguardanti registri generali del processo e registri speciali (program counter, registri di stato), informazioni di scheduling
- Informazioni di controllo del processo, dove sono presenti informazioni di gestione della memoria (MMU) come per esempio puntatori alle tabelle, informazioni di acocunting, informazioni relative alle risorse e ai semafori

Lo **scheduler** è la componente più importante del kernel. Esso gestisce l'avvicendamento dei processi, e quando viene richiamato decide quale processo deve essere in esecuzione. Interviene quando viene richiesta o termina una operazione di I/O. Lo **schedule** è la sequenza temporale di assegnazione delle risorse. Lo **scheduling** è l'azione di calcolare uno schedule. Lo **scheduler** è la componente software che calcola lo schedule.
Tutte le volte che avviene un interrupt, il processore è soggetto ad un **mode switching** (modalità utente → modalità supervisore). Durante la gestione

dell'interrupt se lo schedule decide di eseguire un altro processo, il sistema è soggetto ad un **context switching**. Durante le operazioni di un context switching viene **salvato** lo stato del processo attuale nel PCB corrispondente e successivamente lo **carica** per l'esecuzione

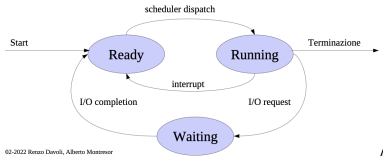


10.0.1 Scheda di funzionamento di un kernel



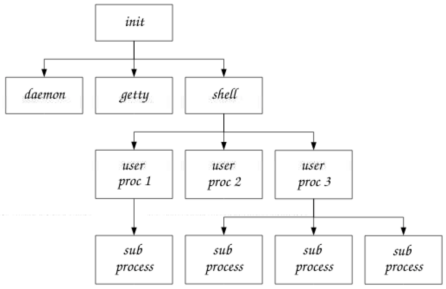
Un processo può avere diversi stati, i principali sono:

- **Running**: il processo è in esecuzione
- **Waiting**: il processo è in attesa di qualche evento esterno e non può essere eseguito
- **Ready**: il processi può essere eseguito, ma attualmente il processore è impegnato in altre attività



Tutte le volte che un processo entra nel sistema, viene posto in una delle code gestite dallo scheduler. Nella maggior parte dei S.O i processo sono formati in forma gerarchica, quando un processo crea un nuovo processo, il processo creante viene detto padre e il creato figlio, si va a creare in questo modo un albero di processi

10.0.2 Gerarchia processi UNIX

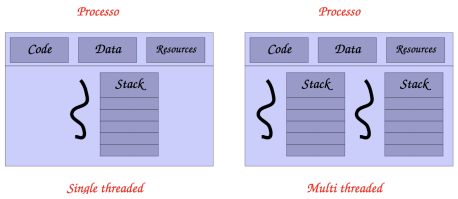


La nozione di processo discussa in precedenza assume che ogni processo abbia una singola linea di controllo Un **thread** è l'unità base di utilizzazione della CPU. Ogni thread possiede:

- La propria copia dello stato del processore
- Il proprio program counter
- Uno stack separato

I thread appartenenti allo stato processo condividono inoltre:

- Codice
- Dati
- Risorse di I/O



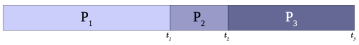
Utilizzando i thread si hanno benefici i quali:

- condividono lo spazio di memoria e le risorse allocate degli altri thread ello stesso processo, rendendo più semplice l'implementazione di certe applicazioni
- Economia, allocare memoria e risorse per creare nuovi processi è costoso. e anche context switching lo è
- Gestire i thread è in generale più economico, quindi è preferibile creare thread e fare context switching all'interno di un processo
- Rende l'implementazione più efficiente

Un sistema operativo può implementare i thread in **User thread** (livello utente) e Kernel thread (livello kernel). Gli user thread vengono supportati sopra il kernel e vengono implementati da una **thread library** a livello utente molto efficiente, ma può causare blocchi all'intero sistema. I kernel thread invece vengono supportati dal sistema operativo e non si hanno problemi di precedenza; se un thread eseguie una operazione di I/O, il kernel può selezionare un altro thread in attesa di essere eseguito ma questo va a discapito della velocità. Ad oggi vengono utilizzati maggiormente **kernel thread**. Gli user thread vengono mappati su un kernel thread e questo può creare problemi di scalabilità.

11 Scheduler 2

Per rappresentare uno schedule si usano i diagrammi di **Gantt**. Nel caso si debba rappresentare lo schedule di più risorse il diagramma di Gantt risulta composto da più righe parallele.



Gli eventi che possono causare un context switch sono:

- Quando un processo passa da stato running a stato waiting (system call bloccante)
- Qaundo un processo passa dallo stato running allo stato ready (interrupt)
- Quando un processo passa dallo stato waiting allo stato ready
- Quando un processo termina

Uno scheduler si dice **non-preemptive** o cooperativo se:

- Se i context switch avvengono solo nelle condizioni 1 e 4
- Il controllo della risorsa viene trasferito solo se l'assegnatario attuale lo cede volontaariamente

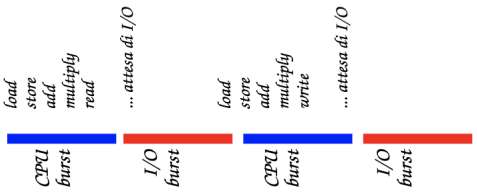
Uno schedule si dice **preemptive** se:

- Se i context switch possono avvenire in ogni condizione
- E' possibile che il controllo della risorsa venga tolto all'assegnatario attuale a causa di un evento
- Tutti gli scheduler moderni

Lo scheduler cooperativo **non richiede** alcuni meccanismi hardware, come ad esempio timer programmabili. Lo scheduling preemptive invece permette di utilizzare al meglio le **risorse**. I criteri di scelta di uno scheduler sono:

- **Utilizzo della risorsa (CPU)**, la percentuale di tempo in cui la CPU è occupata ad eseguire processi deve essere massimizzata
- **Throughput**, il numero di processi completati per unità di tempo dipende dalla lunghezza dei processi e deve essere massimizzato
- **Tempo di turnaround**, tempo che intercorre dalla sottomissione di un processo alla sua terminazione deve essere massimizzato
- **Tempo di attesa**, il tempo trascorso da un processo nella coda ready deve essere massimizzato
- **Tempo di risposta**, tempo che intercorre fra la sottomissione di un processo e il tempo di prima risposta particolarmente significativo nei programmi interattivi, deve essere minimizzato

Durante l'esecuzione di un processo si alternano periodi di attività svolte dalla CPU (**CPU burst)** e periodi di attività di I/O (**I/O burst**). I processi caratterizzati da CPU burst molto lunghi si dicono **CPU bound** mentre quelli molto brevi si dicono **I/O bound**



Esistono diversi algoritmi di scheduling:

- **First Come, First Served** (FIFO), dove il processo che arriva per primo, viene servito per primo. Questo causa elevati tempi medi di attesa e di turnaround e i processi CPU bound ritardano i processi I/O bound

- **Shortest-Job First**, dove la CPU viene assegnata al processo ready che ha la minima durata del CPU burst successivo. Questo algoritmo è **ottimale** rispetto al tempo di attesa, in quanto è possibile dimostrare che produce il minor tempo di attesa ma è **impossibile da implementare** è possibile solo fornire delle **approssimazioni** attraverso delle medie approssimate. SJF può essere soggetto a **starvation**

- **Round-Robin**, è basato sul concetto di quanto di tempo (time slice), un processo non può rimanere in esecuzione per un tempo superiore alla durata del quanto di tempo (democratico).
  L'implementazioni è organizzata come una **coda**, un processo può lasciare il processo **volontariamente** in seguito un'operazione di I/O oppure può **esaurire il suo quanto di tempo** senza completare il suo CPU burst, nel qual caso viene aggiunto in fondo alla coda dei processi pronti.
  Se il quanto di tempo è **breve** il sistema è **meno efficiente** perché deve **cambiare il processo** attivo più spesso, se è **lungo** in presenza di numerosi processi pronti ci sono **lunghi** periodi di **inattività** di ogni singolo processo.

Le proprietà possono essere **statiche** dove la priorità non cambia durante la vita di un processo al contrario della **dinamica**:

- **Definite dal sistema operativo**
- **Definite esternamente**

La priorità basata su **aging**, è una tecnica che consiste nell'incrementare gradualmente la proprietà dei processi in attesa, posto che il range di variazione della priorità sia limitato, nessun processo rimarrà in attesa per un tempo indefinito.
Nello scheduling a **classi di priorità** è possibile creare diverse classi di processi con caratteristiche simili e assegnare ad ogni classe specifiche priorità. Utilizzando inoltre il **multilivello** è possibile associare una politica specifica adatta alle caratteristiche della classe. Uno scheduler multilivello cerca prima la classe di priorità massima che ha almeno un processo ready.
Un altro algoritmo di scheduling è quello **Real-time**, in questo caso la correttezza dell'esecuzione non dipende solamente dal valore del risultato, ma anche dall'istante temporale nel quale il risultato viene emesso. Sono presenti due modalità la quale **hard real-time** nelle quali la deadline di esecuzione dei programmi non devono essere superate in nessuno caso (cruise-control), e **soft real-time** nella quale sono concessi errori occasionali.

12 Gestione risorse e deadlock

Un sistema di elaborazione è composto da un insieme di risorse da assegnare ai processi presenti. I processi competono nella'accesso alle risorse (memoria, stampanti, processore, dischi...). Le risorse possono essere suddivise in classi, le risorse appartenti alla stessa classe sono equivalenti. Le risorse di una clase vengono dette **istanze** della classe. Il numero di risorse in una classe viene detta **molteplicità** del tipo di risorsa. Un processo non può richiedere una specifica risorsa, ma solo una risorsa di una specifica classe.

Le risorse possono essere di **assegnazione statica**, che avviene al momento della creazione del processo e rimane valida fino alla terminazione (descrittore processi), e risorse ad **assegnazione dinamica**, dove i processi richiedono le risorse durante la loro esistenza e le rilasciano quando non ne hanno più necessità (I/O, aree memoria).
Le richieste possono essere **singole** ad una risorsa, oppure **multiple** che si riferiscono a una o più classi.
Le richieste possono essere anche **bloccante** dove il processo richiedente si sospende se non ottiene immediatamente l'assegnazione, e richieste **non bloccante**, dove la mancata assegnazione viene notificata al processo richiedente, senza provocare la sospensione.
Le risorse possono essere **condivisibili**, dove una risorsa non può essere assegnata a più processi contemporaneamente (processi, stampanti, sezioni critiche), e risorse **condivisibili** (file di sola lettura).
Una risorsa si dice **prerilasciabile** se la funzione di gestione può sottrarla ad un processo prima che questo l'abbia effettivamente rilasciata. Il processo che subisce il prerilascio deve sospendersi e la risorsa prerilascita sarà successivamente restituita al processo. Quindi una risorsa è prerilasciabile se il suo stato **non si modifica** durante l'utilizzo, oppure il suo stato può essere facilmente salvato e ripristinato (processore, blocchi partizione).
Sono risorse non prerilasciabili le risorse il cui stato non può essere salvato e ripristinato (stampanti, sezioni critiche).

12.1 Deadlock

I deadlock impediscono ai processi di terminare correttamente. Le risorse bloccate in deadlock non possono essere utilizzate da altri processi. Le condizioni necessarie e sufficienti per avere un deadlock sono:

- **Mutua esclusione / non condivisibili**, le risorse coinvolte devono essere non condivisibili
- **Assenza di prerilascio**, le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano
- **Richieste bloccanti (hold and wait)** le richieste devono essere bloccanti, e un processo che ha già ottenuto risorse può chiederne ancora
- **Attesa circolare**, sono necessario una sequenza di processi dove una ogni processo ha bisogno in senso circolare la richiesta seguente

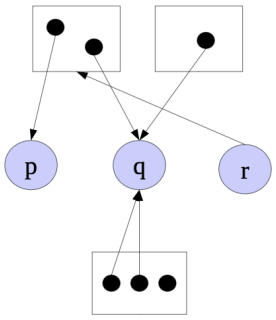
Devono valere tutte contemporaneamente affinché un deadlock si presenti nel sistema.

12.2 Grafo di Holt

Il grafo di Holt è un grafo:

- **diretto**, gli archi hanno una direzione
- bipartito, i nodi sono suddivisi in due sottoinsieme chiamati insiemi delle **risorse** e insieme dei **processi** e possono andare in direzione **unidirezionale**
- gli archi **risorsa** $\rightarrow$ **processo**, indicano che la risorsa è assegnata al processo
- gli archi **processo** $\rightarrow$ **risorsa**, indicano che il processo ha richiesto la risorsa

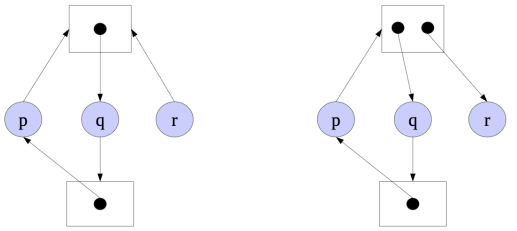
Nel caso di classi contenenti più istanze di una risorsa, l'insieme delle risorse è partizionato in classi e gli archi di richiesta sono diretti alla classe e non alla singola risorsa.
I **processi** sono rappresentati da cerchi, le **classi** sono rappresentati come contenitori rettangolari e le **risorse** come punti all'interno delle classi.



Nell'implementazione il grado di Holt viene memorizzato come grafo pesato (con pesi ai nodi risorsa e pesi sugli archi).
Per gestire i deadlock utilizziamo metodi i quali:

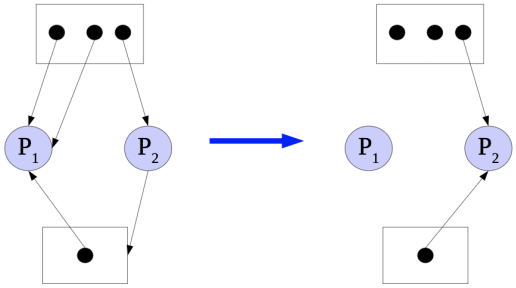
- Deadlock detection and recovery, utilizza metodi di recovery per recuperare l'ultimo stato corretto per riconoscere gli stati di deadlock, viene aggiornato continuamente il grafo di Holt registrando le assegnazioni
- Deadlock prevention /avoidance, impedisce di entrare in uno stato di deadlock
- Ostrich algorithm (algoritmo dello struzzo), ignora il problema

Se le risorse sono ad accesso mutualmente esclusivo, non condivisibili e non prerilasciabili lo stato di deadlock **se e solo se** il grafo di Holt contiene un ciclo. La presenza di un cliclo nel caso di Holt **non** è condizione sufficiente per avere deadlock



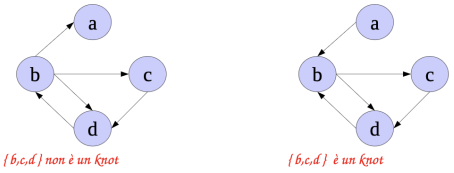
Deadlock *No Deadlock*

Un grafo di Holt si dice **riducibile** se esiste almeno un nodo process ocon solo archi entranti. Consiste nell'eliminare tutti gli archi di tale nodo e riassegnare le risorse ad altri processi.



Riduzione per P_1

Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, lo stato non è di deadlock se e solo se il grafo di Holt è **completamente riducibile**, esiste una sequenza di riduzioni che elimina tutti gli archi del grafo. Dato un nodo n, l'insieme dei nodi raggiungibili di n viene detto **insieme di raggiungibilità** di n ($R(n)$), partendo da un qualunque nodo di M, si possono ragigungere tutti i nodi di M e nessun nodo all'infuori di esso.



{ b,c,d } non è un knot *{ b,c,d } è un knot*

Dato un grado di Holt con una sola richiesta sospesa per processo, se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, allora il grafo rappresenta uno stato di deadlock se e solo se esiste uno knot. Dopo aver rilevato un deadlock bisogna risolvere la situazione, e la soluzione può essere:

- **Manuale**, l'operartore viene informato e eseguirò alucni azioni che permetteranno al sistema di proseguire

- **Automatica**, il S.O è dotato di meccanismi che permettono di risolvere in modo automatico la situazione, in base ad alcune politiche

Nel caso in cui avvenga un deadlock, **tutti i processi vengono terminati** oppure viene eliminato **un processo alla volta**, fino a quando il deadlock non scompare. Lo stato dei processi viene periodicamente salvato su disco (**checkpoint**), in caso di deadlock si ripristina (**rollback**) uno o più processi ad uno stato precedente. Per evitare il deadlock si elimina una delle **quattro** condizioni, in questo modo viene eliminato **strutturalmente**. Prima di assegnare una risorsa ad un processo, si controlla se l'operazione può portare al pericolo di deadlock. Nel metodo di **deadlock prevention** possiamo attaccare:

- La condizione di mutua esclusione, e permettere la condivisione di risorse, (spool di stampa, tutti i processi pensano di usare contemporaneamente la stampante). Lo **spooling** non sempre è applicabile (descrittori). Si sposta il problema verso altre risorse.
- La condizione di richiesta bloccante, dove è possibile richiedere che un processo richieda **tutte le risorse** all'inizio della computazione (riduzione parallelismo).
- La condizione di assenza di prerilascio
- La condizione di attesa circolare, dove alle classi di risorse vengono associati valori di priorità (**allocazione gerarchica**). Ogni processo in ogni istante può allocare solamente risorse di priorità superiore a quelle che già possiede (altamente inefficienti).

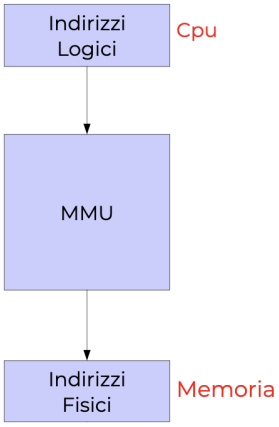
Dijkstra ha sviluppato un algoritmo chiamato **Algoritmo del banchiere** per evitare lo stallo svillupato.

13 Gestione della memoria

La parte del sistema operativo che gestisce la memoria principale si chiama **memory manager**. Il memory manager si occupa di tenere traccia della memoria **libera** e **occupata**, e allocare memoria ai processi e deallocarla quando non più necessita. Il memory manager è software. la Memory Management Unit (**MMU**) è **hardware**.
Con il termine **binding** si indica l'associazione di indirizzi logici di memoria (nomi, variabili) ai corrispondenti indirizzi fisici. Il binding può avvenire:

- Durante la compilazione
- Durante il caricamento
- Durante l'esecuzione

Durante la compilazione gli indirizzi vengono calcolati e resteranno gli stessi ad ogni esecuzione del programma. Il codice generato viene detto codice **assoluto** (kernel). Il codice generato dal compilatore non contiene indirizzi assoluti ma relativi, questo titpo di codice viene detto **rilocabile**; durante il caricamento il **loader** si preoccupa di aggiornare tutti i riferimenti agli indirizzi di memoria coerentemente al punto iniziale di caricamento. Durante l'esecuzione l'individuazione dell'ìndirizzo di memoria effettivo viene effettuata durante l'esecuzione da un componente hardware apposito: la **memory management unit** (MMU), da non confondere con il memory manager (MM).
Nello **spazio di indirizzamento logico** ogni processo è associato ad uno spazio di indirizzamento logico, cioè riferimenti a spazi di indirizzamento. Nello **spazio di indirizzamento fisico**, ad ogni indirizzo logico corrisponde un indirizzo fisico, e la MMU opera come una funzione di traduzione da indirizzi logici a indirizzi fisici.



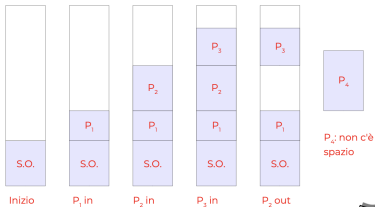
Il **loading dinamico** consente di poter caricare alcune routine di libreria solo quando vengono richiamate. Tutte le routine a caricamento dinamico risiedono su un disco, e quando servono vengono caricate. Le routine poco utili non vengono caricate in memoria al caricamento dell'applicazione. Il sistema operativo fornisce semplicemente una libreria che implementa le funzioni di caricamento dinamico, spetta al **programmatore** sfruttare questa possibilità.
Il linking può essere:

- **linking statico** collega e risolve tutti i riferimenti dei programmi e le routine di libreria vengono copiate in ogni programma che le usa.
- **linking dinamico**, posticipa il linking delle routine di libreria al momento del primo riferimento durante l'esecuzione, consentendo di avere eseguibili più compatti. Esiste una sola istanza della libreria in memoria. Questo porta ad un **risparmio di memoria**, ma può causare problemi di **versioning**

13.1 Allocazione di memoria

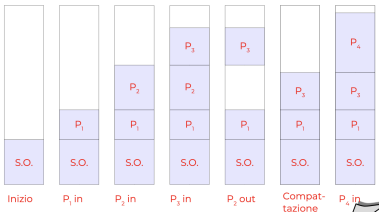
L'allocazione è una delle funzioni principali del **gestore di memoria**. Consiste nel reperire ed assegnare uno spazio di memoria fisica.
Nella allocazione **contigua** tutto lo spazio assegnato ad un processo deve essere formato da celle consecutive, al contrario della memoria **non contigua**. La MMU deve essere in grado di gestire la conversione degli indirizzi.
Una allocazione può essere **statica** se un processo deve mantenere la propria area di memoria dal caricamento alla terminazione, mentre **dinamica**, se durante l'esecuzione, un processo può essere spostato all'interno della memoria.
La memoria disponibile viene suddivisa in **partizioni**. Ogni processo viene caricato in una delle partizioni libere che ha dimensione sufficiente a conternerlo.

Nelle allocazione a **partizione fisse**, se un processo occupa una dimensione inferiore a quella della partizione che lo contiene, lo spazio non utilizzato è sprecato. La presenza di spazio inutilizzato si chiama **frammentazione interna**.
Nella allocazione a **partizione dinamica**, la memoria disponibile viene assegnsata ai processi che ne fanno richiesta. Nella memoria possono essere presenti diverse zone inutilizzate, per effetto della terminazione di processi oppure per non completo utilizzo. Essa è statica e contigua.

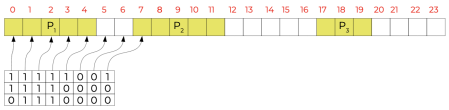


Dopo un certo numero di allcoazioni e deallocaizoni di memoria dovute all'attivaizone e alla terminazione dei processi lo spazio libero appare suddiviso in piccole aree, è il fenomeno della **frammentazione esterna**.

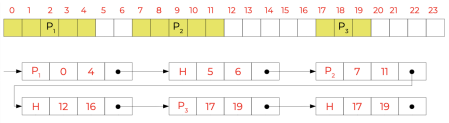
La **compattazione**, permette di rilocare i processi durante la loro esecuzione. Questo operazione risolve il problema della frammentazione esterna. Essa però è una operazione molto onerosa e i processi devono essere **fermi** durante la compattazione (interettivi).



Quando la memoria è assegnata dinamicamente abbiamo bisogno di una struttura dati per manterenere informazioni sulle zone libere e sulle zone occupate. Quindi è possibili utilizzare **mappe di bit**, **liste con puntatori**, ecc.. La mappa di bit viene suddivisia in unità di allocazione e ad ogni unità di allocazione corrisponde un bit in una **bitmap**.



Nelle liste di puntatori si mantiene una lista dei blocchi allcoati e liberi di memoria. Ogni elemento della lista specifica se si tratta di un processo (P) o un blocco libero (H) e la sua dimensione.



L'operazione di selezione di un blocco libero è concettualemente indipendente dalla struttura dati. Per scegliere un blocco di memoria libera possiamo adottare:

- **First fit**, dove si scorre la lista dei blocchi liberi fino a quando non trova il primo segmento vuoto grande abbastanza da contenere il processo
- **Next fit**, come First Fit, ma invece di ripartire sempre dall'inizio, parte dal punto dove si era fermato all'ultima allcocazione.

First fit ha performance **migliori** di next fit.

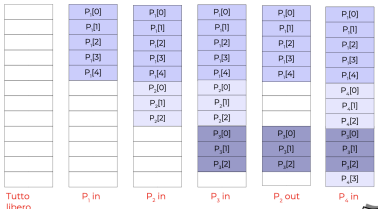
13.2 Paginazione

I meccanismi visti (partizioni fisse e dinamiche) non sono efficienti nell'uso della memoria:

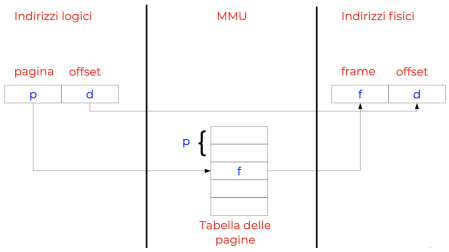
- Frammentazione interna

- Frammentazione esterna

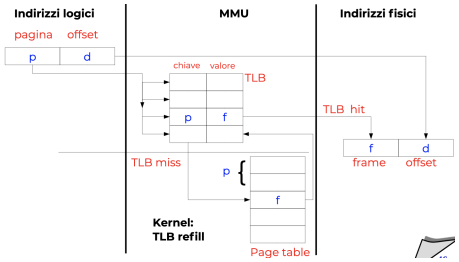
La paginazione **riduce** il fenomeno di frammentazione interna e minimizza il fenomeno della frammentazione esterna. Lo spazio di indirizziamnto logico di un processo viene suddiviso in un insieme di blocchi di dimensione fissa chiamati **pagine**. La **memoria fisica** viene suddivisa in un insieme di blocchi della stessa dimensione delle pagine, chiamati **frame**.



La dimensione delle pagine deve essere una potenza di due, per semplificare la trasformazione da indirizzi logici a indirizzi fisici. La scelta della dimensione deriva da un trade-off, con pagine piccole le tabelle delle pagine cresce di dimensione, mentre con pagine troppo grandi lo spazio di memoria perso per frammentazione può essere considerevole. Solitamente misura 1KB,2KB,4KB.



La tabella può essere contenuta in un insieme di registri ad alta velocità all'interno del modulo MMU. Un **Translation lookaside buffer** (TLB) è costituito da un insieme di registri associativi ad alta velocità. Ogni registro è suddiviso in due parti, una chiave e un valore. Nella operazione di **lookup** viene richiesta la ricerca di una chiave. Se la chiave è presente si ha un **TLB hit**, mentre se non è presente si ritorna un **TLB miss**.



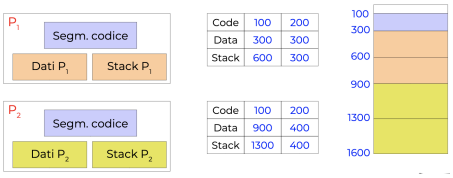
La TLB agisce come **memoria cache** per le tabelle delle pagine. La TLB si basa sul principio di località, ma è costoso. Il numero di registri varia da $8 - 2048$.

In un sistema basato su segmentazione, uno spazio di indirizzamento logico è dato da un insieme di segmenti. Ogni segmento è caratterizzato da un **nome** (un indice) e da una **lunghezza**. Spetta al programmatore o al compilatore la suddivisione di un programma in segmenti

13.3 Confronto paginazione e segmentazione

- **Paginazione**
  - la divisione in pagine è automatica.
  - le pagine hanno dimensione fissa
  - le pagine possono contenere informazioni disomogenee (ad es. sia codice sia dati)
  - una pagina ha un indirizzo
  - dimensione tipica della pagina: 1-4 KB
- **Segmentazione**
  - la divisione in segmenti spetta al programmatore.
  - i segmenti hanno dimensione variabile
  - un segmento contiene informazioni omogenee per tipo di accesso e permessi di condivisione
  - un segmento ha un nome.
  - dimensione tipica di un segmento: 64KB – molti MB

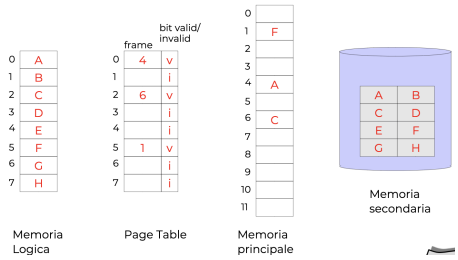
La segmentazione consente la condivisione di codice e dati.



E' possibile utilizzare il metodo della paginazione **combinato** al metodo della segmentazione. Ogni segmento viene suddiviso in **pagine** che vengono allocate in frame liberi della memoria.

13.4 Memoria virtuale

La memoria virtuale è la tecnica che permette l'esecuzione di processi che non sono completamente in memoria. Essa permette di eseguire in concorrenza processi che nel loro complesso hanno necessità di memoria maggiore di quella disponibile. La memoria virtuale però **diminuisce le prestazioni** di un sistema. Ogni processo ha un accesso ad uno **spazio di indirizzamento virtuale** che può essere più grande di quello fisico. Nella sua implementazione si utilizza la **paginazione a richiesta** (**demand paging**), ammettendo però che alcune pagine possano essere in memoria secondaria. Nelle tabella delle pagine si utilizza un **bit v (valid)** che indica se la pagina è presente in memoria centrale oppure no. Quando un processo tenta di accedere ad una pagina non in memoria il processore genera un trap (**page fault**) e un componente del S.O (**pager**) si occupa di caricare la pagina mancante in memoria, e di aggiornare di conseguenza la tabella delle pagine.

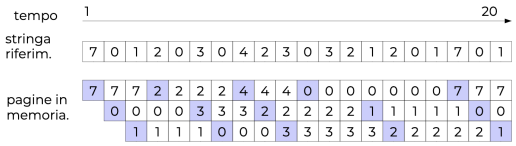


Con il termine **swap** si intende l'azione di copiare l'intera area di memoria usata da un processo:

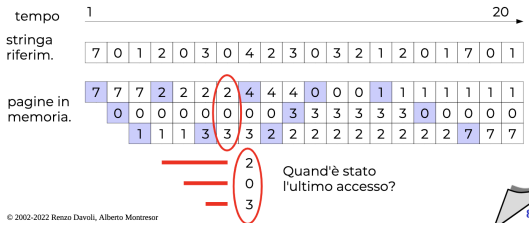
- Dalla memoria secondaria alla memoria principale (**swap-in**)
- Dalla memoria principale alla memoria secondaria (**swap-out**)

Può esserci anche una paginazione su richiesta vista come una tecnica di wap di tipo **lazy** dove viene caricato solo ciò che serve. Il termine **swap area** è utilizzato per indicare l'area del disco utilizzata per ospitare le pagine in memoria secondaria. In mancanza di frame liberi occorre liberarne uno, viene quindi utilizzato un algoritmo di **rimpiazzamento** che minimizza il numero di page fault.
I principali algoritmi sono:

- **FIFO** dove viene liberato il frame che per primo fu caricato in memoria. Semplice implementazione ma talvolta vengono scaricare pagine che sono sempre utilizzate



- **LRU** (Least Recently Used), seleziono la pagina che è stata usata meno recentemente nel passato



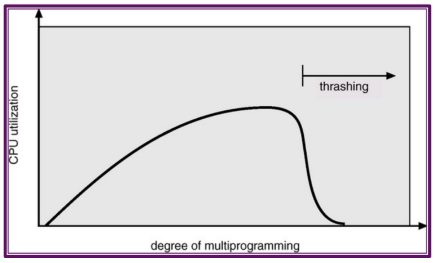
- **Stack**, data una stringa di riferimenti s si indica con $S_t(s, A, m)$ l'insieme delle pagine mantenute, al tempo t dell'algoritmo A, data una memoria di

m frame relativi alla stringa s. Un algoritmo di rimpiazzamento è detto **stack algorithm** se l'insieme delle pagine in memoria con m frame è sempre un sottoinsieme delle pagine in memoria con $m+1$ frame.

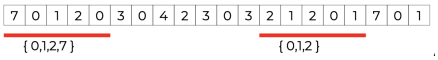
$$S_t(s, A, m) \subseteq S_t(s, a, m+1)$$

- **LFU** si mantiene un contatore del numerodi di accessi ad una pagina, e la pagina con il valore minore viene scelta come vittima. Se una pagina viene utilizzata frequentemente all'inizio, e poi non viene più usata, non viene rimossa per lunghi periodi

Con algoritmo di allocazione si intende l'algoritmo utilizzato per scegliere quanti frame assegnare ad ogni singolo processo. Un processo si dice che è in **trashing** quando spende più tempo per la paginazione che per l'esecuzione.

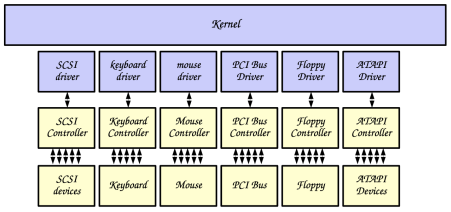


Si dice **working set di finistra**, l'insieme delle pagine accedute nei più recenti Δ riferimenti. Se una pagina non compare in Δ riferimenti successivi in memoria, allora esce dal working set; non è più una pagina su cui si lavora attivamente. Con $\Delta = 5$ abbiamo:



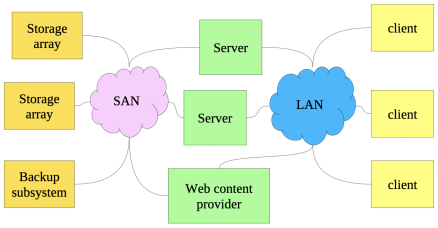
Il working set serve ad evitare il **trashing**, esso da una approssimazione dell'insieme delle pagine utili. Sommando quindi l'ampiezza di tutti i working set dei processi attivi, questo valore deve essere sempre minore del numero di frame disponibili. Per uscire dalla situazione del trashing si **fermano i processi**.

14 Memoria secondaria



I dati vengono scritti/letti a **blocchi** (512-1024 byte). Vengono utilizzate diverse tecniche di gestione dei dispositivi di I/O:

- **Buffering**, per gestire una differenza di velocità tra il produttore e il consumatore di un certo flusso di dati, gestire la differenza di dimensione nell'unità di trasferimento
- **Caching**, mantiene una copia di memoria primaria di informazioni che si trovano in memoria secondaria. La cache mantiene la copia di un'informazione
- **Spooling**, è un buffer che mantiene output per un dispositivo che non può accettare flussi di dati distinti (stampanti)
- **I/O scheduling**



Il gestore del disco può avere numerose richieste pendenti, da parte dei vari processi presenti nel sistema. Valori tipici del tempo di seek sono $8 - 10ms$ e velocità di rotazione pari a $5400 - 7200 - 10000rpm$. Anche in questo caso sono presenti tecniche di scheduling:

- **FCFS**, First Come, First Served (FIFO), con funzionamento di una coda. E' una politica di gestione fair, che non minimizza il numero di seek
- **SSTF**, Shortest Seek Time First, può provocare starvation, seleziona la richiesta che prevede il minor spostamento della testina dalla posizione corrente
- **LOOK**, Algoritmo dell'ascensore, ad ogni istante, la testina è associata ad una direzione, la testina si sposta di richiesta in richiesta. Quando si raggiunge l'ultima richiesta nella direzione scelta, la direzione viene invertita e si eseguono le richieste nella direzione opposta. Usato tuttora
- **C-LOOK**, ha lo stesso principio di funzionamento del metodo LOOK, ma la scansione del disco avviene in una sola direzione. Quando si raggiunge l'ultima richiesta in una direzione, la testina si sposta direttamente alla prima richiesta

14.1 RAID

Per aumentare la velocità di un componente, una delle possibilità è quella di utilizzare il **parallelismo**. L'idea è quella di utilizzare un array di dischi indipendenti, che possano gestire più richieste di I/O in parallelo. Dobbiamo però garantire che i dati letti in parallelo risiedano su dischi indipendenti.
L'utilizzo di più dischi aumenta le probabilità di guasto nel sistema. Per compensare questa riduzione di affidabilità, RAID utilizza meccanismi di parità. La tecnica di RAID si basa su **7 schedi diversi** (0-6):

- **RAID 0**, esso non presenta meccanismi di rindondanza. Può essere utilizzato per applicazioni in cui l'affidabilità non è un grosso problema, ma lo sono la velcoità e il basso costo. I dati vengono distribuiti su più dischi. Il sistema viene viesto come un disco logico, essi vengono suddivisi in **strip**. Utilizzato maggiormente per grandi trasferimenti di dati
- **RAID 1**, la ridondanza è ottenuta duplicando tutti i dati su due insiemi indipendenti di dischi. Il sistema è basato su **striping**, ma questa volta uno strip viene scritto su due dischi diversi. Una richiesta di lettura può essere servita da uno qualsiasi dei dischi che ospitano il dato. Una richiesta di scrittura deve essere servita da tutti i dischi che ospitano il dato
- **RAID 4**, Si utilizza il meccanismo di data striping, con strip relativamente grandi. Viene utilizzato uno **strip di parità** per verificare se sono presenti errori
- **RAID 5**, come RAID 4, ma i blocchi di parità sono sparsi fra i vari dischi. Il vantaggio è che non esiste un disco di parità che diventa un **bottleneck**
- **RAID 6**, come RAID 5, ma si utilizzano due strip di parità invece di uno. Aumenta l'affidabilità (è necessario il gusto di tre dischi affinche i dati non siano utilizzabili)

15 File system

Il compito del **file system** è quello di astrarre la complessità di utilizzo dei diversi media proponendo una interfaccia per i sistemi di memorizzazione. Dal punto di vista dell'utente, un file system è composto da due elementi:

- **File**, unità logica di memorizzazione
- **Directory**, servono per organizzare e fornire informazioni sui file che compongono un file system

Alcuni S.O supportano e riconoscono diversi tipi di file, conoscendo il tipo il S.O può evitare alcuni errori comuni, quali ad esempio stampare un file eseguibile. Esistono tre tecniche principali per identificare il tipo di un file:

- Meccanismo delle estensioni

- Utilizzo di un attributo `tipo` associato al file nella directory
- magic number

In un sistema operativo multitasking, i processi accedono ai file **indipendentemente**. In **UNIX** le modifiche al contenuto di un file aperto vengono rese visibili agli altri processi **immediatamente**. Esistono due tipi di condivisione del file:

- Condivisione del puntatore alla posizione corrente nel file (**fork**)
- Condivisione con distinti puntatori alla posizione corrente

Un disco può essere diviso in una o più **partizioni**, porizioni indipendenti del disco che possono ospitare file system distinti. Il primo settore dei dischi è il cosidetto **master boot record** (MBR):

- E' utilizzato per fare il boot del sistema
- Contiene la **partition table** (tabella delle partizioni)
- Contiene l'indicazione della partizione attiva

Al boot , il **MBR** viene letto ed eseguito.

Ogni partizione inizia con un **boot block**. Il MBR carica i il boot block della partizione attiva e lo esegue. Il boot block carica il **sistema operativo** e lo esegue, L'organizzazione del resto della partizione dipende dal file system

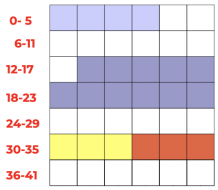
In generale:

- **Superblock**, contiene informazioni sul tipo di file system e sui parametri fondamentali della sua organizzazione
- **Tabelle per la gestione dello spazio libero**, struttura dati contenente informazioni sui blocchi liberi
- **Tabella per la gestione dello spazio occupato**, contiene informazioni sui file presenti nel sistema, e non è presente in tutti i file system
- **Root dir**, directory radice (del file system)
- **File e directory**

15.1 Allocazione contigua

I file sono memorizzati in sequenze contigue di blocchi di dischi. Non è necessario utilizzare strutture dati per collegare i blocchi. L'accesso sequenziale è efficiente, non è necessario eseguire operazioni di seeek.
Come svantaggio abbiamo le problematiche dell'allocazione contigua in memoria centrale:

- Frammentazione esterna
- Politica di scelta delle aree di blocchi liberi da usasre per allocare spazio per un file: first fit, best fit, worst fit...
- I file non possono crescere

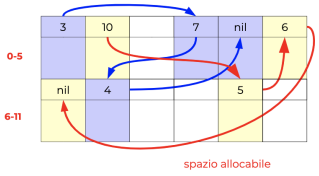


directory

Name	Start	Size
a	0	4
b	13	11
c	30	3
d	33	3

15.2 Allocazione concatenata

Ogni file è costituito da una lista concatenata di blocchi. Ogni blocco contiene un puntatore al blocco successivo. Il descrittore del file contiene i puntatori al primo e all'ultimo elemento della lista. Questo risolve il problema della frammentazione esterna e l'accesso sequenziale è efficiente.
Come svantaggio abbiamo un accesso diretto inefficiente. Progressivamente l'efficienza globale del file system degrada. Se il blocco è piccolo (512 byte) l'overhead per i puntatori può essere rilevante

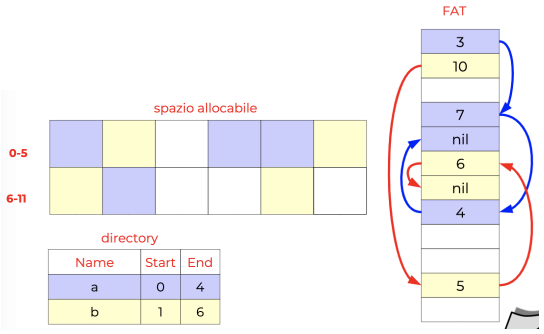


directory

Name	Start	End
a	0	4
b	1	6

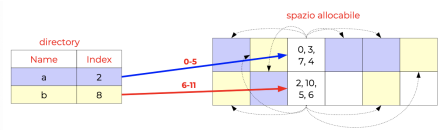
15.3 Allocazione basata su FAT

Invece di utilizzare parte del blocco dati per contenere il puntatore al blocco successivo si crea una tabella unica con un elemento per blocco (**cluster**). I blocchi sono dati interamente dedicati ai dati. E' possibile fare caching in memoria dei blocchi FAT. L'accesso diretto diventa così più efficiente, in quanto la lista puntatori può essere seguita in memoria (utilizzato da DOS, **chiavette**, ecc.).
Lo svantaggio però è che la scansione richiede anche la lettura della FAT, aumentando così il numero di accessi al disco.



15.4 Allocazione indicizzata

L'elenco dei blocchi che compongono un file viene memorizzato in un blocco indice. Per accedere ad un file, si carica in memoria la sua area indice e si utilizzano i puntatori contenuti.
Questa metodologia risolve il problema della frammentazione esterna. E' efficiente per l'accesso diretto. Il blocco indice deve essere caricato in memoria solo quando il file è aperto.
Come svantaggio abbiamo che la dimensione del blocco indice determina l'ampiezza massima del file. Utilizzare blocchi indici troppo grandi comporta un notevole spreco di spazio.



15.5 Allocazione indicizzata in UNIX

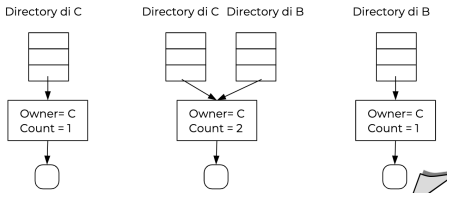
In UNIX ogni file è associato ad un i-node (index node). Un **i-node**, è una struttura dati contenente gli attributi del fie, e un indice di blocchi diretti e indiretti, secondo uno schema misto.

15.6 Implementazione delle directory

Una directory è un **file speciale** contentente informazioni sui file contenuti nella directory. Una directory è suddivisa in un certo numero di directory entry. La directory entry contiene tutte le informazioni necessarie associate ad un file. Le infomrazioni sono contenute negli **inode**; una directory entry contiene un indice di inode (utilizzata in UNIX). Una directory può essere implementata tramite **lista lineare** oppure tramite **tabella hash** dove occorre stabilire la dimensione della tabella di hash e il metodo di gestione delle collisioni. Sono inoltre disponibili due ulteriori implementazioni a grafo aciclico:

- **Link simbolici**, viene creato un tipo speciale di directory entry, che contiene un riferimento al file in questione
- **Hard link**, le informazioni relative al file sono presenti, uguali, in entrambe le directory. Non è necessario una doppia ricerca nel file system. E' impossibile distinugere la copia dell'originale.

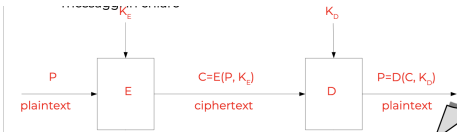
Una struttura a grafo diretto aciclico è più flessibile di un albero, ma crea tutta una serie di problemi dell'originale. Negli hard-link è necessario utilizzare la tecnica degli i-node. Gli i-node devono contenere un contatore di riferimenti.



Per migliorare la performance dei file system possono essere utilizzati meccanismi di caching. Esistono anche file system basati su log (tipo journaling), dove ogni aggiornamento al file system è trattato come una **transazione**. Una transazioni è un'operazione che viene eseguita in modo atomico. Consente di ripristinare rapidamente uno stato coerente. Nei **log** vengono salvate tutte le transizioni.

16 Sicurezza

E' il problema generale, che coinvolge non solo il sistema informatico. Una **security policy** descrive i requisiti di sicurezza policy. I **meccanismi** implementano la security policy. La scelta di una politica di sicurezza dipende da il tipo di attacchi e il valore delle infomazioni contenute nel sistema.
L'insieme dei meccanismi utilizzati per trasformare i messaggi in chiaro **plaintext** in un messaggio cifrato **ciphertext**.



Vengono utilizzate funzioni a **senso unico** cioè che possono essere calcolate in modo semplice ma ristabilire il numero risulta computazionalmente impossibile. Esistono due tipi principali di algoritmi di crittografia:

- Crittografia a **singola chiave**, utilizzano la stessa chiave sia per criptare e sia per decriptare. In questo modo però se viene acquisita una chiave è possibile decriptarla.
- Crittografia a doppia chiave, che utilizza due chiavi distinte, e quindi una chiave per criptare **chiave privata** e una per decriptare **chiave pubblica**. In questo modo posso pubblicare la chiave criptata senza problemi di sicurezza perché per decriptare la chiave è necessaria la privata.

La **security by obscurity**, accade quando la sicurezza viene mantenuta mantenendo segreti gli algoritmi di crittografia (cifrario di cesare).

16.1 DES

La crittografia **DES** (Data Encription Standard) è un algoritmo a chiave segreta di 56 bit. Dato un messaggio m di 64 bit lo si divide in due blocchi a 32 e si esegue una operazione di **XOR** per 16 volte. Ad oggi è abbastanza decriptabile tramite brute-force

16.2 RSA

La crittografia **RSA** prende due numeri primi molto grandi p e q. Si chiami $n = pq$ si scegli poi un valore d in modo tale che sia primo rispetto agli altri due valori presi in consideraizione. infine viene preso e cioè l'inverso moltiplicativo ottenuto dal modulo tra d e l'mcd dei primi due numeri primi.

16.3 Buffer overflow

L'idea generale è quella di fornire ad un programma (server) un insieme di dati di dimensioni superiori a quelle previste. Più del 50% degli incidenti sono dovuti al buffer overflow

16.4 Trojan horse

Sono programmi che replicano le funzionalità di programmi di uso comune che contengono codice *malefico*. Tipicamente catturano informazioni e le inviano al creatore del programma.
Un applicativo può leggere informazioni relative al nostro sistema inviandoli al creatore (spyware). Un altro esempio è l'inserimento di programmi nelle directory normalmente accedibili tramite il **path**, con il nome simile a quelli di programmi noti ($ls \to la$)

16.5 Bombe logiche

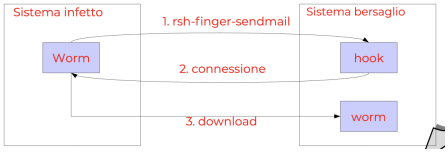
Il cretore di un software utulizzato internamente in una compagnia inserisce un software che può attivarsi sotto particolari condizioni. Ad esempio se il creatore viene licenzioato e il suo nome scompare dal database dei salari, non potendo più accedere al sistema, non può evitare che la bomba logica si attivi.

16.6 Backdoor/Trapdoor

Il creatore di un software può deliberatamente lasciare porte di servizio per entrare aggirando i sistemi di protezione. Come contromisura abbiamo l'analisi del codice sorgente.

16.7 Virus e worm

Un **virus** è un frammento di programma che può infettare altri programmi non maligni modificandoli, un **worm** è un programma che diffonde copie di se stesso in una rete. I worm operano sulle reti, i virus possono usare qualunque supporto. I worm sono programmi autonomi, i virus infettano programmi esistenti. Il primo worm conosciuto (1988) scritto da **Morris** sfruttava alcuni bug di sistema di comunicazione. Esso bloccò la rete per un giorno.



16.8 PAM

PAM (**Pluggable Authentication Module**) è un servizio generale di autenticazione basato su file di configurazione. Un file PAM (*/etc/pam.d/su*) è composto da diversi campi:

- **auth**, procedure per l'autenticazione dell'utente
- account, per modificare gli attributi di un account
- **password**, per modifcare la password
- session, per debug e loggin su syslog

I moduli vengono valutati in ordine, ogni modulo risponde con **grant** o **deny**. Il secondo campo dei file di configurazione:

- **sufficient**, se il modulo ritorna grant, la risposta è positiva e i moduli successivi non vengono considerati
- **requisite**, se il modulo ritorna deny, la risposta è negativa e i moduli successivi non vengono considerati

- **required**, questo modulo deve ritornare grant affinché l'operazione abbia successo
- optional, questo modulo influisce sul risultato solo se è l'unico modulo presente

16.9 Protezione del kernel dal sistema operativo

L'insieme dei meccanismi che separano il **gestore** dalle **risorse gestite** (processi, risorse), è realizzato tramite meccanismi:

- **Mode bit** (kernel/user), meccanismo delgi interrupt
- Protezione memoria e dispositivi

Esistono diversi principi fondamentali di controllo:

- Principio di **accesso mediato**, tutti gli accessi ad un oggetto devono essere controllati
- Principio di **separazione dei privilegi**, un sistema non dovrebbe concedere permetssi in base ad uan singola condizione. Esempio: UNIX permette ad un utente di diventare root se conosce la password di root e fa parte del gruppo *wheel*
- Principio di **failsave default**, nessun soggetto ha diritti per dafault
- Principio di **privilegio minimo**, ogni soggetto ha, in ogni istante, i soli diritti necessari per quella fase dell'elaborazione

In un sistema operativo UNIX:

- Soggetti: processi, thread
- Oggetti: file, dispositivi, processi
- Diritti di accesso: read, write, execute

Tramite il meccanismo delle **capability**, ad ogni dominio vinee associata una lista di **capability**, ovvero coppie: *¡oggetti, diritti di accesso¿*. I processi mantengono le capability e le presentano quali credenziali per accedere all'oggetto. Nel modello tradizionale UNIX viene utilizzata una **umask** per la creazione dei file. Tutti i bit che osno accesi nella maschera, verrano spenti nell'access mode del file creato.