Appunti di Sistemi Operativi T

Edoardo Carra'

May 17, 2022

```
#!/bin/bash
if [ -f prova.txt ]; then
    echo "Il file prova.txt esiste."
else
    echo "Il file prova.txt non esiste o non è un file"
fi
```

Abstract

Appunti di Sistemi Operativi T, anno accademico 2021/22, corso tenuto dalla professoressa Anna Ciampolini. Il link al corso di virtuale e alla stanza di Teams.

1 Introduzione - Il sistema operativo

E' un programma, o meglio un insieme di programmi, che agisce da intermediario tra l'utente e l'hardware del PC:

- Fornisce all'utente una visione astratta e semplificata dell'hardware.
- Gestisce in modo efficace ed efficiente le risorse del sistema.

Il sistema operativo interfaccia programmi applicativi o di sistema con le risorse hardware (CPU, memoria centrale, memoria secondaria, connessioni di rete). Inoltre, mappa le risorse hardware in risorse logiche, accessibili attraverso interfacce ben definite (processi, file system, memoria virtuale). Un sistema operativo deve avere le seguenti caratteristiche:

- Scelta dell'architettura;
- Capacità di condivisione;
- Efficienza;
- Estensibilità;
- Protezione e sicurezza
- Affidabilità/tolleranza ai guasti
- Conformità agli standard

In particolare le ultime 4 sono tipiche di un sistema di grande dimensioni

Il sistema operativo utilizza le API - application programming interface

1.1 L'evoluzione dei sistemi operativi

1. **Prima generazione**: controllo del sistema completamente manuale, linguaggio macchina. Non è presente alcun sistema operativo

2. **Seconda generazione**: sistemi batch semplici, linguaggio di alto livello, possibilità di effettuare operazioni di input mediante delle schede perforate. Aggregazioni di programmi in lotti (*batch = insieme di programmi(job) eseguiti in modo sequenziale*). In questo tipo di sistemi il compito unico del sistema operativo era quello di trasferire il controllo da un job, appena terminato, al prossimo da eseguire. Il sistema operativo risiede in memoria (monitor). I principali svantaggi sono: l'assenza di interazione con l'utente, inattività della CPU causata dalla sospensione di un job che attende un evento, sequenza.

3. **Multiprogrammazione:** per far fronte ai problemi sopra descritti, nascono i sistemi batch multiprogrammati: viene precaricato sul disco un insieme di job (pool).Il sistema operativo ha il compito di caricare in memoria centrale un sottoinsieme dei job precaricati. Tra i job che il sistema operativo ha caricato in memoria centrale, ne viene selezionato uno a cui verrà assegnata la CPU. Qualora il job corrente si pone in attesa di un evento,il sistema operativo assegna la CPU ad un altro job

   La cpu non esegue sequenzialmente i job, ma attraverso dei cambi di contesto(context switch) interrompe e riprende l'esecuzione dei job senza mai portare la CPU in uno stato di wait (perché il job si aspetta una risorsa per esempio). Esistono quindi due tipi di scheduling:

   - *Scheduling dei job(Long-term scheduling): Quali job trasferire dalla memoria secondaria a quella principale.*
   - *Scheduling della cpu(Short-term scheduling): Quali job assegnare alla CPU.*

   La filosofia di questi sistemi multiprogrammati è che se un job non può utilizzare La CPU allora si esegue un altro job. Attenzione! Ogni volta che si esegue un context-switch ovviamente c'è un costo temporale che va pagato.

4. **Sistemi time-sharing:** Nascono della necessità di avere dei sistemi che permettano maggiore *interattivita'* con l'utente, e la possibilità di gestire più utenti che interagiscono contemporaneamente con il sistema operativo *(multi-utenza).*

   - Multi-utenza: il sistema presenta ad ogni utente una macchina virtuale completamente dedicata, in termini di: utilizzo della CPU e di utilizzo di altre risorse.
   - Interattivita': per garantire un accettabile velocità di "reazione" alle richieste dei singoli utenti, il sistema operativo interrompe l'esecuzione di un job dopo un intervallo di tempo prefissato (timeslice), assegnando la CPU ad un altro job.

   Nota bene: Se alzo il timeslice l'interattività diminuisce.

1.2 Interruzioni

Le varie componenti hardware e software del sistema interagiscono con il sistema operativo attraverso delle interruzioni asincrone (Interrupt). Per la precisione ogni interruzione è causata da un evento (richiesta di servizi al SO, completamento operazioni I/O, accesso non consentito alla memoria). Ad ogni interruzione è associata una routine di servizio (handler) per la gestione dell'evento.

- Interruzioni hardware: i dispositivi inviano segnali alla CPU per notificare particolari eventi al sistema operativo.
- Interruzioni software(trap): i programmi in esecuzione possono generare delle interruzioni: quando tentano l'esecuzione di operazioni non lecite oppure quando richiedono l'esecuzione di servizi al sistema operativo (System call) tra quelle offerte dall'API del sistema

Non appena il sistema operativo riceve un'interruzione, interrompe l'esecuzione del processo chiamante, salvando lo stato in memoria, attiva la routine di servizio (handler) passando il controllo della CPU al sistema operativo, ed infine ripristina lo stato precedentemente salvato.

NB: Per individuare le routine di servizio adeguata, si utilizza un vettore delle interruzioni.
Missing drivers...

1.3 Protezione

Nei sistemi che prevedono multiprogrammazione e multiutenza sono necessari alcuni meccanismi di protezione. Nello specifico, le risorse allocate a programmi o utenti devono essere protette nei confronti di accessi illeciti di altri programmi/utenti.
Per prima cosa è necessario impedire al programma in esecuzione di accedere ad aree di memoria esterne al proprio spazio che il sistema operativo gli ha riservato.
In generale il set di istruzioni del processore è solitamente diviso in due set: privilegiate e non privilegiate. Solitamente le istruzioni del primo gruppo possono essere invocate dal sistema operativo (per esempio le istruzioni di shutdown, disabilitare le interruzioni ecc...). Come facciamo a garantire l'esclusività del sistema operativo dell'esecuzione delle istruzioni privilegiate? Lo facciamo a livello hardware. Si utilizzano in particolare dei ring di protezione, cioè dei bit di modo che settano il funzionamento della cpu(solitamente si trovano all'interno dei registri della CPU). Il bit quindi può essere ZERO - **Kernel mode** oppure UNO - **User mode**. La CPU quindi può eseguire istruzioni privilegiate solo in modalità kernel mode. Nel caso di accesso non permesso, il sistema risponde lanciando un'eccezione oppure provvedendo allo shutdown del sistema.

Un utente può eseguire operazioni in kernel mode attraverso il meccanismo delle system call:
il programma invia un interruzione software al sistema operativo che, dopo aver salvato il suo stato, cederà il controllo al sistema operativo. il sistema operativo eseguirà in kernel mode l'operazione richiesta. Una volta terminata l'operazione, il controllo ritorna al programma chiamante (user mode).
2Inoltre ogni sistema suddivide la memoria in segmenti. Ogni segmento è associato a un livello di privilegio. Per esempio l'area di memoria del sistema operativo è di livello zero.

1.4 La struttra del sistema operativo

- Programma: entità passiva, insieme di byte che deve essere eseguito
- Processo: entità attiva, istanza del programma. Entita' dinamica che cambia nel tempo.

Ogni processo è identificato dal programma e dal contesto di esecuzione(PC, registri ecc. . .). Le componenti principali di un sistema operativo(faremo riferimento ai SO multiprogrammati timesharing):

- **Gestione dei processi**:
  1. Creazione/terminazione dei processi
  2. Sospensione/ripristino dei processi
  3. Sincronizzazione/comunicazione dei processi.
  4. Gestione del blocco critico (deadlock) di processi
- **Gestione della memoria centrale CPU(nel nostro corso è unica nel sistema)**:
  1. Separare gli spazi di indirizzi associati ai processi
  2. Allocare/deallocare memoria dei processi.
  3. Memoria virtuale: gestire spazi logici di indirizzi di dimensioni complessivamente superiori allo spazio fisco
  4. Realizzazione di collegamenti tra memoria logica e memoria fisica.(binding)
- **Gestione della memoria secondaria e del file system**:
  1. Allocazione/deallocazione di spazio
  2. Gestione dello spazio libero
  3. Scheduling delle operazioni su disco
  4. Fornire una visione logica uniforme della memoria secondaria:
     - Realizzare il concetto astratto di file, come unità di memorizzazione logica
     - Fornire una struttura astratta per l'organizzazione dei file (directory)
  5. Creazione/cancellazione file

6. Manipolazione dei file
7. Associazione tra file e dispositivi di memorizzazione secondaria

In generale file e directory vengono trattati in modo omogeneo alle periferiche di input output

- **Protezione e sicurezza**:

1. Controllo dell'accesso alle risorse da parte dei processi mediante: autorizzazioni e modalità di accesso.

- **Interfaccia utente**:

1. Interprete dei comandi shell linea di comando.
2. Interfaccia grafica (GUI) = interazione mouse - elementi grafici. Di solito è organizzata a finestre.

L'interfaccia del SO verso i processi e' rappresentato dalle system call. In generale definiamo un **programma di sistema** un programma che utilizza delle system call.

Le componenti sopra elencate possono essere organizzate in modi differenti all'interno del sistema operativo:

1. **Struttura monolitica**: il sistema operativo è costituito da un unico modulo contenente un insieme di procedure, che realizzano le varie componenti. Il principale vantaggio di un sistema monolitico è il basso costo di interazione tra le varie componenti, ma il SO è un sistema complesso e presenta gli stessi requisiti delle applicazioni in the large che non sono soddisfatte dalla struttura monolitica. Per aggiungere e modificare qualcosa bisogna ricompilare tutto. Programmi di questa dimensione portano con sé inoltre molti bug, quindi non si rispetterebbe l'affidabilità del sistema. L'interazione tra le diverse componenti avviene mediante il meccanismo di chiamata a procedura. Quello che il progettista cerca di raggiungere è di raggruppare i moduli che interagiscono di più tra loro.
2. **Struttura modulare**: le varie componenti del sistema operativo vengono organizzate in moduli caratterizzati da interfacce ben definite. Ogni strato ha un insieme di funzionalità che vengono offerte allo strato superiore mediante interfacce (system call). Ogni modulo è indipendente dagli altri. Se ne modifico uno, posso permettermi di ricompilare solo quello a patto di non modificare l'interfaccia.
   - Vantaggi:
     - *Astrazione*: ogni livello è un oggetto astratto, che fornisce ai livelli superiori una visione astratta del sistema.
     - *Modularita'*: possibilità di sviluppo, verifica, modifica in modo indipendente dagli altri livelli.
   - Svantaggi:
     - *Organizzazione gerarchica tra le componenti*, non sempre è possibile, difficoltà di realizzazione.
     - *Scarsa efficienza*: costo di attraversamento dei livelli.
3. **Microkernel**:Il sistema operativo è composto da una parte che esegue in modo kernel. Nei sistemi a microkernel la struttura del nucleo (kernel) è ridotta a poche funzionalità di base mentre il resto del sistema operativo è rappresentato da processi utente. Solitamente anche il sistema operativo è diviso in moduli. Non tutti i moduli devono eseguire in modalità Kernel. Diversa la situazione per i sistemi monolitici. Lì tutto il SO lavora in modalità kernel. E' un sistema affidabile(sono famosi per la loro sicurezza) e personalizzabile ma non molto efficiente a causa delle molte chiamate a system call. Il kernel è composto:
   - Creazione/terminazione dei processi
   - Scheduling della CPU
   - Gestire il cambio di contesti
   - Sincronizzazione/comunicazione tra processi

- Gestione della memoria
- Gestione dell'I/O
- Gestione delle interruzioni
- Implementazione system call.

4. **Kernel ibridi**: una via di mezzo tra i kernel monolitici e quelli a microkernel (XNU- X is not UNIX Mac OS X, Microsoft Windows)

1.5 Macchine virtuali

Dato un sistema caratterizzato da un insieme di risorse (hardware e software), virtualizzare il sistema significa presentare all'utilizzatore una visione delle risorse del sistema diversa da quella reale. Ciò si ottiene introducendo un livello di indirezione tra la vista logica e quella fisica delle risorse.

2 Processi

Un processo è definito come l'istanza di un programma. Allo stesso programma possono quindi essere associati più processi. Uno degli obiettivi principali del sistema operativo e' organizzare e memorizzare gli attributi dei processi.

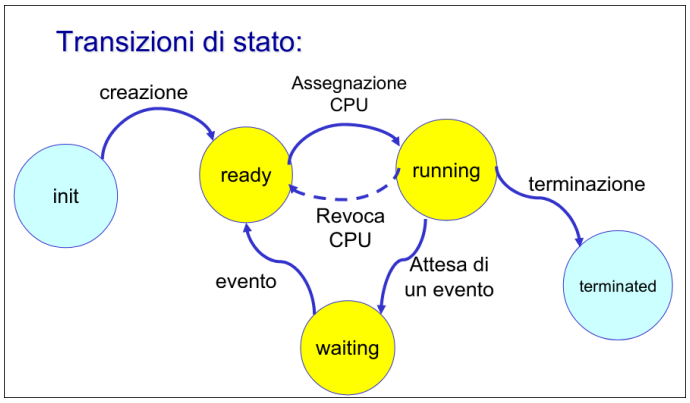
Quali sono gli attributi di un processo?

- Codice;
- Dati: variabili globali;
- Program counter;
- Altri registri;
- Stack: parametri, variabili locali e funzioni/procedure;
- Inoltre a un processo possono essere associate delle risorse (che nei SO UNIX-like sono rappresentati attraverso l'astrazione del file).

2.1 Stato di un processo

Un processo, durante il suo ciclo di vita, si può trovare in vari stati:

1. *init*: stato transitorio. Il processo viene caricato in memoria e il SO inizializza tutti i dati necessari. Un processo viene portato nella fase di init dallo scheduler dei processi.
2. *ready*: il processo pronto per acquisire la CPU.
3. *running*: il processo sta utilizzando la CPU
4. *waiting*: il processo è sospeso in attesa di un evento o di una risorsa.
5. *Terminated*: stato transitorio. È un tempo dovuto alla deallocazione del processo dalla memoria.



- Un processo che si trova nello stato running o stato ready è chiamato processo attivo.
- Un processo che si trova nello stato di waiting è chiamato processo sospeso.

La revoca (la quale non si verifica obbligatoriamente) e l'assegnazione della CPU è sempre effettuata dallo scheduler. Questa operazione di revoca si chiama preemption. Durante questo passaggio da ready a running (oppure waiting-ready-running), deve essere memorizzato il contesto di esecuzione. La struttura dati che che contiene le informazioni necessarie si chiama descrittore di processo (PCB - process control block).

2.1.1 PCB

Il **PCB** è una struttura dati che descrive (descrittore) un processo in qualunque suo stato. Ad ogni processo è associato solamente un descrittore, che lo rappresenta sia staticamente che dinamicamente. Il PCB è una struttura dati che a sua volta è inserito in strutture dati più ampie come code per l'accesso a una risorsa. Per lo stato di ready e waiting sono organizzate delle code in cui vengono inseriti i descrittori dei processi.

Cosa contiene il descrittore di un processo:

1. Stato del processo
2. Registri di CPU (IR, PC ecc..)
3. Informazioni di Scheduling (priorità, puntatori alle code, ecc.. questo varia da SO a SO ovviamente. Non tutti applicano le stesse politiche di scheduling)
4. Informazioni che servono al gestore della memoria (registri base, limite, ...)
5. informazioni relative all'I/O (risorse allocate, file aperti, . . .)
6. Informazioni di accounting (tempo di CPU utilizzato, . . .)

2.1.2 Scheduling

In generale, il sistema operativo compie tre diverse attività di scheduling:

- *Scheduling a lungo termine*: componente del sistema operativo che si occupa di selezionare i programmi presenti nella memoria secondaria da caricare nella memoria centrale (creando i corrispondenti processi)

  NB: Nei sistemi moderni general-purpose è praticamente assente e molto spesso è l'utente che stabilisce il grado di multiprogrammazione.

- *Scheduling a medio termine (Swapping)*: Si occupa di trasferire in memoria secondaria, temporaneamente, dei processi, o parti di essi, in modo da consentire il caricamento di altri processi. Essendo il grado di multiprogrammazione un numero finito, non sempre riusciamo a gestire tutti i processi (**il grado di multiprogrammazione** è il massimo numero di processi che un sistema operativo è in grado di gestire). Solitamente i processi target dello swap sono i processi in stato di waiting.

- *Scheduling a breve termine (CPU)*:E' quella parte del sistema operativo che si occupa della selezione dei processi a cui assegnare la CPU. Una volta selezionato il processo(che va quindi deciso), una parte del sistema operativo (*Dispatcher*) effettuerà un cambio di contesto (context switch). Deve essere molto efficiente, gira centinaia di volte in un secondo. Quando si verifica questo cambio di contesto, il sistema operativo deve salvare lo stato del processo uscente, aggiornando il suo PCB(i registri, risorse ecc...) e trasferire i dati dal PCB del processo entrante alla CPU. Di fatto lo scheduler ha due compiti: il primo è quello decisionale e l'altro operativo, cioè attuare il cambio di contesto.

  Il processo a cui viene tolta la cpu è detto deschedulato.

  Lo scheduling a breve termine gestisce due code:

  - La coda dei processi pronti
  - La coda dei processi in stato di waiting

  Scelte ottimali di scheduling dipendono dalle caratteristiche dei processi. Ad es.:

- processi I/O-bound: maggior parte del tempo in operazioni I/O
- processi CPU-bound: maggior parte del tempo in uso CPU

Overhead: Il passaggio da un processo al successivo può richiedere onerosi trasferimenti da/verso la memoria secondaria, per allocare/deallocare gli spazi di indirizzi dei processi. Troppi cambi di contesto aumentano l'overhead della CPU.

L'overhead dipende da una serie di parametri:

- La dimensione del PCB;
- La frequenza di cambio di contesto;
- Costo trasferimento da/verso la memoria;

Nel progetto dello scheduler, è fondamentale valutare questo genere di parametri. Questo ci porta all'utilizzo di processi leggeri, i THREAD, che hanno la proprietà di condividere codice e dati con altri processi. L'effetto è una diminuzione del PCB e una conseguente diminuzione dell'overhead.

2.2 Operazione sui processi

Ogni sistema operativo multiprogrammato prevede dei meccanismi per la gestione dei processi:

1. Creazione
2. Interazione tra processi
3. Terminazione

Queste sono delle operazioni privilegiate (esecuzione in modo kernel). Il programmatore può quindi operare sui processi soltanto attraverso delle opportune system call.

2.2.1 Creazione

Come si crea un processo? In generale esiste un processo padre che crea un processo figlio(fork). Si genera quindi una gerarchia di processi rappresentabile con una struttura ad albero, la radice è un processo creato dal bootloader chiamato "init"(per esempio in GNU/LINUX si chiama "systemd" con PID 1). UNIX e Windows usa questo tipo di struttura. Esempio: init - shell - nautilus - gedit - ecc...
Il sistema operativo tiene traccia delle relazioni tra i processi. Questo perché al padre potrebbe servire il risultato/i dell'esecuzione del proprio figlio. Quali sono gli aspetti caratteristici di una relazione padre-figlio:

1. *Concorrenza*:
   (a) Padre e figlio procedono in parallelo (es.UNIX).
   (b) Il padre attende la terminazione dei figli.
2. *Condivisione di risorse*:
   (a) Le risorse del padre sono condivise con i figli (es.UNIX).
   (b) Il figlio utilizza risorse solamente se le richiede esplicitamente.
3. *Spazio degli indirizzi*:
   (a) **Duplicato**:lo spazio degli indirizzi del figlio è una copia di quello del padre. Stesso codice, copia degli stessi dati ecc..(UNIX-like system seguono gli standard Posix)
   (b) **Differenziato**: spazio degli indirizzi di padre e figlio con codice e dati diversi

2.2.2 Terminazione

Come si comportano i figli di un padre che termina? E il padre di un figlio che termina? Nel primo caso esistono due possibilità: o tutti i figli terminano, oppure i figli continuano l'esecuzione ma devono cercarsi un nuovo processo padre. Nel secondo caso il padre può rilevare il suo stato di terminazione.

2.3 Thread

Un thread è un unità di esecuzione che può condividere codice e dati con altri thread associati al medesimo task.

> **Task**: insieme di thread che riferiscono lo stesso codice, gli stessi dati e le stesse risorse (il codice e i dati non sono caratteristiche del singolo thread, ma del task al quale appartengono). Thread dello stesso task modificano gli stessi dati!

A differenza dei processi pesanti, un thread può condividere variabili con altri thread aumentando il supporto alla comunicazione tra flussi di esecuzione. Questo porta ad una riduzione del costo del context switch, poiché i PCB dei thread non contengono alcuna informazione relativa al codice e dati. Questo perché risorse, codice e dati sono caratteristiche del task, non del thread!. Ci sarà quindi un PCB per ogni task, a cui aggiungo i PCB, più piccoli, dei thread che fanno riferimento a quel task. Il PCB del thread diventa quindi molto leggero: contiene solo i registri e le informazioni che sono necessarie per l'esecuzione! Un processo single-thread è chiamato anche processo pesante. I thread condividono il codice e possono condividere dati e risorse(files).

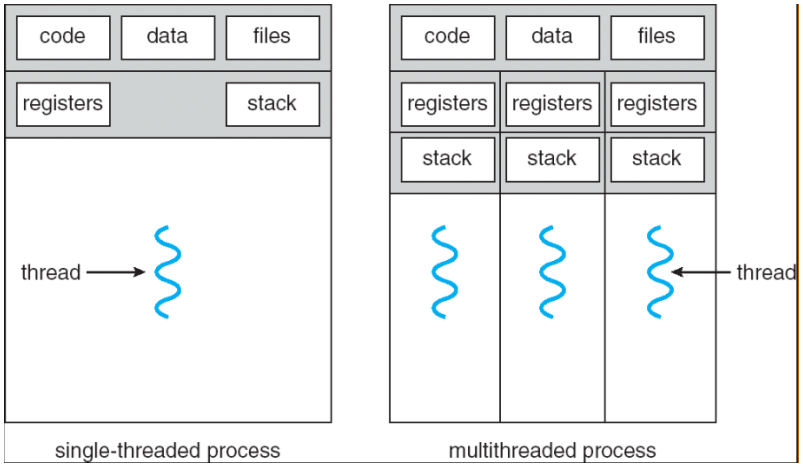


Figure 1: Da questa immagine si puo' notare la diminuzione del PCB per i sistemi multithread.

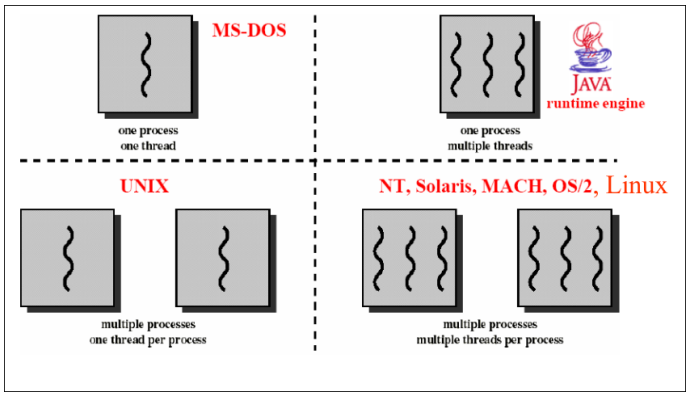


Figure 2: Thread e SO.

Lo svantaggio dell'utilizzo dei thread è la sicurezza e la coerenza dei dati, in quanto un thread ha accesso ai dati di un altro thread.

2.4 Processi indipendenti e interagenti

- Processi interagenti: due processi sono interagenti se l'esecuzione di uno è influenzata dall'esecuzione di un altro e/o viceversa. Esistono diversi tipi di interazione:

1. **cooperazione**:Interazione prevedibile e desiderata, i processi collaborano per il raggiungimento di un fine comune.
2. **Competizione**: Interazione prevedibile ma "non desiderata".
3. **Interferenza**:Interazione non prevista e non desiderata.

L'interazione può avvenire mediante:

1. **Memoria condivisa**(Ambiente globale): Il sistema operativo consente ai processi di condividere variabili: l'interazione avviene tramite l'accesso a variabili condivise.
2. **Scambio di messaggi**(Ambiente locale): I processi non condividono variabili e interagiscono mediante trasmissione/ricezione di messaggi.

Il vantaggio principale di utilizzare processi che possono interagire fra di loro (*condivisione di informazioni*) è la velocità di esecuzione, in quanto vi è una suddivisione dei compiti tra i vari processi (*Modularità*).

- Processi indipendenti: due processi si dicono indipendenti se l'esecuzione di uno non è influenzata dall'altro e viceversa.

2.5 Implementazione dei thread

La creazione de thread può essere realizzata nei seguenti modi:

– **Livello kernel**: Il SO gestisce direttamente i cambi di contesto e SO fornisce strumenti per la sincronizzazione per l'accesso di thread a variabili comuni.

– **A livello utente**: es(POSIX, Java thread) Viene creata un'astrazione tra i thread e il kernel. Questo *gestore di thread* si interpone tra l'utente e il kernel così da effettuare il passaggio da un thread al successivo (nello stesso task) senza la necessità di utilizzare system call. Però, nel fare questo, il SO vede solo processi pesanti: la sospensione di un thread causa la sospensione di tutti i thread del task.

3 UNIX

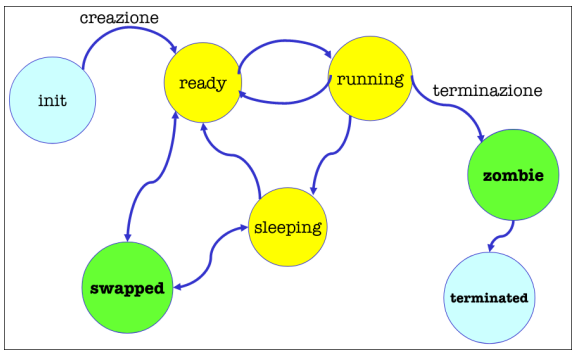
3.1 Caratteristiche di un processo UNIX

Ogni processo ha un proprio spazio di indirizzamento completamente locale e non condiviso. Questo modello è detto **modello ad ambiente locale** (o a **scambio di messaggi**). I processi UNIX NON condividono dati. Il codice però è condiviso. Se ci sono più processi che eseguono lo stesso programma, allora alloco il codice in memoria solo una volta(**codice rientrante**)

3.1.1 Stati di un processo UNIX

Oltre agli stati generali (init, ready, running, sleeping, terminated) si hanno in aggiunta due nuovi:

- **Zombie**: il processo è terminato, ma è in attesa che il padre analizzi lo stato di terminazione.
- **Swapped**: l processo (o parte di esso) è temporaneamente trasferito in memoria secondaria dallo scheduler a medio termine che effettua uno *swap-out* (si applica preferibilmente ai processi più grandi o a quelli in stato di wait). Lo si fa nel caso in cui la memoria primaria è piena, per questo motivo nei sistemi moderni non interviene praticamente mai. Lo *swap-in* privilegerà i processi più corti invece.



3.1.2 Rappresentazione processi UNIX

Il codice dei processi è rientrante, quindi più processi possono condividere lo stesso codice. Per permettere questo il codice ed i dati devono essere separati (**modello a codice puro**). Il sistema operativo gestisce una struttura dati globale in cui sono contenuti i puntatori ai codici utilizzati, eventualmente condivisi, dai processi.
La struttura che contiene i puntatori ai codici dei programmi viene chiamata: **text table**. L'elemento della text table si chiama **text structure** (una text structure per ogni codice attualmente in uso) e contiene: un puntatore al codice ed il numero dei processi che lo condividono: un semplice contatore, utile per capire se è possibile eliminare la text structure nel caso vada a zero.

Text Table	
Code Pointer	Process Counter
0020 0000h	3
00A0 4000h	1
00C0 FFFFh	2
02C0 FFC0h	3
...	...
0020 0000h	1
00A0 4000h	1
20C3 FFEFh	3
E2C0 11C1h	1

A differenza di quanto abbiamo visto prima, il PCB è rappresentato da due strutture dati (invece di una):

1. **Process structure**: contiene le informazioni necessarie al sistema per la gestione del processo (PID, stato, puntatori alle aree dati e stack del processo, *riferimento all'elemento della*

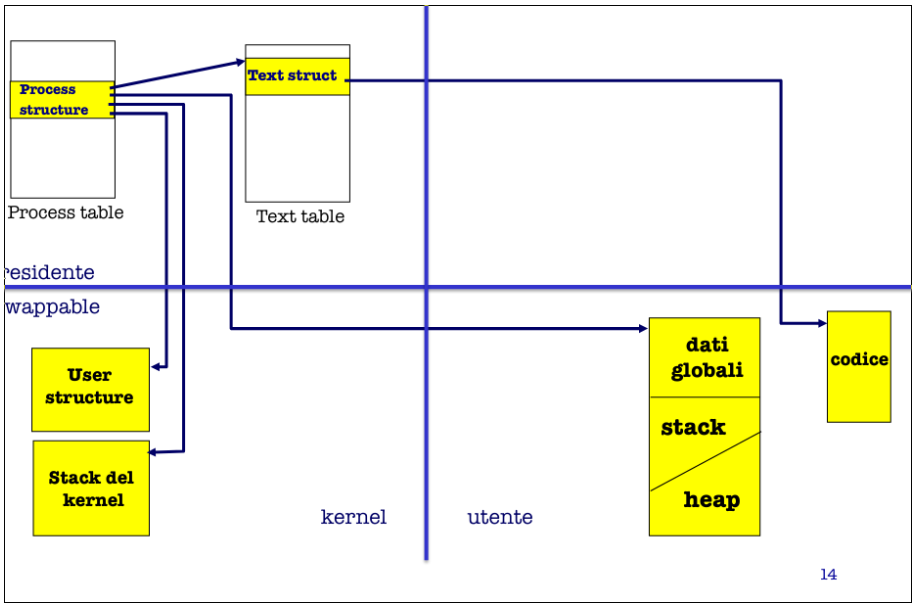
text table associato al codice del processo, informazioni di scheduling, riferimento al PID padre, gestione dei segnali, puntatore al processo successivo nella coda dei processi in cui si trova, *puntatore alla user structure*). Le process structure sono organizzate in un vettore: **process table** (1 elemento per ogni processo).

Process Table						
PID	Stato	Text table	Scheduling	PID padre	...	User structure
1020	waiting	2034	4	1	..	1234
1	waiting	2034	4	1	..	2345
2323	zombie	2034	3	1020	..	2342
1450	swapped	2034	2	1	..	0234
...	...	...	...	...	...	...
0100	running	2034	4	1	..	2134

2. **User structure**: informazioni necessarie solo se il processo è residente in memoria centrale (inutile se è stata swappata) (registri CPU, informazioni sulle risorse allocate (cioè file aperti), informazioni sulla gestione dei segnali, ambiente del processo(directory corrente, utente, gruppo, argc/argv, path)...).

3.1.3 Immagine di un processo

Immagine di un processo: E' l'insieme di aree di memoria e strutture dati, associate al processo. *ATTENZIONE*: non tutta l'immagine è accessibile in modo user.



L'immagine riassume le parti che possono essere accedute in modalità kernel o user, e quelle che possono essere swappate e quelle che non lo possono essere. N.B. Lo stack del kernel è un'area di memoria utilizzata per il passaggio dei parametri tramite system call.
Ogni processo può essere soggetto a swapping: ma non tutta l'immagine di un processo può essere trasferita in memoria: *una parte swappable ed una parte residente non swappable.*

4 System Call

4.1 Fork()

int fork(void);

Questa funzione consente a un processo di generare un processo figlio. Padre e figlio condividono lo stesso codice (per la rientranza del codice), quindi eseguono lo stesso programma. Il figlio eredita una copia di *tutti* i dati del padre (tranne la process structure ovviamente).

La funzione non richiede parametri e restituisce un intero che dipende dal processo:

- 0 per il processo figlio creato.
- maggiore di 0, indicante il PID del figlio, per il processo padre.
- minore di 0 se la creazione non è andata a buon fine.

L'utilizzo della fork() comporta l'allocazione di una nuova process structure associata al processo figlio e l'allocazione di una nuova user structure nella quale viene copiata la user structure del padre. Infine dovrà essere incrementato il contatore della text structure corrispondente. Dopo aver utilizzato la fork:

- Padre e figlio procedono in parallelo(concorrenza).
- Lo spazio degli indirizzi è duplicato. Ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della fork()
- Se la user structure è duplicata, le risorse allocate al padre sono condivise con i figli, le informazioni per la gestione dei segnali sono le stesse e infine il figlio nasce con lo stesso PC del padre.

esempio di fork:

```
1  #include <stdio.h>
2  if(fork()==0){
3      //codice figlio
4  }else{
5      //codice padre che continua la sua esecuzione concorrente
6  }
```

4.2 Exit()

void exit(int status);

Esistono due tipi di terminazione: volontaria e involontaria.

- *Volontaria*: o tramite exit() o per esecuzione dell'ultima istruzione.
- *Involontaria*:azioni illegali, interruzione mediante segnale. Nel caso di terminazione involontaria si salva l'immagine del processo nel "core" per analizzare a posteriori le cause dell'interruzione.

La funzione prevede un parametro status mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione. L'utilizzo della exit() comporta la chiusura dei file aperti non condivisi e la terminazione del processo:

- Se il processo che termina ha figli in esecuzione, il processo init "adotterà" tutti i figli ancora in esecuzione. Nella process structure dei figli, il riferimento al padre deve essere modificato con il valore 1.
- Se il processo termina prima che il padre ne riveli lo stato di terminazione con la system call wait(), il processo passa nello stato zombie. Non è detto che il padre ne verifichi lo stato, nel caso questo termini i figli in stato zombie termineranno con lui.

N.B. Quando termina un processo adottato dal processo init, init rileva automaticamente il suo stato di terminazione. Perciò i processi figli di init non permangono nello stato di zombie.

4.3 Wait()

int wait(int* status);

Questa funzione prevede un parametro "status" che rappresenta l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio. Quindi la wait serve al padre per verificare il risultato di un processo in stato zombie.

Il risultato prodotto dalla wait() è il PID del processo terminato oppure un codice di errore (minore 0). La logica è la seguente:

1. Se tutti i figli non sono ancora terminati, il processo si sospende (stato sleepy) in attesa della terminazione del primo di esso. Si dice in questo caso che la wait si dice sospensiva.
2. Se esiste almeno un figlio in stato zombie, la wait() restituisce immediatamente il suo stato di terminazione.
3. Se non esiste neanche un figlio (la wait() non è *sospensiva*) ritorna un codice di errore.

In caso di terminazione di un figlio, la variabile status raccoglie lo stato di terminazione. Si distinguono due casi:

1. Byte meno significativo di status è zero: si parla di terminazione volontaria primi 8 bit codice di stato e ultimi 8 bit pari a zero.
2. In caso contrario il byte meno significativo descrive il segnale che ha terminato il figlio (terminazione involontaria) primi 8 bit codice di stato saranno inutili e ultimi 8 bit indicano il segnale che ha terminato il figlio.

Esempio di wait, N.B. lo standard POSIX.1 prevede delle macro definite nell'header file ¡sys/wait.h¿ per l'analisi dello stato di terminazione:

```
1  #include <sys/wait.h>
2  int main(){
3      int pid, status;
4      pid=fork();
5      if (pid==0){
6          printf("sono il figlio\n");
7          exit(0);
8      }
9      else {
10         pid=wait(&status);
11         if (WIFEXITED(status))
12             printf("Terminazione volontaria di %d con
13                 stato %d\n", pid, WEXITSTATUS(status));
14         else if (WIFSIGNALED(status))
15             printf("terminazione involontaria per segnale
16                 %d\n", WTERMSIG(status));
17     }
18 }
```

Lo standard POSIX.1 prevede delle macro definite nel file ¡sys/wait.h¿:

- **WIFEXITED(status)**: restituisce vero se il processo figlio è terminato volontariamente. In questo caso la macro **WEXITSTATUS(status)** restituisce lo stato di terminazione
- **WIFSIGNALED(status)**: restituisce vero se il processo figlio è terminato involontariamente. In questo caso la macro **WTERMSIG(status)** restituisce il numero del segnale che ha causato la terminazione

4.4 Exec()

execl(), execle(), execlp, execv(), execve(), execvp()

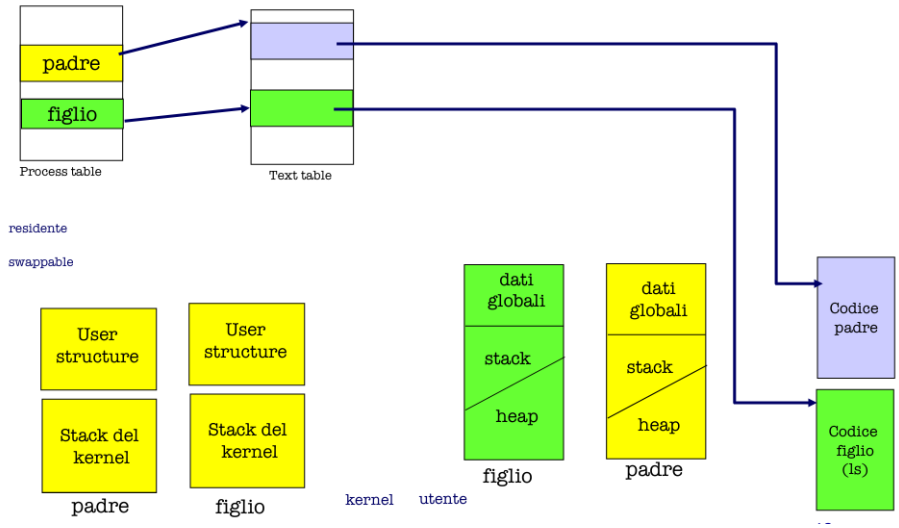
Questa system call permette di differenziare il codice di due processi. In altre parole, vengono sostituiti il codice e gli argomenti di invocazione (es argc e argv) del processo chiamante, con il

codice e gli argomenti di un programma specificato come parametro della funzione. Restituisce un intero solamente se si è verificato un errore.

N.B. Non genera nuovi processi, non confondere fork ed exec.

L'utilizzo di questa funzione comporta il cambiamento dei seguenti attributi:

1. Codice, dati globali, stack e heap.
2. riferimento alla text table
3. Mantiene la stessa process structure e stack del kernel. Invece nella user structure non cambia nulla che non si riferisca al codice (quindi cambia il PC ecc). **Per esempio gli eventuali file aperti, rimangono aperti!!**
4. l'environment cambia solamente se eseguita execle o exexve.



Le variabili di exec, a seconda del suffisso:

- **l**: gli argomenti da passare al programma da caricare vengono specificati mediante una lista di parametri. *es. execl().*
- **p**: il nome del file eseguibile specificato come argomento della system call viene ricercato nel PATH contenuto nell'ambiente del processo. *es. execlp().*
- **v**: gli argomenti da passare al programma da caricare vengono specificati mediante un vettore di parametri. *es. execv()*
- **e**:la system call riceve anche un vettore (envp[]) che rimpiazza l'environment (path, direttorio corrente, ...) del processo chiamante. *es. execle()*

int execl(char* pathname, char* arg0, ... , char argN, (char*)0);

1. **Pathname**: nome del file eseguibile, già compilato.
2. **Arg0**: nome del programma.
3. **Arg1, ... , ArgN**: eventuali parametri.
4. **(char*)0**: delimita la fine dei parametri, è un puntatore nullo

Es. execl("/bin/ls","ls", "-l", "miao", (char*)0);
N.B. una exec non restituisce nulla al processo che la esegue perché il codice viene cambiato. Il valore di ritorno ha senso solamente in caso di errore. Per leggere l'errore si usa l'altra primitiva "perror"(vedi sotto).

int execve(char *pathname, char *argV[], char * env[]);

1. **Pathname**: nome del file eseguibile, già compilato.
2. **ArgV[ ]**: è il vettore degli argomenti del programma da eseguire. *es: char *argv[]="ls", "-l", (char *)0;*
3. **Env[ ]**: è il vettore delle variabili di ambiente da sostituire all'ambiente del processo. *esempio: char *env[ ]="USER=edoardo", "PATH=/home/edoardo/provaExec/", (char*)0 ;*

4.5 Perror()

Per convenzione le system call restituiscono il valore -1 in caso di errore. Inoltre nei sistemi UNIX ogni system call fallita setta la variabile globale **errno**. In questa variabile sarà quindi salvato il codice d'errore dell'ultima system call eseguita. La corrispondenza tra codice di errore e descrizione è definita in ¡sys/errno.h¿
La system call *perror(stringa)* permette di stampare la stringa passata, in aggiunta alla descrizione dell'errore dell'ultima system call eseguita.

```
1  int main(){
2      int pid, status;
3      pid=fork();
4      if(pid<0){
5          perror("Fork fallita:");
6          exit()
7      }
8      else{
9          ...;
10     }
11 }
```

4.6 Include necessari

```
1      #include <stdlib.h> //exit
2      #include <sys/wait.h> //wait
3      #include <unistd.h> //fork, exec
4      ...
```

5 Scheduling

L'obiettivo principale della multiprogrammazione è la massimizzazione dell'utilizzo della CPU. Lo **scheduler** della CPU: è quella parte del sistema operativo che secondo determinate politiche assegna ai processi in stato di ready, il controllo della CPU. Si distinguono quindi due componenti principali in questa attività: lo scheduler decide a quale processo assegnare la CPU e a seguito della decisione viene attuato il cambio di contesto. Il **dispatcher** è la parte di SO che realizza il cambio di contesto.

N.B. quando parliamo di scheduler parliamo di scheduler a breve termine, in quanto nei sistemi moderni lo scheduler a lungo termine è attuato dall'utente. Quello a medio termine invece è lo swapper.

È importante differenziare le parti di esecuzione di un processo:

- **I/O bound**: prevalenza di attività di I/0.
- **CPU bound**: prevalenza di utilizzo della CPU.

Gli algoritmi di scheduling si possono classificare in due categorie:

- **Non pre-emptive**: la CPU rimane allocata al processo running finché esso non si sospende volontariamente o non termina.
- **Pre-emptive**: un processo running può essere prelazionato, cioè lo scheduler può sottrargli la CPU per assegnarla ad un nuovo processo.

  Ovviamente i sistemi a divisione di tempo hanno sempre uno scheduling pre-emptive.

5.1 Criteri di scheduling

: Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni indicatori di performance:

1. **Utilizzo della CPU**: percentuale media di utilizzo CPU nell'unità di tempo.
2. **Throughput** (del sistema): numero di processi completati nell'unità di tempo.
3. **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue.
4. **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento.
5. **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround non dipende dalla velocità di risposta delle periferiche I/O).

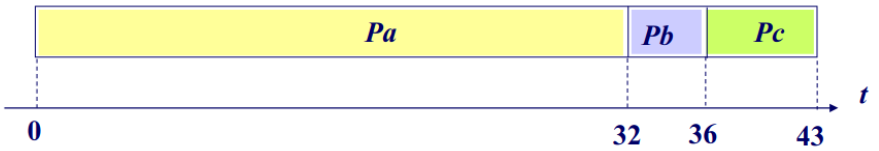
In generale, devono essere massimizzati "utilizzo della CPU" e "throughput" e minimizzati i restanti. Non è possibile però, ottimizzare tutti i criteri contemporaneamente e quindi, a seconda del sistema operativo, le politiche di scheduling possono avere diversi obiettivi.

Per esempio:

- *Nei sistemi batch*: massimizzare throughput e minimizzare turnaround
- *Nei sistemi interattivi(UNIX per esempio)*: minimizzare il tempo medio di risposta dei processi e minimizzare il tempo di attesa.

5.2 Politiche di scheduling

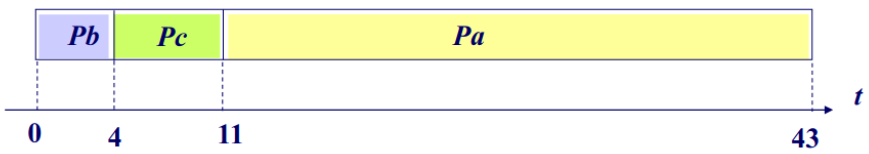
- **FCFS - first come first served**: la coda dei processi pronti è gestita in modalità FIFO. E' un algoritmo non pre-emptive: non è possibile influire sull'ordine dei processi.



Pa è il tipico esempio di processo CPU bound e Pb e Pc possono essere I/O bound.

Uno dei problema di questa politica è che un processo con **CPU burst**(tempo di utilizzo della CPU) molto alto, impedisce agli altri processi di utilizzare la risorsa. In questo caso si parla del cosiddetto **effetto convoglio**, caratterizzato dal fatto che più processi brevi attendono la terminazione di un unico processo più corposo.

- **SJF - shortest job first**: per ogni processo nella ready queue (coda dei processi pronti) viene stimata la lunghezza del CPU burst e viene schedulato il processo con il CPU burst più corto. Questo algoritmo ottimizza il tempo di attesa e può essere: non pre-emptive e pre-emptive. In questo ultimo caso se nella coda arriva un processo con CPU burst minore del CPU burst rimasto al processo in stato di running).



- **RR - Round Robin**: Tipicamente utilizzata nei sistemi time sharing: la coda dei processi è gestita come una coda FIFO circolare. Ad ogni processo viene allocata la CPU per un intervallo di tempo costante (time slice), scaduto questo tempo il processo viene re-inserito in coda e la CPU passa ad un altro processo. La RR può essere vista come un estensione del FCFS con pre-emption periodica.

Questa politica è detta **fair**, cioè tutti i processi vengono trattati in modo equo. Questo però, può generare un problema, siccome non tutti i processi sono uguali. Il Sistema operativo per esempio non può avere lo stesso trattamento di un processo user.

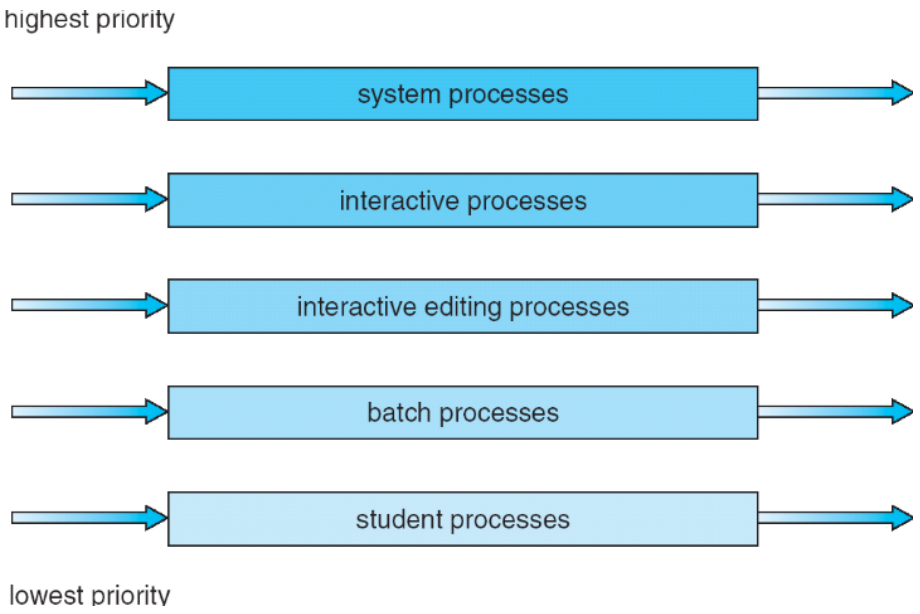
L'efficienza di questa politica di gestione varia in base al time slice:

- *time slice piccolo*: tanti context switch: overhead si alza.
- *time slice alto*: overhead basso ma tempi di risposta più alti.

- **Con priorità**: ad ogni processo vine assegnata una priorità, che può essere:
  - Definita internamente: il sistema operativo attribuisce ad ogni processo una priorità in base a politiche interne.
  - Esternamente: criteri esterni al sistema operativo, stabiliti per esempio dall'utente.(es syscall **nice** in UNIX)

  Lo scheduler seleziona il processo con priorità più alta e nel caso di priorità uguale si attua la politica FCFS. Le priorità possono essere costanti o variare dinamicamente. In generale ogni politica basata su certe caratteristiche dei processi (come priorità, SJF ecc...)presenta un problema: la **starvation** dei processi. In altre parole, la starvation si verifica quando uno o più processi di priorità bassa vengono lasciati indefinitamente nella coda dei processi pronti, perché vi è sempre un processo pronto di priorità più alta. La soluzione a questo problema è la modifica dinamica delle priorità, come ad esempio, la priorità decresce al crescere del tempo di CPU giù utilizzato (tecnica di **aging, feedback negativo**), mentre cresce dinamicamente con il tempo di attesa del processo (tecnica di **promotion, feedback positivo**).

Nei sistemi operativi reali, spesso si combinano diversi algoritmi di scheduling. Un esempio può essere il **multiple level feedback queues - MLFQ**. Questa politica prevede più code associate a diversi livelli di priorità. Ogni coda può avere la propria politica di gestione, RR o FCFS. I processi possono spostarsi tra livelli attraverso i meccanismi di aging e i meccanismi di promotion. Di base lo scheduler parte dalla coda con priorità più alta e sceglie il primo processo che trova, saltando di fatto le code vuote.



- **Politica di scheduling di UNIX**: privilegiare i processi interattivi (cioè i processi con un basso CPU burst), aggiornamento dinamico delle priorità. L'utente può soltanto diminuire la priorità di un processo: comando nice. Alcuni scheduler utilizzano come architettura MLFQ: sono presenti 160 livelli di priorità, più è alta la priorità di un processo, più basso è il suo livello. Esiste un livello di riferimento *pzero* che funge da spartiacque tra i processi utente e quelli di sistema.
- **Scheduling dei thread Java**: la JVM usa una politica di scheduling pre-emptive e basata su priorità.

5.2.1 Stima del CPU burst

In generale è molto difficile stimare il CPU burst di un processo. Quello che si fa è stimare il CPU burst in base alla storia. Un processo, statisticamente, si comporterà come nella sua storia più recente.
Exponential averaging:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- t_n : tempo utilizzato dal processo la volta precedente.
- τ_n: tempo stimato per il processo la volta precedente.
- τ_{n+1}: tempo stimato per la prossima volta.
- α: numero compreso tra 0 e 1.

Ricordati che τ_n può essere a sua volta scritto come questa formula. Si può notare che più α è vicino allo zero meno la stima dipenderà dai valori precedenti del reale valore utilizzato dal processo la volta precedente. Mentre più α tenderà a 1, più dipenderà dalla storia precedente alla stima.

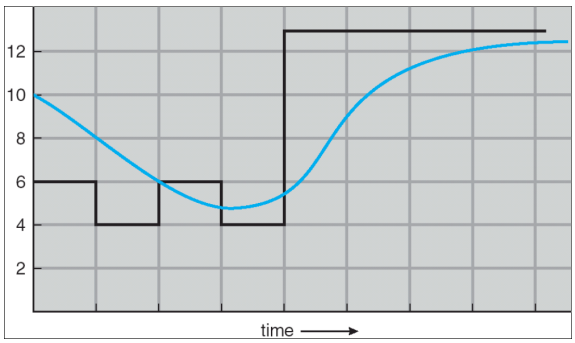


Figure 3: esempio con $\alpha = 0.5$

6 Comunicazioni tra processi

Processi interagenti: due processi sono interagenti se l'esecuzione di uno è influenzato dall'esecuzione di un altro e/o viceversa.
Esistono diversi tipi di interazione:

- **Cooperazione**: interazione prevedibile e desiderata, i processi collaborano per il raggiungimento di un fine comune. Fanno parte della stessa applicazione solitamente.
- **Competizione**: interazione prevedibile ma "non desiderata".Solitamente processi di applicazioni diverse che accedono alla stessa risorsa.
- **Interferenza**: interazione non prevista e non desiderata.

L'interazione può avvenire mediante due meccanismi:

- **Comunicazione**: scambio di informazioni tra i processi interagenti.
- **Sincronizzazione**: imposizione di vincoli temporali sull'esecuzione dei processi.

Modelli di interazione:

- **Memoria ad ambiente locale (processi pesanti)**: I processi non condividono memoria e interagiscono mediante trasmissione/ricezione di messaggi e la sincronizzazione avviene attraverso lo scambio di eventi come i segnali. Un'esempio è UNIX.

  N.B. in questo modello i meccanismi di comunicazione/sincronizzazione vengono realizzati dal sistema operativo.

- **Memoria ad ambiente globale (thread))**: Il sistema operativo consente ai processi di condividere memoria: l'interazione avviene tramite l'accesso a variabili condivise e opportuni strumenti di sincronizzazione come semafori e lock.

  N.B. in questo modello solo i meccanismi di sincronizzazione vengono realizzati dal sistema operativo. (la comunicazione viene realizzata dal programmatore)

Il sistema operativo offre dei meccanismi a supporto della comunicazione tra i processi, quali, **send** (spedizione di messaggi) e **receive** (ricezione di messaggi). Il **naming** è il modo con cui viene identificato il destinatario della comunicazione. La comunicazione avviene attraverso un **canale di comunicazione** ,astrazione creata dal sistema operativo per permettere a due processi di comunicare. Ogni canale di comunicazione ha determinata **capacità**, che indica il massimo numero di messaggi che è in grado di contenere contemporaneamente.
La comunicazione ha varie caratteristiche:

- **Tipo della comunicazione**: diretta o indiretta, simmetrica o asimmetrica, bufferizzata o no ...
- **Caratteristiche del canale**: monodirezionale o bidirezionale, uno-uno o molti-uno ecc, capacità, modalità di creazione: automatica o non automatica.
- **Caratteristiche del messaggio**:dimensione, tipo.

Esistono due modalità di specifica del destinatario:

1. **Comunicazione diretta**: al messaggio viene associato il pid del processo destinatario. I due processi devono conoscersi reciprocamente così da creare automaticamente il canale di comunicazione. Questo tipo di approccio presenta una scarsa modularità, in quanto, la modifica di un processo implica la revisione di tutte le operazioni di comunicazione (difficoltà di riutilizzo). Un esempio potrebbe essere "send(pid, msg)".

   Il canale viene creato automaticamente ed è punto-punto, univoco per la coppia (P1,P2) e bidirezionale.

2. **comunicazione indiretta**: il messaggio viene indirizzato ad una mailbox dalla quale il destinatario potrà prelevarlo. I processi non sono tenuti a conoscersi in quanto i messaggi verranno depositati/prelevati direttamente dalla mailbox, che fungerà da canale di comunicazione. **Una mailbox** è una risorsa astratta condivisibile da più processi che funge da contenitore messaggi.

Attraverso la mailbox, è possibile associare più canali di comunicazione alla stessa coppia mittente destinatario. A differenza della comunicazione diretta, questo canale di comunicazione deve essere creato in modo esplicito, non essendo di fatto automatico.

Il vantaggio di utilizzare questo metodo è il poter associare il canale a più di due processi, creando potenzialmente una comunicazione molti a molti. In questo caso la mailbox viene detta di *sistema*. Nel caso di comunicazione asimmetrica invece, la mailbox si definisce *porta* (terminologia presa dal modello client-server).

Comunicazione asimmetrica: avendo la possibilità di non far conoscere al destinatario il pid del mittente per ricevere il messaggio(come il client server), è possibile stabilire comunicazioni asimmetriche molti a uno.

6.1 Buffering del canale

La capacità del canale di comunicazione indica la lunghezza della coda, gestita secondo la politica FIFO, in cui vengono inseriti i messaggi spediti dal/i mittente/i. Di fatto il canale di comunicazione è gestito come un buffer.

Se la capacità del canale è nulla, allora significa che il canale non può bufferizzare alcun messaggio. In questo caso si dice che la comunicazione è sospensiva: affinché la comunicazione avvenga con successo è necessario sincronizzare i due processi, sospendendo il processo mittente/destinatario. Parliamo di comunicazione **sincrona**.

Figure 4: In questo caso il processo mittente viene sospeso

Ovviamente vale il contrario: se arriva prima la receive, il processo destinatario si sospende.

Nel caso in cui la capacità sia diversa da zero, si distinguono tre casi:

- Buffer non pieno: un nuovo messaggio viene posto in fondo alla coda.
- Buffer pieno: la send è sospensiva
- Buffer vuoto: la receive può(esistono delle implementazioni che avvisano che il buffer è vuoto) essere sospensiva.
- Buffer illimitato: la send non è sospensiva.

In caso la capacità sia maggiore di zero, la semantica della comunicazione è diverso: si dice che la send è **asincrona**.

6.1.1 Send con sincronizzazione estesa

È un tipo di comunicazione sincrona, in cui il mittente si sospende fino a che il destinatario non restituisce una risposta (**reply**) al messaggio inviato. Il mittente potrebbe richiedere una **Remote Procedure Call- RPC** per esempio.

Questa tipologia ricorda molto l'architettura client-server.

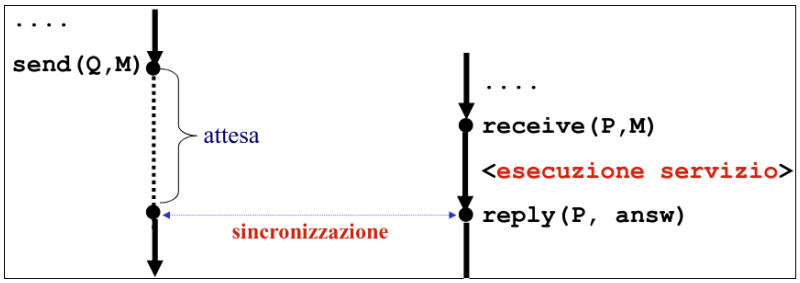


Figure 5: In questo caso il processo mittente viene sospeso fino al reply

6.2 Comunicazione in ambiente UNIX

La comunicazione in ambiente UNIX è sfrutta il modello a *memoria locale*. Esistono tre meccanismi di comunicazione:

1. **Pipe**:Comunicazione locale nell'ambito della stessa gerarchia di processi.
2. **Fifo**: Comunicazione locale tra processi di gerarchie diverse.
3. **Socket**: comunicazione in ambiente distribuito, tra processi in esecuzione su nodi diversi di una rete.

In questo corso ci concentreremo principalmente sulla pipe, perché è un canale di comunicazione standard nei sistemi UNIX (la fifo non è troppo standard). La pipe è un canale di comunicazione:

- *Indiretta (senza naming esplicito).*
- *Canale unidirezionale molti-a-molti.*
- *Bufferizzata con capacità limitata.(buffsize) Quindi la comunicazione è asincrona in generale e non sospensiva.*

6.3 Sincronizzazione in ambiente UNIX

La sincronizzazione permette di imporre vincoli sulle operazioni dei processi interagenti. Ovviamente nel modello a memoria globale sarà necessaria una sincronizzazione con maggiore complessità, in quanto in quanto deve essere garantita la mutua esclusione dei processi nell'utilizzo delle risorse. In ambiente UNIX gli accessi alle risorse "condivise" vengono controllati e coordinati dal sistema operativo. La sincronizzazione avviene mediante meccanismi offerti dal sistema operativo che consentono la notifica.

6.4 Segnali

Un segnale notifica un evento in modo asincrono, proprio come un'interruzione. Quando un qualunque processo riceve un segnale, il processo si interrompe e gestisce l'evento.
In generale un segnale può essere generato dal kernel verso dei processi utente e da processi utente verso processi utente.

Possiamo suddividere i mittenti dei segnali nel seguente modo:

1. Generati da terminale.
2. Generati dal kernel per **eccezioni HW** (come per esempio violazione dei limiti Di accesso in memoria)
3. Generati dal kernel per **interruzioni software**

4. Generati da altri processi.

Quando un processo riceve un segnale può comportarsi in tre modi diversi:

1. *Gestire* il segnale eseguendo una funzione handler definita dal programmatore.
2. *Gestire* il segnale eseguendo un'azione predefinita dal S.O. (**azione di default**).
3. *Ignorare* il segnale (nessuna reazione).

La gestione del segnale è asincrona e analoga alla gestione di un interrupt da parte della CPU. Interruzione asincrona, gestione dell'evento e infine ritorno all'esecuzione del codice.

In qualunque sistema UNIX esistono vari tipi di segnale associati ad un particolare evento, ognuno identificato con un intero che prevede una specifica azione di default. In Linux solitamente sono presenti 32 segnali. L'intero identificativo di un segnale è rappresentato da un nome simbolico definite in **¡signal.h¿**. Questo serve per garantire la portabilità tra i vari sistemi: il numero del segnale può cambiare, mentre il nome resta invariato.

Un possibile esempio di porzione di signal.h:

```
#define SIGHUP 1 /* Hangup (POSIX). Action: exit */
#define SIGINT 2 /* Interrupt (ANSI). Action: exit (^C)*/
#define SIGQUIT 3 /* Quit (POSIX). Action: exit, core dump*/
#define SIGILL 4 /* Illegal instr.(ANSI).Action: exit,core dump */
...
#define SIGKILL 9 /* Kill, unblockable (POSIX). Action: exit*/
#define SIGUSR1 10 /* User-defined signal 1 (POSIX). Action: exit*/
#define SIGSEGV 11 /* Segm. violation (ANSI). Act: exit,core dump */
#define SIGUSR2 12 /* User-defined signal 2 (POSIX).Act: exit */
#define SIGPIPE 13 /* Broken pipe (POSIX).Act: exit */
#define SIGALRM 14 /* Alarm clock (POSIX). Act: exit */
#define SIGTERM 15 /* Termination (ANSI). Act:exit*/
...
#define SIGCHLD 17 /* Child status changed (POSIX).Act: ignore */
#define SIGCONT 18 /* Continue (POSIX).Act. ignore */
#define SIGSTOP 19 /* Stop, unblockable (POSIX). Act: stop */
```

SIGUSR1 e **SIGUSR2** sono segnali "liberi", cioè è possibile sovrascrivere l'azione di default del SO, con una propria implementazione atta alla *sincronizzazione* dei propri processi.
SIGKILL e **SIGSTOP** sono segnali **unblockable** e non è possibile ignorarli.

6.5 Gestione dei segnali

void (* signal(int sig, void (*func)()))(int);

In generale, si imposta nelle prime istruzioni del programma la gestione dei segnali. La funzione richiede l'intero identificativo del segnale e il puntatore alla funzione che gestisce l'interruzione, la quale deve essere del tipo *void gestore(int);*.L'handler prevede un parametro formale di tipo int che rappresenta il numero del segnale effettivamente ricevuto. Questo è utile per differenziare le gestioni utilizzando un unico handler.
La signal restituisce un puntatore al precedente gestore del segnale. In caso non vada a buon fine restituisce **SIG_ERR**. *The void pointer in C (void*) is a pointer which is not associated with any data types. It points to some data location in the storage means points to the address of variables. It is also called general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers*

Nel caso io voglia ignorare l'interruzione basta specificare il valore **SIG_IGN** al posto della funzione. Nel caso volessi ripristinare la gestione dell'evento con l'azione di default basta specificare **SIF_DFL**.

```
#include <signal.h>
void gestore(int);
...
int main()
{...
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
```

```
8     signal(SIGUSR1, SIG_DFL); /*SIGUSR1 torna a default */
9     signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non e
10    ignorabile */
11    ...
12    }
```

6.5.1 SIGCHLD

SIGCHLD è il segnale che il kernel invia a un processo padre quando il figlio termina, è possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione handler per la gestione dell'evento. N.B. l'azione di default per SIGCHLD è SIG_IGN (per questo è necessaria la wait). Attenzione però, solo con wait possiamo conoscere lo stato del figlio. Quindi è necessario che l'handler richiami la wait, che in questo caso non sarà sospensiva poiché è sicuro che sia presente un figlio.

6.5.2 Fork e segnali

Le associazioni segnali-azioni vengono registrate nella user structure del processo e quindi di tutti i suoi figli. Per questo anche i figli ereditano le informazioni riguardanti la gestione degli eventi. Ovviamente le successive signal del figlio non hanno effetto sulla gestione dei segnali del padre, perché appunto è una copia.

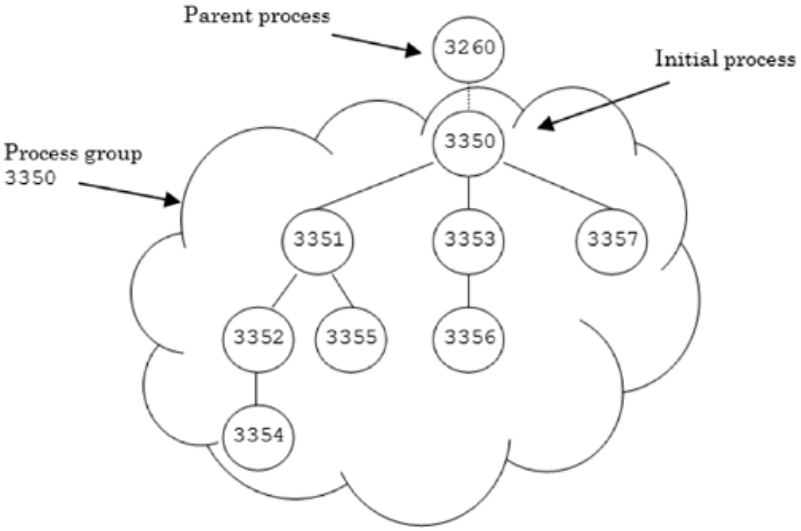
6.5.3 Exec e segnali

Dopo una exec() un processo mantiene la stessa user structure, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo l'exec non sono più visibili). Tutti gli handler settati vengono ripristinati a quelli di default, mentre quelli ignorati o a default rimangono invariati.

6.6 Kill

int kill(int pid, int sig);

I processi possono inviare segnali ad altri processi con la kill. N.B. Quando si parla di gruppo con groupId x, si intende la parte di gerarchia dei processi con padre avente pid = x.



- **sig**: è l'intero che individua il segnale da inviare (per esempio SIGUSR1).
- **pid**:
  1. se pid>0 l'intero è il pid dell'unico processo destinatario.
  2. se pid=0 il segnale è spedito a tutti i processi appartenenti al gruppo del mittente.

3. pid<-1 il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid.
4. pid=-1 vari comportamenti possibili (POSIX non specifica).

6.7 Sleep

unsigned int sleep(unsigned int N);

Provoca la sospensione del processo per un massimo di N secondi al massimo. Se il processo riceve un segnale durante il periodo di sospensione viene risvegliato prematuramente. Restituisce 0 se la sospensione non è stata interrotta da segnali oppure, se il risveglio è stato causato da un segnale al tempo Ns, restituisce il numero di secondi non utilizzati nell'intervallo di sospensione (N-Ns).

6.8 Alarm

unsigned int alarm(unsigned int N);

Imposta un timer che dopo N secondi invierà al processo il segnale SIGALRM. Ritorna 0 se non vi erano time-out impostati in precedenza oppure il numero di secondi mancante allo scadere del time-out precedente.
NB La alarm() non è una funzione sospensiva, il processo continua la sua normale esecuzione. L'azione di default associata a SIGALRM è la terminazione.

6.9 Pause

int pause(void);

Sospende il processo fino alla ricezione di un qualunque segnale. Ritorna -1 (errno = EINTR), ma solo in caso di errore in quanto non dovrebbe essere possibile accedere al valore di ritorno (viene invocato direttamente l'handler).

6.10 Sicurezza dei segnali

Durante l'esecuzione di un syscall è possibile che il processo riceva un segnale. Esistono tre alternative di gestione:

1. Innestamento delle routine di gestione.
2. Perdita del segnale.
3. Accodamento dei segnali.

In generale solo le syscall cosiddette **slow**, sono interrompibili da un segnale. In questo caso la syscall soggetta dell'interruzione restituirà -1/setterà errno a EINTR.
In questo caso in base alla situazione e alla syscall, la ri-esecuzione può essere automatica oppure comandata dal processo basandosi sul valore restituito.

7 File system

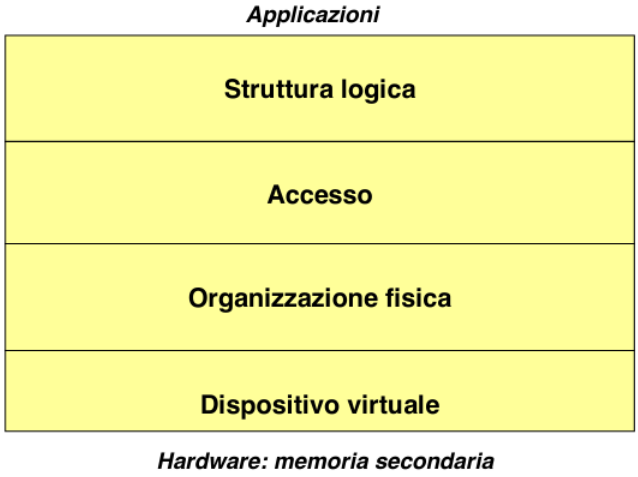
E' quella componente del sistema operativo che fornisce i meccanismi di accesso e memorizzazione delle informazioni (programmi e dati) allocate in memoria di massa.

Esso permette di realizzare i concetti astratti di:

- **File**: unità logica di memorizzazione.
- **Direttorio**: insieme di file.
- **Partizione**: insieme di file associato ad in particolare dispositivo fisico.

7.1 Organizzazione del file system

La struttura di un file system può essere rappresentata da un insieme di componenti organizzate in vari livelli:



- **Struttura logica**: presenta alle applicazioni una visione astratta delle informazioni memorizzate, basata su file, directory, partizioni, ecc.. Realizza le operazioni di gestione di file e directory: copia, cancellazione, spostamento, ecc.
- **Accesso**: definisce e realizza i meccanismi per accedere al contenuto dei file; in particolare:
  - Definisce l'unità di trasferimento da/verso file: record logico.
  - Realizza i metodi di accesso (sequenziale, casuale, ad indice).
  - Realizza i meccanismi di protezione.
- **Organizzazione fisica**: rappresentazione di file e directory sul dispositivo:
  - **Allocazione** dei file sul dispositivo (unità di memorizzazione = blocco): mapping di record logici su blocchi. Vari metodi di allocazione.
  - **Rappresentazione** della struttura logica sul dispositivo.
- **Dispositivo Virtuale**: presenta una vista astratta del dispositivo, che appare come una sequenza di blocchi, ognuno di dimensione data costante.

7.2 Struttura logica

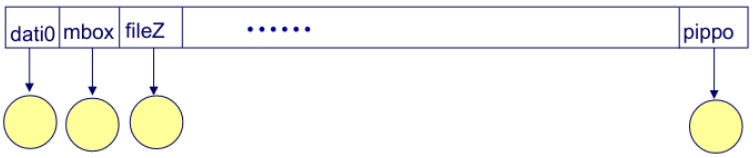
Un file è un insieme di informazioni (programmi, dati (in rappresentazione binaria), dati (in rappresentazione testuale)). Ogni file è individuato da (almeno) un nome simbolico (Linux permette di averne più di uno) ed è caratterizzato da un insieme di attributi: tipo, indirizzo(puntatore alla memoria secondaria), dimensione, data e ora dall'ultima modifica.
Nei sistemi multiutente è specificato anche il **proprietario** e i **diritti di accesso** per gli altri utenti del sistema.

Descrittore del file: è la struttura dati che contiene gli attributi del file. Ogni descrittore di file deve essere memorizzato in modo persistente:il SO mantiene l'insieme dei descrittori di tutti i file presenti nel file system in apposite strutture in memoria secondaria.

7.2.1 Directory

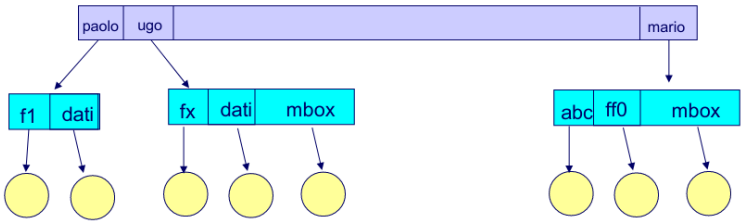
L'organizzazione delle directory varia da sistema operativo a sistema operativo. Gli schemi più comuni che sono stati adottati sono: a livello, a due livelli, ad albero, a grafo ciclico.

- **A livello:** una sola directory per ogni file system.

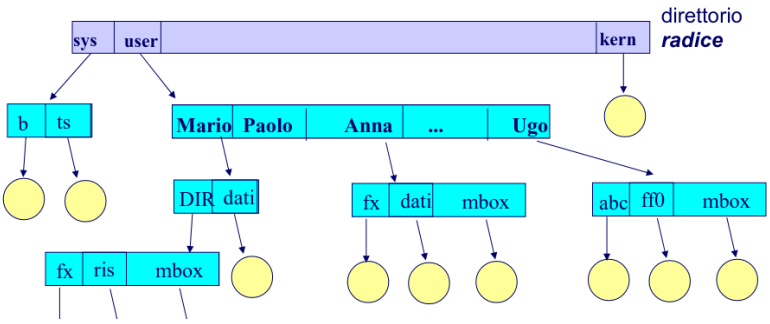


  Difficoltà di gestione della multiutenza e dell'unicità dei nomi.

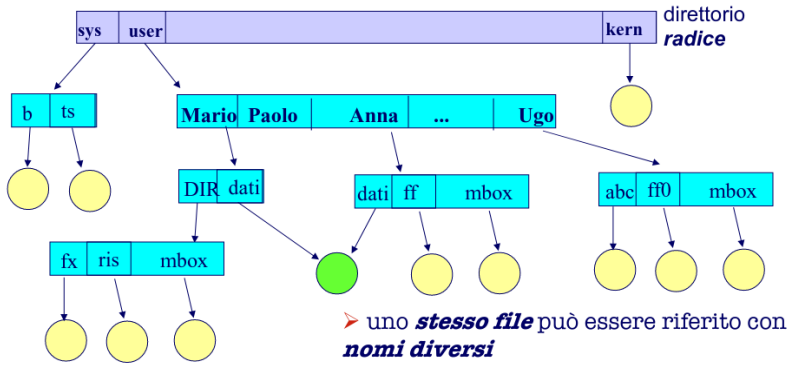
- **Due livelli**: primo livello (directory principale): contiene una directory per ogni utente del sistema. Secondo livello: directory utenti (a un livello).



- **Struttura ad albero**: organizzazione gerarchica a N livelli. Ogni direttorio può contenere file e altri direttori.



- **Struttura a grafo aciclico (es. UNIX)**: estende la struttura ad albero con la possibilità di inserire **link** differenti allo stesso file.



7.2.2 Partizione

Una partizione contiene un'unica struttura fra quelle sopra elencate. Perciò se un file system contiene più di una partizione, allora contiene più schemi ad albero, a grafo aciclico ecc...(di fatto una partizione è una gerarchia). Quello che il filesystem fa solitamente è di collegare più partizioni logicamente, attraverso il meccanismo di mounting. Alcuni SO richiedono solitamente il **mounting esplicito** della partizione all'interno del file system.

7.3 Accesso

Compito del SO è consentire l'accesso **on-line** ai file: ogni volta che un processo modifica un file, tale cambiamento è immediatamente visibile a tutti gli altri processi.
Ogni volta che devo accedere ad un file devo sapere: indirizzi dei record logici a cui accedere, altri attributi del file (diritti di accesso, ecc.), contenuto del file(record logici). Se ogni volta devo accedere a tutte queste informazioni passando per la memoria secondaria è un costo elevatissimo. Perciò ogni volta che si apre un file il SO aggiorna in memoria una struttura dati che registra i file attualmente in uso (i descrittori dei file aperti). Inoltre viene fatto il **memory mapping** dei file aperti: i file aperti (o porzioni di essi) vengono temporaneamente copiati in memoria centrale rendendo gl accessi più veloci.
Questo necessita di due operazioni aggiuntive:

- **Apertura**: introduzione di un nuovo elemento nella tabella dei file aperti e eventuale memory mapping del file.
- **Chiusura**: salvataggio del file in memoria secondaria ed eliminazione dell'elemento corrispondente dalla tabella dei file aperti.

7.3.1 Struttura interna dei file

Ogni dispositivo di memorizzazione secondaria viene partizionato in blocchi(o record fisici).

> **Blocco**: unità di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Sempre di dimensione fissa.

Le applicazioni vedono il file come un insieme di record logici:

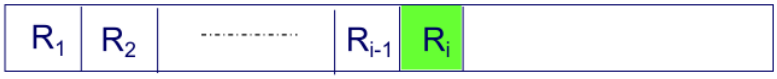
> **Record logico**: unità di trasferimento logico nelle operazioni di accesso al file (es. lettura, scrittura di blocchi). Solitamente di dimensione variabile.

Vale la relazione: **RECORD LOGICO << BLOCCO**. Quindi i record logici vengono **impaccati** all'interno dei blocchi. Sarà il sistema operativo quindi che si occuperà di stabilire *la corrispondenza tra record logici e blocchi.*

L'accesso a file può avvenire secondo varie modalità: accesso sequenziale, accesso diretto, accesso a indice. Il metodo di accesso è indipendente: dal tipo di dispositivo utilizzato e dalla tecnica di allocazione dei blocchi in memoria secondaria.

7.3.2 Accesso sequenziale

Il file è rappresentato a livello di accesso come una sequenza **ORDINATA [R1 , R2 , ..., RN]** di record logici.
Per accedere ad un particolare record logico Ri, è necessario accedere prima agli (i-1) record che lo precedono nella sequenza. Per ogni file aperto quindi verrà mantenuta l'informazione del puntatore al record logico corrente. Per leggere/scrive gli il prossimo record logico è possibile usare la **readnext** e la **writenext**.



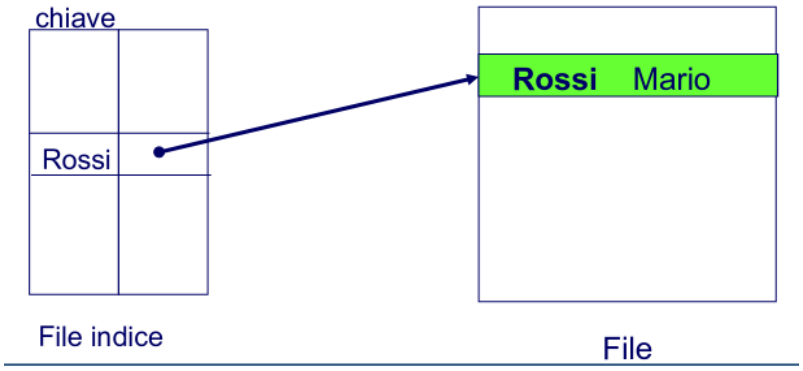
Questa tecnica di accesso è adottata dalla famiglia UNIX.

7.3.3 Accesso diretto

Il file è un insieme R1 , R2 , ..., RN di record logici numerati: si può accedere direttamente a un particolare record logico specificandone il numero.
È possibile utilizzare operazioni del tipo: **read i** e **write i**. Questo risulta molto utile quando si vuole accedere a grossi file per estrarre/aggiornare poche informazioni (ad esempio nell'accesso a database).

7.3.4 Accesso a indice

Ad ogni file viene associata una struttura dati contenente l'indice delle informazioni contenute. Per accedere a un record logico, si esegue una ricerca nell'indice (utilizzando una chiave). Questo tipo di accesso è poco diffuso.



7.3.5 Protezione

In fase di accesso è necessario garantire la protezione in scrittura, lettura ed esecuzione. In UNIX ogni utente appartiene ad uno o più gruppi su cui possono essere definiti dei privilegi di accesso per una determinata directory o file. I privilegi sono definiti quindi su tre livelli: **owner, group, public**. Esiste però un superuser che può modificare e gestire ogni singolo file/directory.

7.4 Organizzazione fisica

Questo livello si occupa della realizzazione fisica del file system sul dispositivo fisico. In particolare si occuperà di realizzare i descrittori dei file e di organizzarli nella memoria secondaria. Inoltre sarà necessario gestire anche lo spazio libero sulla memoria secondaria.
Il **blocco** è l'unità di allocazione su disco. Ogni blocco è associato ad un unico file e contiene un insieme di record logici. Le tecniche più comuni di allocazione su disco sono: **allocazione contigua, allocazione a lista, allocazione a indice**.

7.4.1 Allocazione contigua

Ogni file è mappato su un insieme di blocchi fisicamente contigui. Il principale vantaggio è il costo della ricerca di un blocco. possibilità di *accesso sequenziale e diretto.*

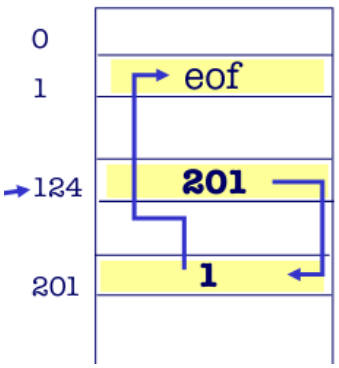
Il problema principale è l'individuazione dello spazio libero per l'allocazione di un nuovo file. Inoltre si crea **frammentazione esterna**: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili perché non c'è abbastanza . Questo richiede azioni di **compattazioni o frammentazione** del disco: si riportano tutti i blocchi allocati in posizione contigue eliminando gli spazi tra i file. il costo di quest'operazione è molto onerosa, i file poi potrebbero cambiare di nuovo, quindi va considerata una minima tolleranza affinché i file possano cambiare.
Quindi nel descrittore del file è sufficiente salvare dove sta il primo blocco del file.

7.4.2 Allocazione a lista concatenata

I sistemi Windows utilizzano questa tecnica. I blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata. I vantaggi sono l'eliminazione della frammentazione esterna e un minor costo di allocazione, poiché è facile individuare dove allocare un nuovo file (il file system mantiene informazione dei blocchi liberi). Non serve che i blocchi siano contigui, basta che sia presente un numero sufficiente di blocchi sul dispositivo.
Gli svantaggi possono essere una minor resistenza ai guasti: se il link è errato c'è una *perdita di concatenamento* e tutta la coda non potrà mai essere eliminata perché è persa. Inoltre i puntatori occuperanno un certo spazio nel blocco, diminuendone quindi la capacità.
L'accesso diretto è più complesso da implementare e il costo di ricerca di un blocco è maggiore.

Quindi nel descrittore del file è sufficiente salvare dove sta il primo blocco del file

Alcuni SO (ad es. windows, OS/2, dos, ntfs) realizzano l'allocazione a lista in modo più efficiente e robusto: per ogni partizione, viene mantenuta una tabella (**FAT - file allocation table**) in cui ogni elemento rappresenta un blocco fisico. Concatenamento dei blocchi sui quali è allocato un file è rappresentato nella FAT.



Questo permette di avere un backup e un metodo di scorrimento della lista più efficiente e veloce.

7.4.3 Allocazione a indice

tutti i puntatori ai blocchi utilizzati per l'allocazione di un determinato file sono concentrati in un unico blocco per quel file (**blocco indice**). Il blocco indice ovviamente comporta un maggior utilizzo di memoria, soprattutto se parliamo di file di piccole dimensioni. Solitamente il tempo di accesso di allocazione a lista è più lenta di quella contigua, per questo quella a indice può essere un'alternativa.
I vantaggi sono gli stessi dell'allocazione a lista e in più è facilitata la realizzazione ad accesso diretto(nel blocco indice i blocchi sono sequenziali). Inoltre si ottiene maggiore velocità di accesso (rispetto a liste). Quindi nel descrittore del file è sufficiente salvare l'indirizzo del blocco indice.

Quale tecnica è la migliore? Dipende dal contesto. Esistono sistemi operativi che utilizzano soluzioni ibride in base alla dimensione del file (piccola-contigua, grandi-indice). Solitamente i fattori da considerare sono:

- Il grado di utilizzo della memoria.

- Tempo di accesso medio al blocco.
- Realizzazione dei metodi di accesso.

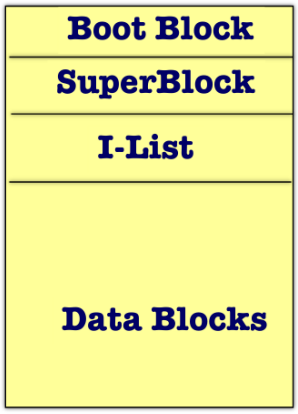
Solitamente Windows predilige allocazione concatenata con supporto a FAT, mentre UNIX predilige quella a indice.

7.5 Il File System di UNIX

L'organizzazione logica di UNIX è a grafo aciclico diretto. Il filesystem LINUX è solitamente **EXT**. Tutto in UNIX è un file, e sono divisi in tre tipi di file: **file ordinari, direttori, dispositivi fisici(nella directory /dev)**.
Ad ogni file è associato un unico descrittore: **i-node** identificato da un intero **i-number**. Gli i-node sono salvati in un vettore **i-list** in cui l'indice è l'i-number. Ogni file sono associati uno o più nomi.

7.5.1 Organizzazione fisica

Il metodo di allocazione utilizzato in UNIX è quello a indice. La formattazione del disco avviene in blocchi fisici di dimensione: **512-4096 Bytes**.
La superficie del disco File System è partizionata in 4 regioni:



- **Boot block**: contiene le procedure di inizializzazione del sistema (da eseguire al **bootstrap**).
- **Super block**: fornisce i limiti delle 4 regioni (quindi il partizionamento), il puntatore a una lista dei blocchi liberi, il puntatore a una lista degli i-node liberi.
- **I-list**:contiene i descrittori di tutti i file.
- **Data block**: è l'area del disco effettivamente disponibile per la memorizzazione dei file.

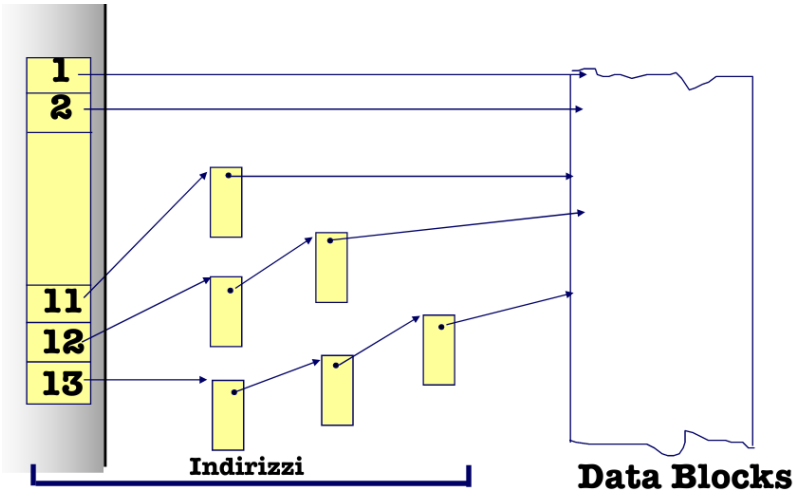
7.5.2 i-node

È il descrittore del file. Contiene i seguenti attributi:

1. Tipo di file: ordinario, direttorio, speciale(dispositivi).
2. User-id del proprietario, e group-id
3. Dimensione.
4. Data.
5. 12 bit di protezione.
6. Numero di links (più nomi).
7. 13-15 indirizzi di blocchi, a seconda della versione.

A cosa servono quei 13-15 indirizzi di blocchi? Anche se UNIX utilizza l'allocazione a indice, quindi l'allocazione del file non è su blocchi fisicamente contigui, nell' i-node sono contenuti puntatori a blocchi (ad esempio 13), dei quali:

- I primi 10 riferiscono blocchi di dati (indirizzamento diretto)
- 11esimo indirizzo: indirizzo di un blocco indice, contenente a sua volta indirizzi di blocchi dati (**primo livello di indirettezza**).
- 12esimo indirizzo: indirizzo di un blocco indice che contiene indirizzi di blocchi indice **due livelli di indirettezza**.
- 13esimo indirizzo: indirizzo di un blocco indice che contiene indirizzi di blocchi indice che contengono indirizzi di altri blocchi indice.**tre livelli di indirettezza**.



ò512 byte un blocco, 32 byte un indirizzo, 128 indirizzi per blocco. 5KB accessibili direttamente, 128*512 a **indirezione singola** = 64KB, 128*128*512 a **indirezione doppia** = 8MB, 128*128*128*512 a **indirezione tripla** = 1GB. Più il file aumenta di dimensione più l'accesso risulterà costoso a causa dei livelli di indirettezza.

7.5.3 Direttorio

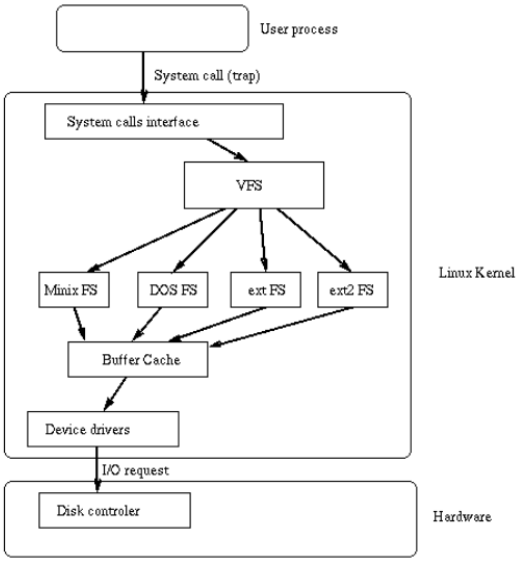
Anche le directory sono rappresentate nel file system da un file. Il file contiene un insieme di record con la struttura *nome relativo: i-number*, uno per ogni file contenuto nella directory, con l'aggiunta delle directory **. e ..** .
N.B. Ricorda che per eseguire cd su una directory servono i diritti di esecuzione sulla directory. Quindi in caso di esecuzione di una cd, si apre il file della directory corrente e si cerca la directory obiettivo della cd, si legge il record e si va a leggere nella i-list il descrittore della directory per capire se si hanno i permessi.

7.5.4 EXT2

Il file system EXT2 è stato adottato da Linux nel 1993 al posto di EXT e riesce a gestire da 16GB a 4TB. L'amministratore può decidere la dimensione del blocco da 1024 a 4096 bytes e la dimensione dell'i-list. Inoltre il sistema prevede la creazione di gruppi i quali includono data blocks e i-node memorizzati in tracce adiacenti e una copia delle strutture di controllo (superblock e descrittore filesystem) aumentando di fatto l'affidabilità. Così viene garantita la località di inode e relativi file (all'interno dello stesso gruppo) e di tutti i blocchi di un file.

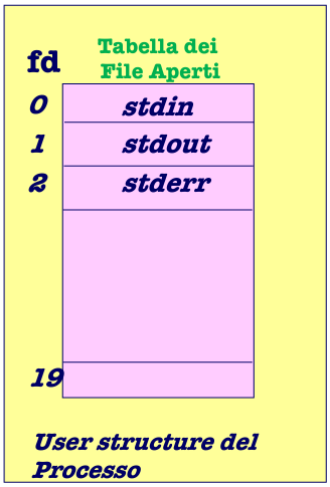
EXT2 è stato sostituito con **EXT4** il quale aggiunge in un file di log tutte le azioni operate al disco (sistema di *journaling*).

Linux prevede l'integrazione con filesystem diversi da Ext, grazie al **Virtual File System** il quale intercetta ogni system call relativa all'accesso al file system e, all'occorrenza, provvede al collegamento con file system "esterni".



7.5.5 Livello di accesso

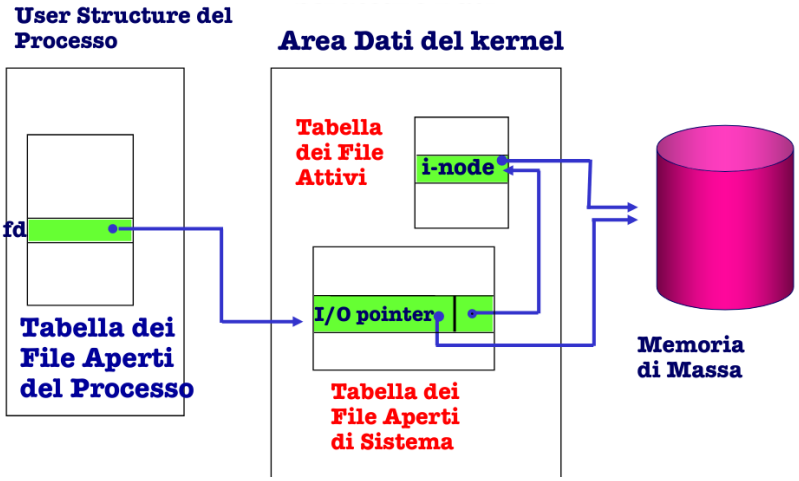
UNIX utilizza l'**accesso sequenziale**. Il puntatore al file si chiama **I/O pointer** e registra la posizione corrente. Il record logico è il singolo byte, quindi il file è visto come una sequenza di byte.
In generale ad ogni processo è associata una *tabella dei file aperti* di dimensione limitata (tipicamente 20) presente nella user structure del processo. Ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero: **file descriptor**. Inoltre contiene un puntatore al In ogni tabella esistono tre file aperti automaticamente: **0-stdin, 1-stout, 2-stderr**.



Il kernel gestisce altre due strutture dati, globali, allocate nell'area dati del kernel:

- La **tabella dei file**:per ogni file aperto, contiene una copia del suo i-node onde evitare ripetuti accessi al disco.
- la **tabella dei file aperti di sistema**: ha un elemento per ogni operazione di apertura relativa a file (aperti e non ancora chiusi); ogni elemento contiene:
  - l'I/O pointer, che indica la posizione corrente all'interno del file.
  - un puntatore all' i-node del file nella tabella dei file attivi.

Quindi se due processi aprono separatamente lo stesso file F, la tabella conterrà due elementi distinti associati a F.



In caso di fork, il figlio eredita la copia dei file aperti e quindi anche il puntatore allo stesso record della tabella dei file aperti. *Padre e figlio condividono lo stesso IO pointer.*

7.6 System call di accesso a file

L'apertura di un file comporta l'inserimento di un elemento (individuato da un file descriptor) nella prima posizione libera della Tabella dei file aperti del processo, l'inserimento di un nuovo record nella Tabella dei file aperti di sistema e la copia dell' i-node nella tabella dei file attivi (se il file non è già in uso).

7.6.1 Open

int open(char nomefile[],int flag, [int mode]);

- **Nomefile** è il nome del file (relativo o assoluto).
- **Flag** esprime il modo di accesso; ad esempio **O_RDONLY**, per accesso in lettura, **O_WRONLY**, per accesso in scrittura, **O_RDWR** per accesso in scrittura e lettura.
- **Mode** è un parametro richiesto soltanto se l'apertura determina la creazione del file (flag **O_CREAT**): in tal caso, mode specifica i bit di protezione (ad esempio, codifica ottale di 4 cifre).
- Il valore restituito dalla open è il file descriptor associato al file, o -1 in caso di errore

Se la open ha successo, il file viene aperto nel modo richiesto, e l'I/O pointer posizionato sul primo elemento (tranne nel caso di **O_APPEND**). Nei flag è possibile abbinare anche O_CREAT e **O_TRUNC**, per accesso in scrittura: la lunghezza del file viene troncata a 0. L'abbinamento deve essere effettuato con l'operatore **bitwise OR** |.
Le modalità di apertura sono definite nella libreria <**fcntl.h**>.

7.6.2 Creat

int creat(char nomefile[], [int mode]);

- **Nomefile** è il nome del file (relativo o assoluto) da creare.
- **Mode** specifica i 12 bit di protezione per il nuovo file.

- il valore restituito dalla creat è il file descriptor associato al file, o -1 in caso di errore.

N.B. Il file creato è sempre aperto in scrittura di default.
Esempio: **fd2=creat("f2.new", 0777);**
che è del tutto equivalente a **fd2=open("f2.new",O_WRONLY|O_CREAT, 0777);**

7.6.3 Close

int close(int fd);

- **Fd** è il file descriptor del file da chiudere.
- Restituisce l'esito della operazione (0, in caso di successo, <0 in caso di insuccesso).

Se la close ha successo il file viene memorizzato sul disco, viene eliminato l'elemento di indice fd dalla Tab. dei file aperti del processo e Vengono eventualmente eliminati (se non condivisi con altri processi) gli elementi corrispondenti dalla Tab. dei file aperti di sistema e dalla tabella dei file attivi.

7.6.4 Read

int read(int fd, char *buf, int n);

Ogni operazione è *sequenziale, atomica e sincrona.*

- **Fd** è il file descriptor del file da leggere.
- **buf** è l'area in cui trasferire i byte letti.
- **n** è il numero di caratteri da leggere.
- In caso di successo, restituisce un intero positivo (¡=n) che rappresenta il numero di caratteri effettivamente letti. (se n=15 ma il file ne contiene ancora 9, la read restituisce 9. Questo non significa che la read è fallita).

L'IO pointer verrà incrementato di n bytes. Alla fine di ogni file è presente il carattere **eof**.

7.6.5 write

int write(int fd, char *buf, int n);

Ogni operazione è *sequenziale, atomica e sincrona.*

- **Fd** è il file descriptor del file da scrivere.
- **buf** è l'area da cui trasferire i byte da scrivere.
- **n** è il numero di caratteri da scrivere.
- in caso di successo, restituisce un intero positivo uguale a n, che rappresenta il numero di caratteri effettivamente scritti. Se il numero è diverso da n, la write è fallita.

L'IO pointer verrà incrementato di n bytes. Alla fine di ogni write, l'IO pointer punta sempre alla fine del file eof.

7.6.6 Lseek

int lseek(int fd, int offset, int origine);

System call atta allo spostamento dell'I/O pointer.

- **Fd**: è il file descriptor del file.
- **Offset**: è lo spostamento in byte rispetto all'origine.
- **Origine**: **SEEK_SET - 0** inizio del file, **SEEK_CUR - 1** posizione corrente, **SEEK_END - 2** fine del file.
- In caso di successo restituisce un intero che rappresenta la nuova posizione dell'I/O pointer.

N.B. lseek(fd,0,2) restituisce la dimensione in byte del file.

7.6.7 Unlink

int unlink(char* name);

In generale l'effetto della system call unlink è decrementare di 1 il numero di link del file specificato (viene modificato il numero di links nell'i-node); nel caso in cui il numero dei link risulti 0 (cosa che accade nella gran parte dei casi), allora il file viene cancellato deallocando tutti i blocchi di quel file nella memoria di massa.
N.B. il file non deve essere stato necessariamente aperto dal processo.

- **Name**: nome del file.
- Restituisce 0 se l'operazione è andata a buon fine, altrimenti restituisce -1.

7.6.8 Link

int link(char* oldname, char* newname);

Incrementa il numero dei link associato al file (viene modificato il numero di links nell'i-node), aggiorna il direttorio (aggiunta di un nuovo elemento). È l'operazione inversa dell'unlink.
N.B. il file deve esistere già, non crea un file!! Per quella c'è la creat.

- **Oldname**: nome del file a cui aggiungere un link.
- **Newname**: nome da aggiungere al file.
- Ritorna 0 in caso di successo, -1 se fallisce. Questo accade quando oldname non esiste, newname esiste già oppure oldname e newname appartengono a file system diversi (in questo caso si usano **softlink** mediante **symlink**).

7.7 Protezione e privilegi

Ogni file è accessibile secondo tre diverse modalità: scrittura, lettura, esecuzione. Il proprietario può concedere o negare il permesso di accedere al file ad altri utenti. Mentre esiste un utente privilegiato (**root**) che ha accesso incondizionato ad ogni file del sistema.
Ad ogni file sono associati 12 bit di protezione (nell'i-node) di cui:

- 9 bit (**rwx**) di lettura (read), scrittura (write) ed esecuzione (execute) per utente proprietario (User), utenti del gruppo (Group) e tutti gli altri utenti (Others) - **UGO**.
- 3 bit di permessi per file eseguibili [**Set-User-Id(SUID), Set-Group-Id(SGID), Save-Text- Image(sticky)**]

Al processo che esegue un file eseguibile è associato dinamicamente uno **User-Id** (e **Group-Id**)), chiamato **User-ID effettivo**. Di default lo uid è quello che lancia il processo. In questo caso uid effettivo coincide con lo **User-ID reale**.
Nel caso si voglia cambiare questa impostazione (sempre nel caso di file eseguibili) è possibile settare a 1 SUID oppure SGID. Settando SUID è possibile cambiare il valore del uid effettivo in quello del proprietario del file. Lo scopo è quello di assumere gli stessi diritti del proprietario del file. SGID funziona in modo analogo con il gruppo.
Save-Text-Image l'immagine del processo rimane in area di swap anche dopo che il processo è terminato, in maniera tale da velocizzare un futuro riavvio.

7.7.1 /etc/passwd

Il formato del file /etc/passwd è il seguente: *nomeUtente:hashpasswd:uid:gid:descrizione utente: home directory utente:shell da lanciare dopo il login.* Tutti gli utenti possono leggere da questo file, ma nessuno può scrivere se non root. Ma allora come si può cambiare la password? Si utilizza il comando passwd. Questo è un chiaro esempio dell'utilità di SUID. L'utente quando vuole cambiare la propria password deve "fingere" di avere gli stessi privilegi di root per modificare /etc/passwd.

[-rwsr-xr-x 1 root root 59640 gen 25 17:26 /usr/bin/passwd] come si può notare passwd ha SUID=1 così da poter effettuare quello che è chiamato *cambio di dominio di protezione*: lo uid effettivo non ha lo stesso dominio di quello reale.
Per provare il campio di uid effettivo e non quello reale esegui in un terminale passwd, poi prova in un altro *ps -eo pid,euid,ruid,command—grep passwd.*

7.7.2 Access

int access(char* pathname, int amode);

Verifica, controllando il campo dei permessi dell'i-node, se su un dato file sono concessi i permessi specificati per l'utente con l'uid reale del file che esegue la primitiva. Ritorna un booleano (0 corrisponde a vero), esprime il diritto da verificare e può essere:(sono in codifica ottale) *00 existence (esistenza), 01 execute access (accesso in esecuzione), 02 write access (accesso in scrittura), 04 read access (accesso in lettura).*

7.7.3 Stat

int stat(const char *path, struct stat *buf);

Permette di analizzare in modo generale l'i-node di un file. Buff è un puntatore a una struttura di tipo STAT nella quale vengono restituiti gli attributi del file (definito nell'header file <**sys/stat.h**>). In caso di errore restituisce -1. Potrebbe essere definita così:

```
1   struct stat{
2       dev_t st_dev;     /* ID of physical device containingfile */
3       ino_t st_ino;     /* i-number*/
4       mode_t st_mode;    /* protection bit and filetype*/
5       nlink_t st_nlink;    /* number of hard links*/
6       uid_t st_uid;     /* userID of owner*/
7       gid_t st_gid;     /* groupID of owner*/
8       dev_t st_rdev;    /* deviceID (if special file) */
9       off_t st_size;    /* total size, in bytes*/
10      blksize_t st_blksize; /* block size for the file system */
11      blkcnt_t st_blocks; /* number of blocks allocated*/
12      time_t st_atime;    /* time of last access*/
13      time_t st_mtime;    /* time of last modification*/
14      time_t st_ctime;    /* time of last status change*/
15  };
```

Per interpretare il valore di st_mode, sono disponibili alcune costanti e macro in <sys/stat.h>. Ad esempio:

- **S_ISREG(mode)** è un file regolare? (flag S_IFREG)
- **S_ISDIR(mode)** è una directory? (flag S_IFDIR)

- **S_ISCHR(mode)** è un dispositivo a caratteri (file speciale)? (flag S_IFCHR)
- **S_ISBLK(mode)** è un dispositivo a blocchi (file speciale)? (flag S_IFBLK)

7.7.4 Chmod e chown

int chmod (char *pathname, char *newmode);
int chown(char *pathname, int owner, int group);

N.B. Per chown solamente root può eseguire la syscall.

7.8 Direttori

Così come per i file, lettura/scrittura di un direttorio può avvenire soltanto dopo l'operazione di apertura. Si noti NESSUN processo può scrivere su un direttorio, altrimenti ci sarebbero dei problemi di integrità e di sicurezza del file system. Solo il kernel può modificare un direttorio.

7.8.1 Chdir

int chdir (char *nomedir);

Cambia la directory corrente. Restituisce 0 in caso di successo.Ricorda può fallire se l'utente del processo non ha i diritti di esecuzione su quella directory.

7.8.2 Opendir

#include <dirent.h>
DIR *opendir (char *nomedir);

Permette di aprire una directory. La funzione restituisce un valore di tipo puntatore a DIR. Diverso da NULL se l'apertura ha successo: per gli accessi successivi, si impiegherà questo valore per riferire il direttorio. Restituisce NULL invece, in caso di insuccesso. (solitamente perché non ho i diritti o perché la directory non esiste).

7.8.3 Readdir

#include <sys/types.h>
#include <dirent.h> //per il tipo dirent
struct dirent *descr;
descr = readdir (DIR *dir);

Per leggere una directory si passa il valore restituito da opendir. In caso di successo, la readdir legge un elemento dal direttorio dato e restituisce un puntatore di tipo **dirent**. Se la read non ha successo restituisce NULL.

N.B. d_namelen è necessaria perché in UNIX i file sono di lunghezza variabile.

```
1 struct dirent {
2     long d_ino; /* i-number */
3     off_t d_off; /* offset della prossima entry*/
4     unsigned short d_reclen; /* lunghezza del record */
5     unsigned short d_namelen; /* lunghezza del nome */
6     char *d_name; /* nome del file */
7 }
```

7.8.4 Chdir

int mkdir (char *nomedir, int mode);

Creazione di una directory con bit di protezione specificati. In caso di successo nel file system comparirà un nuovo file contenente due entry: . e .. .

8 Comunicazione tra processi UNIX

Come visto in precedenza i processi Unix non possono condividere memoria (modello ad ambiente locale), l'iterazione tra processi, quindi, può avvenire: mediante le condivisione di file (molto complesso, sincronizzazione dei processi), oppure attraverso specifici strumenti di **IPC - Inter Process Communication** (pipe, fifo, socket).

8.1 Pipe

È un canale unidirezionale (accessibile ad un estremo in lettura ed all'altro in scrittura) con capacità limitata (è in grado di gestire l'accodamento di un numero limitato di messaggi, gestiti in modo FIFO) che permette la comunicazione tra processi in modalità asincrona. La pipe rappresenta un chiaro esempio di comunicazione indiretta (Mailbox), in quanto un processo può effettuare la comunicazione senza specificare direttamente il destinatario/mittente.
NB: la pipe può anche consentire una comunicazione "bidirezionale" tra i processi, ma va rigidamente disciplinata.
La scelta che è stata fatta è stata di garantire una certa omogeneità tra file e pipe, in modo da poter utilizzare le stesse syscall. Si accederà quindi alla pipe utilizzando la syscall read e write.

int pipe(int fd[2]);

- **Fd** è un puntatore a un vettore di 2 file descriptor, che verranno *inizializzati* dalla system call. Dopo l'esecuzione della pipe, fd[0] conterrà il file descriptor dell'estremo di lettura della pipe e fd[1] conterrà il file descriptor dell'estremo di scrittura della pipe.
- La system call restituisce: un valore negativo in caso di fallimento oppure 0 se ha successo.

La write e la read da/verso pipe possono essere sospensive, la sincronizzazione avviene automaticamente: in altre parole se la pipe è vuota, un processo che legge si blocca, viceversa, se la pipe è piena, un processo che scrive si blocca.
La grande limitazione è che soltanto i processi appartenenti a una stessa gerarchia (cioè, che hanno un antenato in comune) possono scambiarsi messaggi mediante la pipe. Questo perché se un padre esegue una pipe, solo i suoi discendenti potranno accedere alla perché la tabella dei file attivi di processo è estesa a tutto il gruppo del processo.

Chiusura della pipe: ogni processo può chiudere un estremo della pipe con una close (solitamente l'estremo della comunicazione che non usa). Ma un estremo della pipe viene effettivamente chiuso, quando tutti i processi che ne avevano visibilità hanno compiuto una close sui due file. Se un processo tenta una lettura da una pipe vuota in cui lato di scrittura è effettivamente chiuso, la read ritorna 0. Mentre se si tenta una scrittura su una pipe il cui lato di lettura è effettivamente chiuso: la write restituisce -1 e viene inviato il segnale **SIGPIPE** al processo.

La pipe non è persistenze. Non rimane traccia nel filesystem, del canale di comunicazione.

8.1.1 Dup

int dup(int fd);

L'effetto di una dup è copiare il recordo associato a fd nella tabella dei file aperti nella prima posizione libera nella tabella. Restituisce il nuovo file descriptor, oppure -1 in caso di errore. La principale funzionalità della dup è poter realizzare la ridirezione di comandi su file e la piping di comandi. N.B. senza la dup non si potrebbe creare la piping di comandi

```
1 main(int argc, char **argv){
2     int pid1, pid2, fd[2],i, status;
3     pipe(fd);
4     pid1=fork();
5     if (!pid1){
6         close(fd[1]);
7         close(0); //chiudo stdin
8         dup(fd[0]); /* ridirigo stdin sulla pipe */
9         close(fd[0]);
```

```
10          execlp(argv[2], argv[2],(char *)0);
11          exit(-1);
12          ...
13  }
```

8.2 FIFO

Affinché due processi possano comunicare anche se non appartengono allo stesso gruppo, Unix offre la **FIFO**. Questa, a differenza della pipe, è persistente e permette la comunicazione tra più processi non appartenenti alla stessa gerarchia.
La fifo è *unidirezionale* e del tipo first-in-first-out, è rappresenta da un file nel filesystem e quindi *persistente* ed è quindi garantita omogeneità con le modalità di accesso a file.

8.2.1 Mkfifo

int mkfifo(char* pathname, int mode);

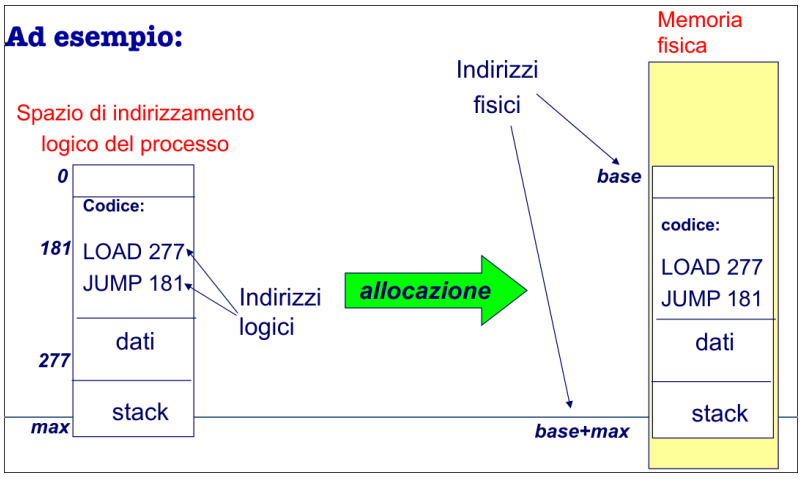
Permette di creare una file passando il nome del file (**pathname**) con i relativi permessi(**mode**). Restituisce 0 in caso di successo oppure un valore negativo in caso contrario. N.B. una mkfifo crea ma non apre il file!!! Quindi stai attento, dopo aver creato la fifo, ricordati di aprire il file.
Per eliminare una fifo utilizzare la primitiva unlink.

9 Multiprogrammazione e gestione della memoria

L'obiettivo principale della multiprogrammazione è l'uso efficiente delle risorse computazionali (efficienza nell'uso della CPU, velocità di risposta dei processi...) e la necessità di mantenere più processi in memoria centrale, gestendo la memoria in modo da consentire la presenza contemporanea di più processi, garantendo la sicurezza e l'integrità dei dati. A livello hardware ogni sistema ha un unico spazio di memoria accessibile direttamente da CPU e dispositivi.
Compiti del Sistema Operativo sono:

- **Accesso alla memoria centrale**: deve svolgere load e store di dati e istruzioni. Gli indirizzi possono essere simbolici (riferimenti a celle fisiche utilizzati nel codice **sorgente** mediante nomi simbolici come variabili, funzioni ecc...), logici (riferimenti a celle nello spazio logico di indirizzamento) e fisici(riferimenti assoluti a livello fisico). Ogni processo dispone di un proprio spazio di indirizzamento logico che viene allocato nella memoria fisica.

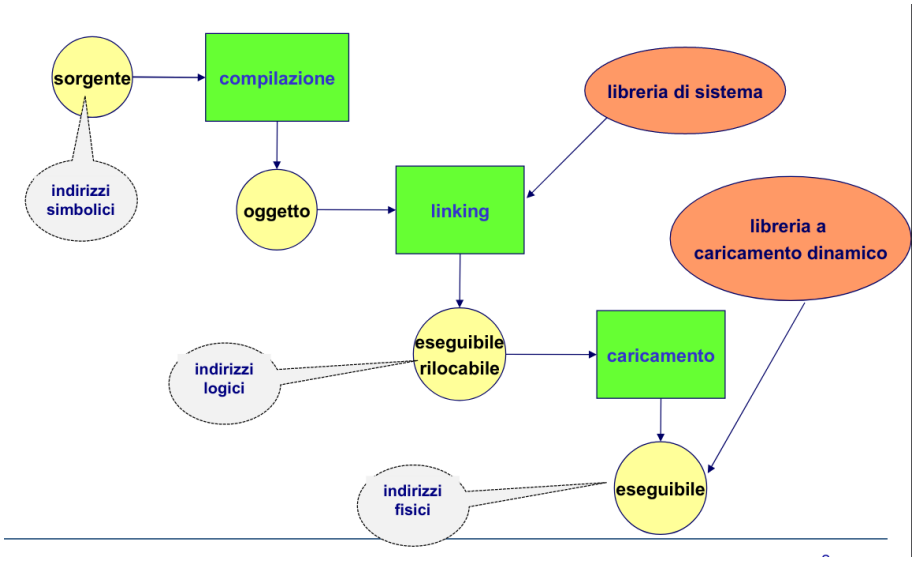
N.B. lo **spazio di indirizzamento logico di un processo** è un intervallo sempre continuo e va da 0 fino ad un massimo P. Nella memoria centrale invece, l'intervallo avrà un certo offset rispetto alla base della memoria, che potrà variare con il passare del tempo ==> necessità di binding dinamico. In genere, un programma in forma eseguibile contiene riferimenti allo spazio logico di indirizzamento e non agli indirizzi fisici! N.B. gli indirizzi simbolici non vengono risolti tramite il binding degli indirizzi. Associazione simbolo-indirizzo logico viene svolto in fase di compilazione.



- **Binding degli indirizzi**, ossia l'associazione ad ogni indirizzo logico ad un indirizzo fisico (assoluto). Può essere svolto:
  - **Staticamente**, solo se si è a conoscenza dell'allocazione in memoria a priori. A sua volta può essere svolto a tempo di compilazione: il compilatore sostituisce i simboli con gli indirizzi fisici (es. DOS), oppure a tempo di caricamento: in cui il compilatore genera degli indirizzi relativi che saranno poi convertiti dal loader in indirizzi assoluti. Con quest'ultima

– **dinamicamente**, a tempo di esecuzione: in cui un processo può essere spostato da un' area a un'altra durante l'esecuzione. Questo meccanismo viene realizzato con un'infrastruttura hardware dedicata atta a velocizzare l'operazione di binding.

Le fasi di sviluppo possono essere riassunte nel seguente schema. Notare che l'*eseguibile rilocabile* esplicita la libertà del caricatore di allocare in una qualunque zona di memoria.

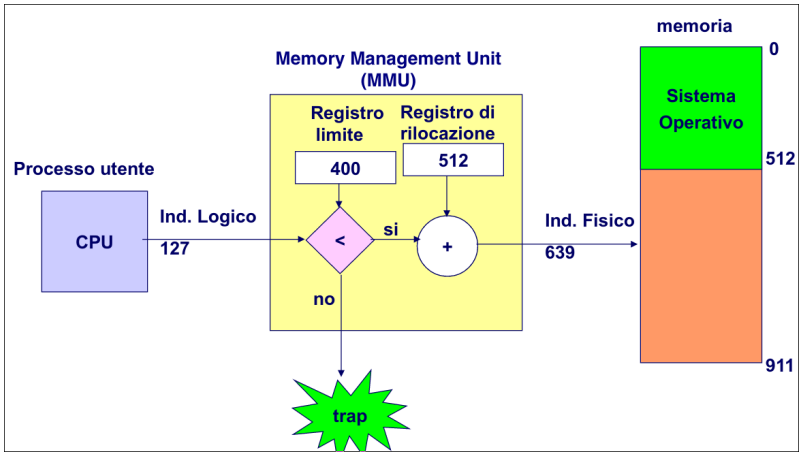


9.1 Allocazione della memoria centrale

Esistono più modi di allocazione della memoria centrale: **allocazione contigua** (gli indirizzi fisici di un processo sono continui come lo spazio di indirizzamento logico) e **allocazione non contigua**.

9.1.1 Allocazione contigua a partizione singola

Primo approccio molto semplificato: la parte di memoria disponibile per l'allocazione dei processi di utente non è partizionata. Questo permette a un solo processo alla volta di risiedere in memoria centrale, non permettendo di fatto la multiprogrammazione.
La memoria viene quindi suddivisa in due aree: una parte bassa solitamente affidata al SO e la parte alta al processo in esecuzione. Quale garanzia si ha che il processo risiedente in memoria non acceda alla zona di memoria del SO, illegalmente? Viene utilizzato un **registro di rilocazione** (offset dell'indirizzo logico del processo pari a SO_MAX+1) che viene sempre sommato all'indirizzo logico, il quale deve essere sempre compreso tra 0 e il **registro limite**.



Si introduce il **MMU - memory management unit**.

9.1.2 Allocazione contigua a partizione multipla

Affinché si possa realizzare la multiprogrammazione, si introducono più partizioni, una per processo e indipendenti.

- **Partizioni fisse (MFT, Multiprogramming with Fixed number of Tasks**: La memoria fisica disponibile per lallocazione dei processi è suddivisa a priori in un numero prefissato di partizioni. Così facendo la dimensione di ogni partizione è conosciuta a priori. Quando un processo viene caricato, il SO cerca una partizione libera di dimensione sufficiente ad accogliere il suo spazio di indirizzamento (si sceglie la più piccola con queste caratteristiche per diminuire la frammentazione interna).

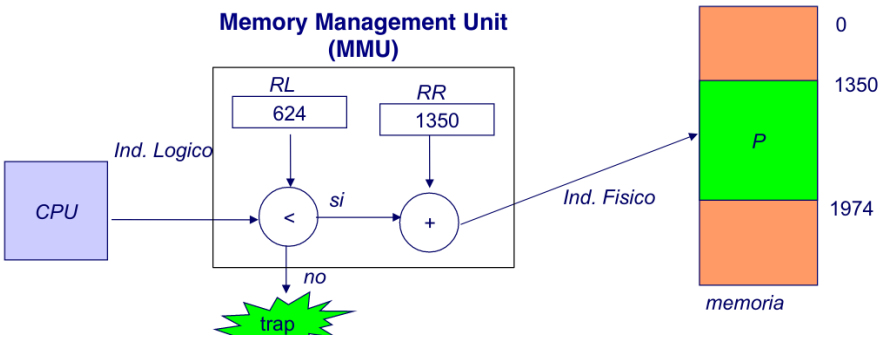
  Non sempre è possibile trovare la partizione idonea per un dato processo: si potrebbero verificare dei problemi di sottoutilizzo della memoria - **frammentazione interna**. Inoltre il grado di multiprogrammazione è dato dal numero di partizioni e il processo più grande che posso gestire è dato dalla dimensione della partizione più estesa in memoria.

- **Partizioni variabili (MVT, Multiprogramming with Variable number of Tasks)**: Ogni partizione è creata dinamicamente e dimensionata in base alla dimensione del processo da allocare: quando un processo viene caricato, SO cerca un'area sufficientemente grande per allocarvi dinamicamente la partizione associata (in realtà un po' più grande per lasciare spazio alla processo di espandersi). Elimina la frammentaione interna e il grado di multiprogrammazione è variabile. Inoltre il limite dello spazio di indirizzamento è limitato solamente allo spazio fisico disponibile.

  Il costo di questa dinamicità è dato dall'overhead introdotto dall'analisi dello spazio disponibile in memoria, che aumenta all'aumentare della grandezza della memoria. Per la gestione di questo overhead esistono più politiche:

  1. **First fit**, viene scelto il primo spazio disponibile per il processo a favore della velocità e della frammentazione esterna (memoria libera è sempre più frammentata) dovendo ricorrere alla **compattazione** (corrispettivo della frammentazione).
  2. **Best fit**, viene scelta la zona di memoria che minimizzi la frammentaione della memoria (zone di memoria libera più piccola).
  3. **Worst fit**, viene scelta la zona di memoria che massimizzi la frammentaione della memoria (zone di memoria libera più grande in modo da poter inserire nuovi processi).

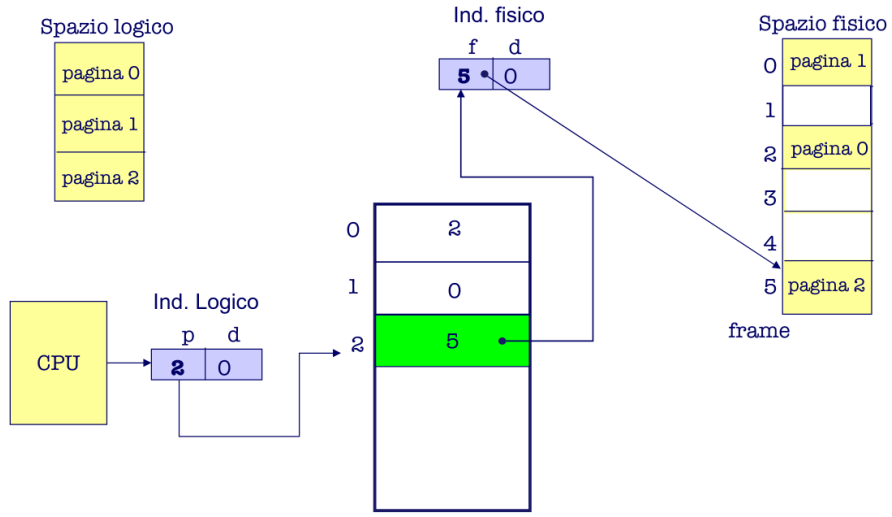
Per quanto riguarda la protezione, il discorso è un po' più complicato del caso a partizione singola. Nel caso di partizioni è sempre presente una MMU, in cui per ogni processo esiste una coppia di registri **registro di rilocazione -** V_{RR} e **registro limite -** V_{RL}. Il procedimento è uguale a quello della partizione singola, ad eccezione dei registri V_{RR} e V_{RL} che vengono cambiati ogni cambio di contesto.



9.1.3 Paginazione - allocazione non contigua

L'obiettivo è ridurre la paginazione interna la frammentazione interna e quella esterna. L'idea di base è la suddivisione a priori della memoria fisica in **frame/pagina fisica** di dimensione fissa, solitamente intorno ai 4 KB, che possa ospitare porzioni di processo. Lo spazio logico del processo sarà anch'esso suddiviso in **pagine logiche** di dimensione pari alle pagine fisiche. L'associazione pagine logiche pagine fisiche viene poi definita arbitrariamente dal SO.

Con questa politica elimino completamente la frammentazione esterna. Per allocare un nuovo processo di N pagine logiche, necessito di N pagine fisiche disponibili. N.B. La frammentazione interna non è completamente risolta, siccome lo spazio di indirizzamento di un processo potrebbe non essere un multiplo intero del frame. La frammentazione interna è limitata superiormente dal numero di celle contenute all'interno di un frame. Infine è possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo per realizzare la memoria virtuale.
Per quanto riguarda l'accesso in memoria, in un **sistema paginato** l'informazione è individuata da due coordinate: la pagina p e l'offset d relativo all'inizio della pagina. Quindi dati m bit per esprimere l'indirizzo: n bit saranno utilizzati per esprimere l'offset, mentre m-n bit verranno utilizzati per esprimere la pagina. Prendiamo il caso classico in un cui m:32 e n:12 m-n:20. Il numero di pagine che posso indirizzare è quindi pari a $2^{20} = 1MB$, mentre il massimo offset sarà pari a $2^{12} = 4KB$. Fisicamente l'informazione sarà anch'essa individuata da un offset, identico a quello logico, e da un numero di frame su cui sarà mappato una pagina logica del processo. Binding tra indirizzi logici e fisici può essere quindi realizzato mediante una tabella delle pagine associata al processo, su cui sarà salvato il mapping logico-fisico.



Come gestire la traduzione indirizzo logico - indirizzo fisico nel modo più veloce possibile? La tabella può essere molto grande per processi con spazio di indirizzamento molto ampi. Si potrebbe spostare la tabella delle pagine associate al processo nei registri della CPU, ma questo aumenterebbe l'overhead introdotto dai cambi di contesto e limitando la dimensione massima della tabella. Le due principali soluzioni solitamentre adottate sono:

1. La tabella risiede in memoria centrale e il suo indirizzo di partenza è salvato in un registro **page table base register - PTBR**. Per ogni accesso necessito di due operazionI, una di traduzione e una di accesso all'informazione.

2. Nei sistemii moderni la tabella è parzialmente salvata nella cache. In questa è presente la **translation look-aside buffers - TLB**, una struttura atta a memorizzare parte della tabella delle pagine associate al processo (solitamente prendo le pagine con accesso più frequente). L'accesso è *associativo*, quindi se una pagina è già presente in cache l'accesso è veloce, altrimenti è necessario spostare le pagine target dalla memoria centrale alla cache.

   La TLB parte quindi vuota, riempiendosi con l'avanzare del processo. La gestione della TLB può esere valutata secondo un indice chiamato **HIT-RATIO**, che indica la percentuale media di volte in cui una pagina viene trovata in TLB.

La tabella delle pagine ha dimensione fissa e può non essere completamente utilizzata. Per capire quali entry della tabella siano valide e quali no si possono utilizzare due soluzioni: il **bit di validità**, 1 se quella pagina è valida 0 se non è valida, oppure mantenere all'interno di un registro **Page Table Length Register** il numero di pagine valide. Queste due soluzioni possono essere adottate contemporaneamente.

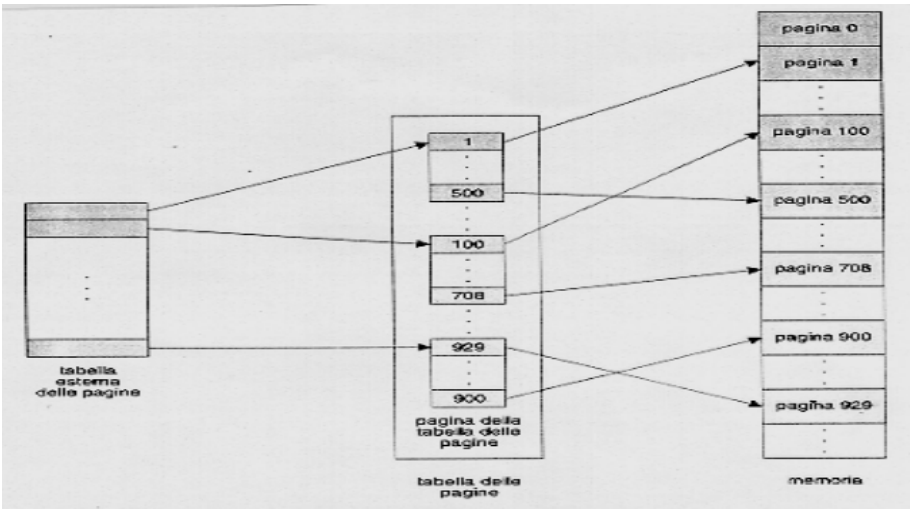
Inoltre per ogni pagina è possibile esprimere dei bit di protezione, per esempio esprimere il vincolo di *read-only* per quanto riguarda le pagine che contengono il codice.

9.1.4 Paginazione multilivello

Quando lo spazio logico di indirizzamento di un processo è molto esteso si ha un elevato numero di pagine e la tabella delle pagine del processo assume grandi dimensioni, che per sua natura deve essere allocato in modo contiguo (generando gli stessi problemi dell'allocazione contigua). Per risolvere questo problema si può utilizzare la **paginazione a più livelli**, che consiste nell'allocazione non contigua anche della tabella delle pagine, ossia si applica la paginazione alla tabella delle pagine. In questo modo è possibile indirizzare spazi logici di dimensioni elevate (N.B. non aumento la capacità di indirizzamento ma cerco di risolvere i problemi dovuti all'allocazione contigua), permettendo di fatto di mantenere in memoria soltanto le tabelle interne (secondo livello) che servono e non l'intera tabella.
Lo svantaggio principale di questa tecnica, è l'aumento del tempo di accesso dovuto all'aumento di indirettezza di accesso alla memoria. L'indirizzo logico passa dall'essere composto dalla pagina p e dall'offset d, ad essere composto dalla terna P1, P2, d:

- P1 è il primo livello, che è l'indice di accesso alla **tabella esterna**, la quale cella contiene l'indirizzo della pagina della tabella delle pagine del processo.
- P2 indica invece l'offset della pagina(**tabella interna**) puntata da P1, la quale cella conterrà il riferimento alla pagina dello spazio di indirizzamento del processo.
- d rappresenta l'offset della pagina dello spazio di indirizzamento del processo.

Per rappresentare il calcolo si potrebbe pensare di procedere con il seguente calcolo: frame = M[tabellaEsterna[P1]+P2], valore = M[frame+d].



Per quanto riguarda lo spazio di indirizzamento, è chiaro che aumentando i livelli lo spazio di indirizzamento rimane invariato. Quello che varia è come la memoria della tabella delle pagine viene organizzata. Prendendo il caso con solamente un livello, con 32 bit di indirizzo, 12 bit sono utilizzati dall'offset, mentre i restanti 20 vengono utilizzati per indicizzare le pagine all'interno dello spazio di indirizzamento. In questo caso la tabella potrebbe arrivare fino ad un massimo 1 MB. Nel caso invece di paginazione a due livelli, è possibile ottenere 1K tabelle di secondo livello a loro volta composta di 1K entry.

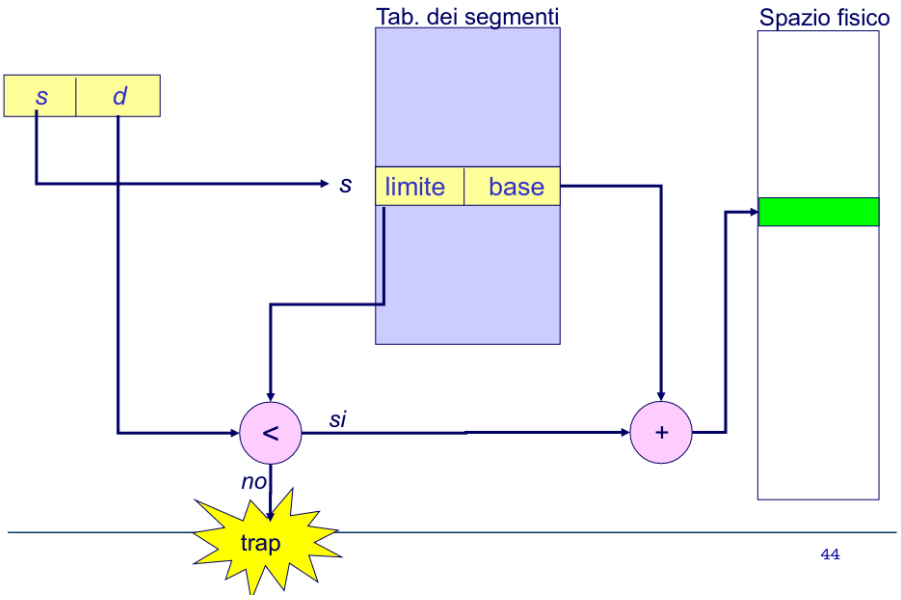
9.1.5 Tabella delle pagine invertita

Per limitare l'occupazione della memoria e diminuire l'overhead introdotto dal cambio di contesto (il contenuto della cache non è più utile e va sovrascritto) in alcuni SO si usa un'unica struttura dati globale: **la tabella delle pagine invertita**, la quale contiene un elemento per ogni frame e non per ogni pagina logica. Ogni elemento rappresenta quindi un frame, indice della tabella, e se è allocato contiene: il pid che identifica il processo a cui è assegnato il frame, il p che identifica il numero di pagina logica e un d che è l'offset all'interno della pagina. Preso un indirizzo logico si cerca nella tabella l'elemento che contiene la coppia (pid,p) e l'indice dell'elemento trovato rappresenta il numero del frame allocato alla pagina logica p. Il tempo di ricerca è più alto (da un accesso diretto ad un vettore si passa ad un accesso per ricerca) e c'è un problema di condivisione

di codice tra processi (*rientranza*) in quanto è difficile associare un frame a più pagine logiche di processi diversi.
Il vantaggio maggiormente apprezzabile è la diminuzione dell'overhead.

9.1.6 Segmentazione

La segmentazione si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti(segmenti), caratterizzate da un nome e lunghezza variabile. Essi sono divisi secondo un criterio semantico dello spazio di indirizzamento (es: codice-dati-stack-heap), ogni segmento viene allocato in memoria in modo contiguo senza un ordine preciso e ad ognuno di essi il SO associa un intero attraverso il quale lo si può riferire. Ad ogni segmento possono essere associati diritti di accesso, per esempio *read-only*. Nella segmentazione ogni indirizzo logico ha una struttura del tipo: <**segmento−offset**>, dove il segmento è il numero rappresentativo del segmento nel sistema. Per effettuare il binding tra segmenti e indirizzi fisici, è presente una **tabella dei segmenti** che ha una entry per ogni segmento, descritto dalla coppia : <**base(registro STBR)−indice(registro STLB)**>. La base indica l'indirizzo fisico della prima cella del segmento, mentre il limite indica la dimensione del segmento.



La tabella dei segmenti con implementazione globale può avere dimensioni elevate siccome ogni processo necessita di una tabella. Questa, in base alle caratteristiche, può essere realizzata su registri di CPU, in memoria centrale oppure su cache.

La segmentazione è l'evoluzione della tecnica di allocazione a partizioni variabili, la quale prevede solamente un segmento per processo. Il problema principale di questa soluzione è la frammentazione esterna, che viene risolta con l'allocazione dei segmenti con politiche adatte alla situazione (best fit, worst fit, ..). La segmentazione e la paginazione possono essere unite nella **segmentazione paginata**, in cui lo spazio logico è segmentato e ogni segmento è suddiviso in pagine, con questo metodo si elimina la frammentazione esterna e non è necessario mantenere nella cache/memoria l'intero segmento, ma basta caricare solo le pagine necessarie. Le strutture presenti nel sistema saranno, per ogni processo, una tabella dei segmenti del processo e una tabella delle pagine per ogni segmento. Il principale vantaggio introdotto della segmentazione è una maggiore gestione della protezione della memoria e una suddivisione strutturata dello spazio di indirizzamento del codice.
Linux utilizza una gestione della memoria basata su segmentazione paginata a tre livelli.

9.2 Memoria virtuale

La dimensione della memoria può rappresentare un vincolo importante riguardo alla dimensione dei processi e al grado di multiprogrammazione. Si può quindi volere un sistema di gestione della memoria che consenta la presenza di più processi in memoria, indipendentemente dalla dimen-

sione dello spazio disponibile e una memoria che svincoli il grado di multiprogrammazione dalla dimensione effettiva della memoria.

9.2.1 Overlay

Mantenere in memoria solamente il codice e i dati che sono necessari al momento, oppure quelli che vengono utilizzati più frequentemente. Sta al programmatore suddividere codice e dati. Conoscere il funzionamento del processo per suddividere il codice in più parti da caricare in memoria. Non è una soluzione scalabile, si preferisce un metodo gestito interamente dal sistema operativo e non processo-dipendente.

9.2.2 Gestione della memoria virtuale

Di solito la memoria virtuale è realizzata mediante tecniche di paginazione su richiesta in cui tutte le pagine di ogni processo risiedono in memoria di massa (**backing store**) e durante l'esecuzione alcune di esse vengono trasferite all'occorrenza in memoria centrale, è infatti presente un modulo del SO chiamato **pager** che realizza i trasferimenti delle pagine (*swapper di pagine*). Esistono alcune implementazioni del pager,ccome il **pager lazy**. Questo trasferisce in memoria centrale una pagina alla volta, su richiesta. Durante la vita di un processo, può essere necessario lo swap-in del processo. In questo caso è necessaria la presenza di uno swapper affiancato al pager che gestisca i trasferimenti di interi processi. Il pager, prima di eseguire lo swap-in di un processo, può stimare le pagine di cui il processo avrà bisogno nella fase di caricamento.
Si noti che lo swapper lavora ad una frequenza molto minore di quella del pager. Inoltre il lo swapper non fa riferimento al backing store, ma fa riferimento ad un'altra area dedicata.

Con la memoria virtuale, una pagina dello spazio logico di un processo può quindi essere allocata in memoria centrale o secondaria e lo si distingue dai bit di validità presenti nella tabella delle pagine: il bit settato a zero significa che la pagina è in memoria secondaria oppure è invalida. Il processo nel momento della traduzione da indirizzo logico a fisico, consulta la tabella delle pagine del processo. Se il bit di validità della pagina corrispondente vale 0 viene inviato un'interruzione al SO **page fault**. Ricordati che la TP ha dimensione fissa e non tutte le pagine sono valide, per questo il processore deve decidere se la causa dell'interrupt è un accesso illegale oppure il dato si trova in memoria di massa.
Al momento dell'interruzione il processore:

1. Salvataggio del contesto di esecuzione del processo(resgistri, stato, tabella delle pagine).
2. Verifica del motivo del page fault: mediante una tabella interna al kernel si verifica se il processo ha violato i vincoli di protezione con un riferimento illegale terminando il processo, oppure se la pagina è in memoria secondaria (riferimento legale).
3. Copia della pagina in un frame libero.
4. Aggiornamento della tabella delle pagine.
5. Ripristino del processo ed esecuzione dell'istruzione interrotta.

In questo modo viene garantita la trasparenza al processo. Questo non sa nulla della memoria virtuale, è il SO che maschera il tutto.

In seguito a un page fault se è necessario caricare una pagina in memoria centrale potrebbero non esserci frame liberi. In questo caso si utilizza la **sovrallocazione**: viene sostituita una pagina vittima Pvitt allocata in memoria con la pagina da caricare Pnew. Per prima cosa si individua la Pvitt, la si salva su disco, si carica la Pnew nel frame liberato, si aggiornano le tabelle e si riprende il processo. Durante questo procedimento, la sostituzione di una pagina può richiedere 2 trasferimenti da/verso il disco (scaricare la vittima e caricare la pagina nuova).
E' possibile che la pagina vittima in memoria centrale non sia stata ancora modificata dalla sua copia presente su disco (es: pagine di codice read-only) e in quel caso si può semplicemente cancellare la vittima dalla memoria centrale senza copiarla sul disco, in quanto uguale. Viene così inserito in ogni elemento della tabella delle pagine un bit di modifica (**dirty bit**). Se è settato a 1 la pagina ha subito almeno un aggiornamento da quando è stata caricata in memoria, se invece è settato a 0 significa che la pagina non è stata modificata. L'algoritmo di sostituzione esamina il bit di modifica della vittima ed esegue swap-out della vittima solo se il dirty-bit è settato, altrimenti si limita a cancellarla.

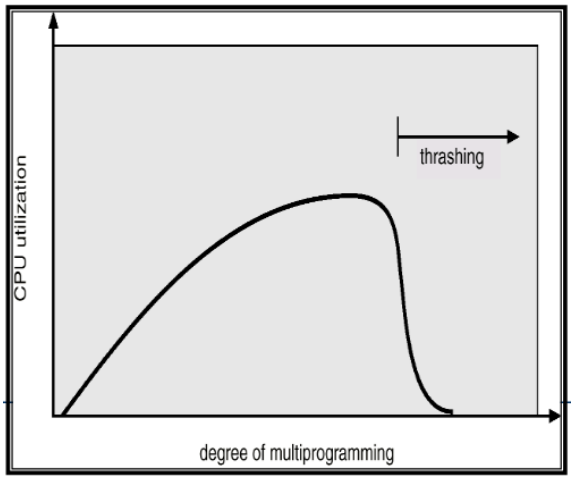
9.2.3 Algoritmi di sostituzione

Il fine di ogni algoritmo di sostituzione è di sostituire quelle pagine la cui probabilità di accesso a breve termine è bassa.

1. **LFU (Least Frequently Used)**: viene sostituita la pagina che è stata usata meno frequentemente (in un intervallo di tempo prefissato); in questo caso è necessario associare un contatore d'accessi ad ogni pagina. Molti più accessi in memoria, necessita di un algoritmo di ricerca. Frequenza di utilizzo
2. **FIFO** viene sostituita la pagina che è da più tempo caricata in memoria (indipendentemente dal suo uso); in questo caso è necessario memorizzare la cronologia dei caricamenti in memoria (attraverso timestamp). Cronologia di caricamento
3. **LRU (Least Recently Used)** viene sostituita la pagina che è stata usata meno recentemente; in questo caso è necessario registrare la sequenza degli accessi alle pagine in memoria. Anche questo si può realizzare con il timestamp, dove l'elemento della tabella delle pagine contiene un campo che rappresenta l'istante dell'ultimo accesso alla pagina (costo di ricerca della pagina vittima) o con uno stack dove ogni elemento rappresenta una pagina e l'accesso a una pagina provoca lo spostamento dell'elemento corrispondente al top dello stack (verso il fondo stanno quelle accedute meno recentemente). Il pager selezionerà quindi le pagine nella parte più bassa dello stack. Spesso vengono usate versioni semplificate di LRU, inserendo un bit di uso associato alla pagina che al momento del caricamento è inizializzato a 0 e quando la pagina viene acceduta, viene settato. Periodicamente i bit di uso vengono resettati. Verrà sostituita una pagina avente bit di uso uguale a 0. Il criterio potrebbe inoltre considerare il dirty bit: se infatti ci fossero più pagine non usate di recente (cioè con bit di uso uguale a 0), ne verrebbe scelta una non aggiornata (cioè con dirty bit uguale a 0). Cronologia di utilizzo

9.2.4 Working set

Al crescere del numero dei processi e con la saturazione della memoria, si può raggiungere un calo significativo dell'utilizzo della CPU per effettuare il paging: il processo impiegherà più tempo per la paginazione che per l'esecuzione: in questo caso si parla di **trashing**. Per contrastarlo si usano tecniche di gestione della memoria che si basano su **pre-paginazione**, cioè a differenza del paging-lazy, non si aspetta che la memoria sia satura per intervenire. Si cerca quindi di prevedere il set di pagine di cui il processo da caricare ha bisogno per la prossima fase di esecuzione: il **working set**. Il working set può essere individuato in base a criteri di località temporale (la tecnica del working set non lavora con il principio di località spaziale).



Un processo in una certa fase di esecuzione usa solo un sottoinsieme relativamente piccolo delle sue pagine logiche, che varia lentamente nel tempo. È su questo che si basa la tecnica del working set.

1. **Località spaziale**: alta probabilità di accedere a locazioni vicine nello spazio logico/virtuale) a locazioni appena accedute.
2. **Località temporale**: alta probabilità di accesso a locazioni accedute di recente.

Dato un intero Δ, il working set di un processo P nell'istante t è l'insieme di pagine $\Delta(t)$ indirizzate da P nei più recenti Δ riferimenti, delta definisce la **finestra** del working set. Δ caratterizza il working set, esprimendo l'estensione della finestra dei riferimenti. Se Δ è piccolo il working set è insufficiente a garantire località (alto numero di page fault). Se Δ è grande: utilizzo di memorio per l'allocazione di pagine non necessarie. Ad ogni istante, data la dimensione corrente del working set WSS_i di ogni processo P_i, si può individuare D=$\sum_{n=1}^{\infty} WSS_i$ cioè la richiesta del numero totale dei **frame**. Se m è il numero totale di frame liberi può esserci spazio per l'allocazione di nuovi processi se D<m o swapping di uno (o più) processi se D>m.

Il caricamento in memoria di un processo, consiste nel caricamento di un working set iniziale. Il SO mantiene in memoria solamente il working set di ogni processo aggiornandolo dinamicamente, in base al principio di località temporale: all'istante t vengono mantenute le pagine usate dal processo nell'ultima finestra $\Delta(t)$. Tutte le altre pagine possono essere sostituite.

9.3 Gestione della memoria in UNIX

9.3.1 Versioni iniziali

In generale i sistemi operativi si dividono in sistemi che utilizzano working set e sistemi che non lo utilizzano. Ad esempio UNIX non lo utilizza. Nelle prime versioni di UNIX, la memoria era puramente segmentata e non era presente memoria virtuale. Così, per ridurre l'overhead introdotto dallo swapper, si è ricorso all'utilizzo del codice rientrante (fork). Si è scelta la tecnica di allocazione contigua, con tecnica di *first fit*, sia in memoria centrale che in memoria secondaria. (con tutti i problemi del caso)

In assenza di memoria virtuale, swapper ricopre un ruolo chiave per la gestione delle contese di memoria da parte dei diversi processi (prime versioni di UNIX). Infatti lo swapper periodicamente viene attivato per provvedere eventualmente a swap-in (processi piccoli e processi da più tempo swapped) e swap-out (processi inattivi, processi da più tempo in memoria) di processi. Lo swapper era attivato con una cadenza dell'ordine dei 4 secondi.

9.3.2 Nuove versioni

Le maggiori differenze sono l'introduzione della memoria virtuale e la tecnica di allocazione è a segmentazione paginata. L'allocazione di ogni segmento non è più contiguo.

UNIX non utilizza la tecnica del working set ma utilizza altre tecniche di *pre-paginazione* per evitare il trashing. UNIX carica nei frame liberi della memoria pagine che non sono strettamente necessarie, settando il bit di validità a 0. In caso di page fault se il frame si trova già in memoria basta modificare la tabella delle pagine (in particolare il bit di validità) e la lista dei frame liberi. Lo stato di allocazione dei frame viene mantenuta in una struttura dati interna al kernel che viene consultata in caso di page fault. Questa struttura dati prende il nome di **core map**. Quindi ora uno dei possibili motivi per cui il bit di validità della pagina è uguale a zero può essere il fatto che la pagina è stata pre-caricata(vedi gestione del page fault).
Nota che di base il pager è lazy, con l'aggiunta di un sistema di pre-paginazione onde evitare trashing.

L'algoritmo di sostituzione in caso di sovrallocazione è un **LRU** modificato, spesso chiamato con il nome di **algortimo di seconda chance**. Ad ogni pagina viene associato un bit di uso:

1. Il bit viene settato a 0 all'inizio del caricamento. Se acceduta viene settata a 1.
2. Al momento del page fault, il pager setta a 0 tutte le pagine con bit d'uso a 1 (quindi viene resettato). Nel caso in cui una pagina abbia bit d'uso pari a 0, viene spostata in memoria centrale.
3. La pagine vittima viene resa invalida ponendo salvando il frame vittima nella tabella dei frame liberi. Inoltre in caso di presenza del dirty bit, la pagina viene copiata in memoria solo se il dirty bit è settato a 1. In caso invece non sia presente, la pagina viene salvata in memoria di default.

L'algoritmo di sostituzione, paging, viene eseguito sempre dal processo **pagedeamon (pid=2)**.

Altre tecniche per affrontare il trashing si basano sulla definizione di tre costanti: $lotsfree < desfree < minfree$

1. **lotsfree**:numero minimo di frame liberi per evitare sostituzione di pagine.
2. **minfree**: numero minimo di frame liberi necessari per evitare swapping dei processi.
3. **desfree**: numero desiderato di frame liberi.

l'idea è la stessa di un algoritmo di pre-paginazione: prevenire la saturazione della memoria con anticipo. Mantenendo monitorato il numero di frame liberi, se questo valore raggiunge il valore di lotsfree, il pager si attiva con anticipo evitando quindi il trashing. Il sistema di paging però può andare in sovraccarico se:

- Il numero di frame liberi è minore di minfree.
- Il numero medio di frame liberi nell'unità di tempo è minore di desfree.

Onde evitare che il pagedeamon monopolizzi la CPU, lo scheduler attiva lo swapper quando il pager è in sovraccarico oppure sta usando la CPU per più del 10% del tempo.

La paginazione nei sistemi GNU/Linux avviene a due o tre livelli in base al processore. I blocchi di codice sono caricati in modo dinamico e i moduli caricati devono essere ad allocazione contigua. Nei sistemi GNU/Linux la memoria viene suddivisa in più aree:

1. **Area codice kernel**: zona di memoria dedicata al kernel, le pagine di quest'area è detta **locked** cioè sono esclusi dai normali meccanismi di sostituzione e paginazione.
2. **Kernel cache**: heap del kernel, **locked**.
3. **Area moduli gestiti dinamicamente**: allocazione mediante algoritmo **buddy list** (*allocazione contigua dei singoli moduli*)
4. **Buffer cache**: gestione I/O su dispositivi a blocchi. Per velocizzare gli accessi in memoria.
5. **Inode cache**: copia degli inode utilizzati recentemente (vedi tabella file attivi).
6. **Page cache**: pagine non più utilizzate in attesa di sostituzione.
7. Processi utenti...

Ogni processo può utilizzare fino a 4 GB in memoria: 1GB riservato al kernel e i restanti possono essere utilizzati come spazio di indirizzamento virtuale. Lo spazio di indirizzamento di ogni processo può essere suddiviso in un insieme di regioni omogenee e contigue, dov ogni regione è costituita da una sequenza di pagine accomunate dalle stesse caratteristiche di protezione e di paginazione.

9.3.3 MS Windows XP

Il sistema windows gestisce la memoria con il **clustering delle pagine**: in caso di page fault, viene caricato tutto un gruppo di pagine attorno a quella mancante (page cluster). Ogni processo ha un working set minimo (numero minimo di pagine sicuramente mantenute in memoria) e un working set massimo (massimo numero di pagine mantenibile in memoria). Qualora la memoria fisica libera scenda sotto una soglia, SO automaticamente ristabilisce la quota desiderata di frame liberi (**working set trimming**), eliminando pagine appartenenti a processi che ne hanno in eccesso rispetto a working set minimo. Quindi in parole povere, per ogni processo caricato in memoria, in caso di saturazione della memoria tutti i processi che hanno delle pagine in più rispetto al working set minimo vengono eliminate.

10 Programmazione concorrente nel Modello a Memoria Comune

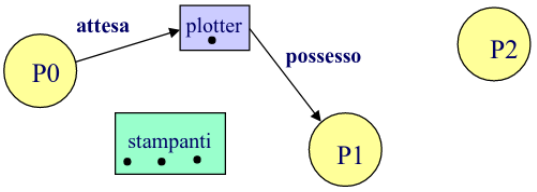
Se la macchina concorrente è organizzata secondo il modello ad ambiente globale (o modello a memoria comune) il processo viene sostituito con il **thread**. Risorse e dati vengono messe in condivisione e si introduce la necessità di sincronizzare gli accessi alle risorse condivise. Il linguaggio di programmazione concorrente offre costrutti per esprimere la soluzione a problemi di sincronizzazione mentre è il nucleo della macchina concorrente (può essere il SO oppure JVM) che realizza i meccanismi di sincronizzazione.

Ogni applicazione concorrente può essere rappresentata da un insieme di componenti, suddiviso in due sottoinsiemi disgiunti: componenti **attivi** (in questo modello i thread) e componenti **passivi** (le risorse).
Le risorse Le risorse sono raggruppate in **classi**. Ogni classe identifica l'insieme di tutte e sole le operazioni che un processo può eseguire per operare su risorse di quella classe. Si dice **privata di un processo P** se P è il solo processo che può eseguire operazioni sulla risorsa. Invece si dice risorsa **comune (o globale)** a più processi se è una risorsa su cui più processi possono operare. Con le risorse comuni è necessario garantire il vincolo di **mutua esclusione**.
Data un'operazione I(d), che opera su un dato d comune a più processi, essa si dice **indivisibile** (o atomica) , se, durante la sua esecuzione da parte di un processo P, il dato d non e' accessibile ad altri processi. In pratica gli stati intermedi dell'operazione non sono rilevabili dagli altri processi.
La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di **sezione critica**. Ad un insieme di variabili comuni possono essere associate una sola sezione critica (usata da tutti i processi) o più sezioni critiche (**classe di sezioni critiche**). Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo. Ciò significa Una sola sezione critica di una classe può essere in esecuzione ad ogni istante.

10.1 Deadlock

Un insieme di processi è in **deadlock** se ogni processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo dell'insieme. Per descrivere il deadlock, si utilizzano i grafi di allocazione delle risorse:

- I processi sono cerchi.
- Le risorse sono dei rettangoli.
- Se gli archi vanno dal processo alla risorsa, il processo è in attesa della risorsa. Al contrario il processo possiede la risorsa.



Si noti che un processo va avanti solo se possiede tutte le risorse di cui necessita.

Le condizioni necessarie e sufficienti affinché si verifichi il deadlock sono:

1. **MUTUA ESCLUSIONE**:le risorse sono utilizzate in modo mutuamente esclusivo.
2. **POSSESSO E ATTESA**: ogni processo che possiede una risorsa può richiederne un'altra.
3. **IMPOSSIBILITA' DI PREEMPTION**: una volta assegnata ad un processo, una risorsa non può essere sottratta al processo (no preemption).
4. **ATTESA CIRCOLARE**: esiste un gruppo di processi P0, P1, ... , PN in cui P0 attende una risorsa posseduta da P1, P1 attende una risorsa posseduta da P2, ... e PN attende una risorsa posseduta da P0.

Se almeno UNA delle QUATTRO condizioni non è verificata NON C'E' DEADLOCK. Se voglio sviluppare un'applicazione concorrente, devo fare in modo che almeno una delle quattro condizioni non sia verificata.
Le situazioni di deadlock devono essere evitate o prevenendo il blocco critico oppure rilevandolo:

- **PREVENZIONE**: può essere STATICA (si impongono dei vincoli a priori che evitano almeno una delle 4 situazioni sopra citate) o DINAMICA (prima che un processo P richieda la risorsa R, si controlla se c'è la possibilità di deadlock).
- **RILEVAZIONE / RIPRISTINO DEL DEADLOCK**: non c'è prevenzione, ma il S.O. utilizza algoritmi di rilevazione della presenza del deadlock se già in atto e attua un algoritmo di ripristino che può avvenire con la terminazione o con la prelazione del processo. (Soluzione molto costosa e inefficiente)

Per quanto riguarda la prevenzione statica è necessario eliminare a livello di progetto, almeno una delle condizioni necessarie affinché si verifichi deadlock.

1. Si utilizzano risorse condivisibili (in certi casi non è possibile e anche in questo caso non elimina il deadlock perché dipende dalla molteplicità della risorsa);
2. Si impedisce ad ogni processo di possedere una risorsa mentre ne richiede un'altra; Impedire ad ogni processo di richiedere più di una risorsa alla volta.
3. Possibilità di sottrarre la risorsa al processo (in certi casi non è possibile);
4. Si stabilisce un rigido ordinamento nell'acquisizione delle risorse da parte di ogni processo. (per esempio un processo non può mai acquisire una risorsa Ri se è già in possesso di una risorsa Rj con j¿i, la soluzione che evita l'attesa circolare è quella in cui i processi che effettuano le acquisizioni delle risorse rispettano lo stesso ordine). (domanda: questa soluzione elimina veramente il deadlock in qualunque caso?)

Nota bene questo tipo di prevenzione, non è sempre facile da applicare: bisogna avere un certo controllo sulla struttura del codice, cosa non sempre possibile per esempio nei sistemi distribuiti.

La prevenzione dinamica invece si basa sulla prevenzione del deadlock in fase di allocazione. Un esempio di algoritmo di prevenzione dinamica potrebbe essere l'algoritmo del banchiere. I processi sono i clienti che richiedono risorse, cioè soldi.La banca non può permettere contemporaneamente a tutti i clienti di ottenere il credito massimo: il sistema non potrebbe disporre di risorse a sufficienza. L'obiettivo è quindi l'individuazione di una sequenza di processi **salva**, tra tutte le possibili sequenze di esecuzione dei processi. Def: una sequenza di processi P0, P1, .. Pn è salva se per ogni processo Pi le richieste di risorse che Pi può ancora effettuare possono essere soddisfatte con le risorse attualmente libere, più le risorse allocate a tutti i processi Pj (j¡i)

10.2 Soluzioni della mutua esclusione

Il problema della mutua esclusione nasce quando più di un processo alla volta può avere accesso a variabili / risorse comuni. La regola di mutua esclusione stabilisce che una sola sezione critica di una classe (insieme di sezioni critiche) può essere in esecuzione ad ogni istante.

Un algoritmo risolve il problema della mutua esclusione se:

1. Sezioni critiche della stessa classe eseguite in modo esclusivo;
2. Quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione ad altri processi;
3. Assenza di deadlock

Lo schema generale di risoluzione può essere schematizzato in:
$< PROLOGO >$ SEZIONE CRITICA $< EPILOGO >$

1. **PROLOGO**: ogni processo prima di entrare in una sezione critica deve chiedere l'autorizzazione che gli garantiscono l'uso ESCLUSIVO della risorsa, se questa è libera, oppure ne impediscono l'accesso se questa è già occupata.
2. **EPILOGO**: al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la sezione critica.

In generale è possibile realizzare questo schema risolutivo con l'utilizzo di tre tipi di tecniche:

- **Software**:la soluzione non richiede particolari meccanismi di sincronizzazione ma sfrutta solo la possibilità di condivisione di variabili (es. algoritmo di Dekker). NB sono meccanismi implementati a livello di programmi utente, non viene fatto uso dei meccanismi messi a disposizione dal kernel.
- **Hardware**: Il supporto è fornito direttamente all'architettura HW: che può essere disabilitare e abilitare gli int.
- **SO** - Soluzioni basate su strumenti di sincronizzazione: prologo ed epilogo sfruttano strumenti software per la sincronizzazione realizzati dal kernel del sistema operativo.

10.2.1 Soluzioni software

Non presuppongono alcun supporto particolare alla sincronizzazione tra i processi né da parte del sistema operativo, né da parte dell'HW. Assunzione di base: l'hardware garantisce la mutua esclusione solo a livello di lettura e scrittura di una singola parola di memoria.
Il problema che si presenta spesso nella maggioranza degli algoritmi di risoluzione è la mancata gestione di una possibile pre-emption oppure il mancato rispetto del secondo vincolo: Quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione ad altri processi. Questo avviene se l'esecuzione dei due processi è alternata da un variabile turno la quale sancisce un'alternanza tra le due variabili obbligatoria.

L'algoritmo di Dekker risolve i problemi elencati. Per due processi richiede tre variabili condivise: 2 flag e una variabile turno. Per ciascun processo esiste esattamente un flag. Un flag impostato (flag = true) segnala che il processo corrispondente potrebbe trovarsi in esecuzione della sezione critica. La variabile turno funziona come una specie di segnalino di turno. La condizione d'ingresso per la iterazione è il flag dell'altro processo: se è impostato, allora l'altro processo si trova in esecuzione della sezione critica, oppure della propria iterazione. in quest'ultimo caso è lo stato di turno che stabilisce l'ulteriore procedere. Se turno contiene il numero dell'altro processo, il flag viene c cancellato e l'esecuzione riprende da principio. In questo modo, l'altro processo ottiene la possibilità di abbandonare l'iterazione (in caso vi ci si trovava) e di accedere alla sezione critica. Dopo la sezione critica il flag viene cancellato.
Nota bene: la variabile turno non sancisce l'alternarsi obbligatorio dei processi, ma viene utilizzata solamente nel caso in cui entrambi stiano eseguendo la sezione critica! Quindi il secondo vincolo è rispettato.

```
1    /* processo P1: */
2    main(){
3         ...
4         busy1=1;
5         while (busy2==1)
6         if (turno==2){
7              busy1=0;
8              while(turno!=1);
9              busy1=1;
10        }
11        <sezione critica A>;
12        turno=2;
13        busy1=0;
14        ...
15   }
16   /* processo P2: */
17   main(){
18        ...
19        busy2=1;
20        while (busy1==1)
21        if (turno==1){
22             busy2=0;
23             while(turno!=2);
24             busy2=1;
25        }
26        <sezione critica B>;
27        turno=1;
28        busy2=0;
29        ...
30   }
```

La soluzione potrebbe risultare complicata da estendere per N processi. Per questo si potrebbe utilizzare **l'algoritmo di Peterson**, sicuramente più compatto dell'algoritmo di Dekker. Il funzionamento è simile a quello di prima. Se il processo sta per entrare in una sezione critica imposta un flag e setta turno in modo da andare in stato di wait. Solo che essendo atomica la fase di scrittura di turno, l'ultimo dei due processi che setta turno andà in stato di wait.

```
1    /* processo P1: */
2    main(){
3         ...
4         busy1=1;
5         turno=2;
6         while(busy2 && turno==2);
7         <sezione critica A>;
8         busy1=0;
9         ...
```

```
10      }
11
12      /* processo P2: */
13      main(){
14          ...
15          busy2=1;
16          turno=1;
17          while(busy1 && turno==1);
18          <sezione critica B>;
19          busy2=0;
20          ...
21      }
```

In queste soluzioni è possibile evidenziare un problema ricorrente: **l'attesa attiva o busy waiting**. Un processo in attesa di poter accedere alla risorsa, consuma tempo di CPU. È possibile quindi aggiungere dei requisiti all'algoritmo che realizza la mutua esclusione: Ai requisiti sopra citati aggiungiamo quindi:

4. Assenza di starvation, il processo deve poter accedere a una risorsa in un tempo finito.
5. Eliminare il busy waiting, ossia sospendere l'esecuzione di un processo per tutto il tempo in cui non può avere accesso alla sezione critica.

10.2.2 Soluzioni hardware

La più semplice soluzione hardware prevede nel disabilitare le interruzioni nell'epilogo e abilitarle nel prologo. La soluzione però non è adatta a sistemi con più processori. Inoltre rende insensibile il sistema ad ogni stimolo esterno per tutta la durata di qualunque sezione critica.

Nella maggior parte dei sistemi, il processore non solo permette di leggere e scrivere in modo atomico, ma permette di eseguire un'istruzione atomica di lettura e scrittura contemporanea della memoria: **test_and_set**.

```
1 int test-and-set(int *a){ //passo un indirizzo di memoria
2     int R;
3     R=*a; // leggo l'indirizzo di memoria
4     *a=0; // setto a zero l'indirizzo di memoria
5     return R;
6 }
```

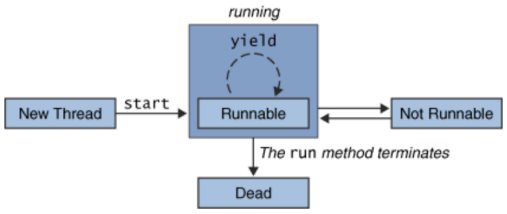
Tramite questa istruzione è possibile realizzare un meccanismo hardware-based (lock/unlock) per la soluzione del problema di mutua esclusione. Si prende X una variabile associata ad una classi di sezioni critiche inizializzata al valore uno (1-libera, 0-occupata).
Definiamo su x le seguenti operazioni **ATOMICHE**:

```
1 void lock(int *x){
2     while (!test-and-set(x)); // se la risorsa e occupata, quindi x=0, setta di
      nuovo a zero e non succede nulla, se e libera setta a zero e la occupa
3 }
4 void unlock(int *x){
5     *x=1;
6 }
```

Si noti che è presente della busy waiting e inoltre è possibile incorrere in starvation, senza un'opportuna gestione delle risorse.

11 JAVA thread

THREAD: *singolo flusso sequenziale di controllo all'interno di un processo*. Condivide *codice, dati e spazio di indirizzamento* con gli altri thread associati. Nota però che *stack e program counter* devono essere privati. Ad ogni programma Java corrisponde l'esecuzione di un task (processo), contenente almeno un singolo thread, corrispondente al metodo main() sulla JVM (java virtual machine). (Ricorda che le variabili statiche sono condivise in quanto globali)



Uno thread che viene portato in stato runnable, perché è stato invocato il metodo start per esempio, non sta necessariamente utilizzando tempo di CPU. Esso è in stato di *ready* ed è pronto ad essere schedulato dallo scheduler della JVM. Un thread in esecuzione può essere forzato ad essere deschedulato con il metodo **yield**, il quale nelle vecchie versioni di Java veniva utilizzato per perettere l'alternarsi di thread con uguale priorità. Ora yield è deprecato e i meccanismi di alternanza tra thread con ugual priorità sono gestiti a livello di JVM.
Lo stato not runnable è simile allo stato di wait nei sistemi a memoria locale. Lo stato di dead può essere raggiunto perché un altro thread ha invocato il metodo stop su di lui, oppure per "morte naturale".

Ogni thread ha un ID assegnato dalla JVM che è ottenibile tramite getName().

11.1 Sincronizzazione

Ogni tipo di interazione tra thread avviene tramite oggetti comuni:

1. Interazione di tipo **competitivo** (mutua esclusione): meccanismo di **objects locks** e costrutto **synchronized**.
2. Interazione di tipo **cooperativo**: meccanismo di **wait-notify** (semafori).

11.1.1 Mutua esclusione

Ad ogni oggetto viene associato un oggetto **object lock** che rappresenta lo stato dell'oggetto (libero/occupato). L'associazione del lock ad ogni oggetto viene fatta in modo automatico dalla JVM. I tipi primitivi ovviamente non essendo oggetti devono essere utilizzati attraverso le corrette classi *wrapper* per poter utilizzare i lock.
Si dice che ogni oggetto possiede un **monitor**, cioè un meccanismo di gestione della mutua esclusione che non si limita solamente all'utilizzo del lock. Il monitor, oltre a garantire il rispetto della mutua esclusione sull'oggetto/porzione di codice, permette la gestione per ogni oggetto di un **entry set**, in cui i thread posti in stato di wait vengono gestiti attraverso opportune politiche di assegnamento della risorsa dalla JVM. Il lock non garantisce una simile gestione.
Esiste anche la possibilità di implementare l'interfaccia Lock, la quale però garantisce solamente il vincolo di mutua esclusione senza garantire alcuna sicurezza sulla gestione della coda di wait. (per maggiori informazioni consultare https://www.geeksforgeeks.org/difference-between-lock-and-monitor-in-java-concurrency/)

Nella pratica il costrutto synchronized viene sostituito dal compilatore con:

- Un prologo che prova ad acquisire la risorsa. Se fallisce il thread viene inserito nel entry set dell'oggetto soggetto della synchronized.
- Un epilogo che tenta di liberare la risorsa: la JVM sceglie il prossimo thread dal entry set da notificare lasciando la risorsa occupata, oppure in caso di entry set vuoto libera la risorsa.

NOTA BENE: Un metodo di una classe synchronized, se invocato, comporta l'entrata del thread nel monitor dell'oggetto! Quindi tutti i thread che vorranno utilizzare metodi synchronized

dell'oggetto saranno messi in stato di wait. Questo perché per utilizzare un meotodo synchronized di un oggetto è necessario entrare nel monitor dell'oggetto, che è soltanto uno. Tutti gli altri metodi continueranno ad essere richiamabili.
È bene porre il synchronized sui metodi che cambiano lo stato interno di un oggetto. Ricorda che l'obiettivo primario è serializzare le operazioni.

Qual è la tipica situazione di utilizzo della notazione a blocco di synchronized? Non sempre è possibile accedere al codice delle classi e non tutte sono state pensate per un utilizzo in ambiente concorrente. Per questo è possibile utilizzare sincronizzarne l'utilizzo con la notazione a blocco.
Quale tra i due metodi è meglio utilizzare in generale? Implementare runnable oppure estendere Thread? In generale si tende ad estendere Thread solamente nel caso in cui si voglia aggiungere delle funzionalità alla classe Thread, sovrascrivendo altri metodi rispetto a "run". Quindi solitamente si preferisce estendere runnable.

11.1.2 Semafori

Un semaforo è uno strumento di sincronizzazione che consente di risolvere qualunque problema di sincronizzazione tra thread nel modello ad ambiente globale. Generalmente è realizzato dal nucleo del sistema operativo. Un semaforo è un dato astratto rappresentato da un intero NON NEGATIVO a cui è possibile accedere solo tramite le due operazioni P e V (s è il semaforo):

- **P(&s)**: ritarda il processo fino a che il valore del semaforo diventa maggiore di 0 e quindi decrementa tale valore di 1; (P in olandese sta per decremento, ricordati prelievo)
- **V(&s)**: incrementa di 1 il valore del semaforo. (V in olandese sta per incremento)

Le due operazioni sono implementate in modo **atomico**, in modo tale che siano sezioni critiche della stessa classe (????)(per creare utilizziamo LOCK / UNLOCK). Il valore del semaforo viene modificato da un solo processo alla volta. Se s=0 il semaforo è rosso, mentre se s¿0 il semaforo è verde.

L'attesa dei processi della risorsa non può essere attiva. La soluzione è quella di cambiare lo stato del processo quando la risorsa è occupata. Ad ogni semaforo, inoltre, è associata una coda **Qs** nella quale sono posti i descrittori dei processi che attendono l'autorizzazione a procedere. La politica della coda deve essere rigorosamente FIFO altrimenti è possibile incorrere nella starvation. Nella nostra applicazione saranno presenti tante code quante sono i semafori.

N.B. Quando sospendo un thread nella P perchè la risorsa è occupata, è importante che il processo rilasci la sezione critica di P, altrimenti si può incorrere nel deadlock cioè tutti i processi che vogliono entrare dentro p vengono messi in attesa.
In generale è buona norma rilasciare la sezione critica prima che un thread venga messo in sospeso, onde evitare una situazione di deadlock.

```
1  typedef struct{ int value;
2      queue Qs;
3      int lock; // inizializzato a 1
4  } semaphore;
5
6  void p(semaphore *s) {
7      lock(s->lock);
8      if(s->value==0)
9      {
10         unlock(s->lock); //uscita sez. critica
11         //<sospensione processo in s->Qs>
12         lock(s->lock); //entrata sez. critica
13     }
14     else s->value--;
15     unlock(s->lock);
16     }
17
18 void v(semaphore *s) {
19     lock(s->lock);
20     if (<s->Qs non e' vuota>)
21         // <il descrittore del primo processo viene rimosso dalla coda ed il suo
22         // stato modificato in pronto>
23     else s->value++;
24     unlock(s->lock);
25 }
```

Nota che anche se lock e unlock introducono attesa attiva, questa è limitata a pochissime righe di codice perché se nel caso peggiore il processo viene messo in attesa libera subito la sezione critica.

Quali sono gli utilizzi dei semafori? Uno e la possibilita di sincronizzare due thread (ordine di esecuzione di parti di codice). Questi tipi di messaggi vengono sono di tipo **"mutua esclusione"**. Il dubbio sorge spontaneo: qual è la differenza tra i blocchi/metodi synchronized e i semafori mutex? di base risolvono lo stesso tipo di problema, però mentre synchronized blocca tutti i metodi/blocchi dello stesso monitor, i semafori possono essere differenziati.

```
<codice di p1>
    ...
    p(%s)
    <codice da eseguire dopo>
    ...
<fine codice di p1>

<codice di p2>
    <codice da eseguire prima>
    v(%s)
    ...
<fine codice di p2>
```

Un altro utilizzo può essere lo scambio di massaggi. Per questo tipo di problemi si può utilizzare un buffer condiviso di una certa capacità N. Per la corretta alternanza di produzione e consumazione del messaggio può essere utilizzata una coppia di semafori. La soluzione deve poter bloccare il produttore in caso il buffer sia pieno e bloccare il consumatore in caso il buffer sia vuoto.
L'idea sta nel capire che in questo tipo di problema sono presenti due tipi di risorse: il buffer e i messaggi. Quando il consumatore legge un messaggio, libera il buffer e consuma messaggi. Al contrario il produttore, il quale consuma buffer e libera(crea) messaggi. Questo tipo di semafori vengono detti di **tipo risorsa**.
In generale in questo tipo di problemi si procede individuando quali sono le risorse in gioco e si associano ad essi N semafori di tipo risorsa di cardinalità C_i. Per ogni entità runnable poi, si stabilisce la relazione tra l'entità E_i e la risorsa R_i: produce o utilizza?

```
/* Processo produttore:*/
main()
{
    for (;;){
        <produzione messaggio>;
        p(&spazio_disp);
        <deposito messaggio>;
        v(&msg_disp);
    }
}

/* Processo consumatore:*/
main()
{
    for (;;){
        p(&msg_disp);
        <prelievo messaggio>;
        v(&spazio_disp);
        <consumo messaggio>;
    }
}
```

Attenzione, in caso di più produttori però è necessario garantire ulteriormente la mutua esclusione per quanto riguarda l'accesso al buffer (se due produttori scrivono sullo stesso indice è un problema). Questo si può risolvere aggiungendo un mutexP per i produttori e un mutexC per i consumatori(solo se sono più di uno).
Un altro problema si genera se per esempio nel caso siano presenti N risorse disponibili uguali(per esempio 4 stampanti). Per ogni risorsa del **pool** associo un semaforo. Il pool di risorse deve essere gestito efficientemente in modo tale che un processo P non sia autonomo nella gestione delle risorse (se due processi chiedono l'accesso alla 1 senza sapere che sia la due che la uno sono libere, almeno uno dei due si bloccherà). Si introduce quindi un **gestore delle risorse** del pool. È lui che amministra in modo efficiente le risorse del pool.

L'idea è che questa entità fornisca due metodi:

1. *unsigned int richiesta();* (ottenere, se disponibile, una qualunque risorsa del pool)

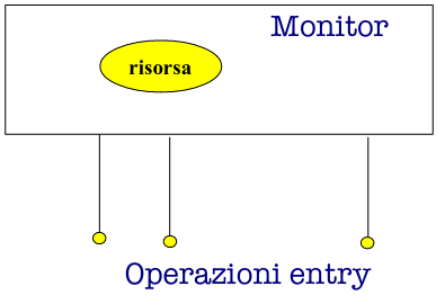
2. *void rilascio(unsigned int x);* (rilasciare la risorsa assegnata)

Avrò bisogno di un semaforo per la mutua esclusione del gestore mutex, il quale è a tutti gli effetti una risorsa che permette di aggiornare coerentemente lo stato delle risorse. Inoltre sarà presente un altro tipo di semaforo risorsa ris, di cardinalità pari alla risorsa del pool. Inoltre è presente un vettore di booleani che indica quali risorse siano libere e quali no.

```
1  int Richiesta ()
2  {
3      unsigned int x, i;
4      p(&ris);
5      p(&mutex);
6      i=0;
7      do
8      i++;
9      while (! Libero[i]);
10     x = i;
11     Libero[i] = 0;
12     v(&mutex);
13     return x;
14 }
15
16 void Rilascio (unsigned int x)
17 {
18     unsigned int i;
19     p(&mutex);
20     i=x;
21     Libero[i]= 1;
22     v(&mutex);
23     v(&ris);
24 }
```

11.1.3 Monitor

E' un costrutto sintattico che associa un insieme di operazioni (**entry**) ad una struttura dati comune a più processi, tale che le operazioni entry siano le sole operazioni permesse su quella struttura e siano mutuamente esclusive (sono le uniche operazioni che consentono di modificare le variabili locali). Possono essere definite anche delle operazioni non entry, ma che non sono accessibili dall'esterno.Scopo del monitor è controllare l'accesso alla risorsa da parte processi concorrenti, in accordo a determinate politiche. Le variabili locali definiscono lo stato della risorsa associata al monitor. L'idea è che le variabili locali definiscano lo stato della risorsa nel corso dell'esecuzione delle operazioni entry.



L'accesso avviene mediante due livelli di sincronizzazione:

1. Il primo garantisce che solo un processo alla volta possa aver acceso alle variabili comuni del monitor (realizzato direttamente dal linguaggio e dai metodi entry del monitor). I thread sospesi in attesa di *entrare nel monitor*, vengono salvati in una coda associata al monitor chiamata **entry queue**.

2. Il secondo controlla l'ordine con il quale i processi hanno accesso alla risorsa in base ad una condizione di sincronizzazione. Questa viene realizzata dal programmatore secondo una determinata politica di accesso alla risorsa.

Il secondo livello si può realizzare attarverso le variabili **conditions**. Su questo tipo di variabili sono definite due operazioni:

- **wait(condVar)**: sospende sempre il processo introducendolo nella coda individuata dalla variabile *condVar*, liberando automaticamente il monitor. Il risveglio, causato sempre da una signal sulla stessa variabile condizione, comporta la riacquisizione del monitor da parte del thread e la ripresa della normale esecuzione del programma.

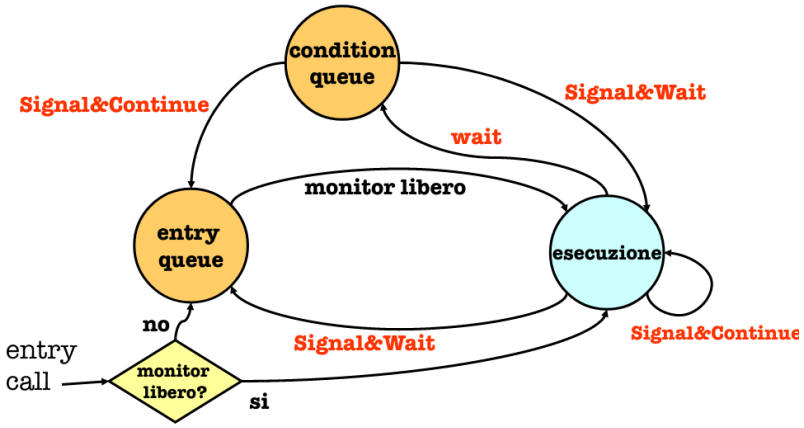
- **signal(condVar)**: riattiva un processo in attesa nella coda individuata dalla variabile cond. A differenza della *release*, la signal non è mai sospensiva

Per la gestione della signal, esistono due strategie fondamentali (il thread Q risveglia con una signal il processo P):

1. **SIGNAL_AND_WAIT**: il processo P riprende immediatamente l'esecuzione prendendo il posto di Q nel possesso del monitor, e il processo Q viene sospeso e inserito nella entry queue;

2. **SIGNAL_AND_CONTINE**: il processo Q prosegue la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver risvegliato il processo P, il quale verrà inserito nella entry queue. N.B. a differenza della soluzione signal_and_wait P deve comunque ritestare le condizioni di sincronizzazione prima di rientrare nel monitor.(nella pratica va inserito un ciclo) Questo perchè prima di P, altri thread possono accedere al monitor cambiando nuovamente lo stato della risorsa.

   Java adotta questa strategia.



E' anche possibile risvegliare tutti i processi sospesi sulla variabile condizione utilizzando la **SIGNAL_ALL** (variante della signal_and_continue). Ovviamente tutti i processi risvegliati verranno inseriti nella entry queue del monitor prima di poter proseguire con l'esecuzione.

12 Bash

La shell è un interprete comandi che permette all'utente di eseguire comandi da standart input oppure da file comandi. Esistono varie versioni di shell, la più famosa è la bourne shell presente nella maggioranza dei sistemi operativi in /bin/bas.

```
loop forever
<LOGIN>
    do{
        scanf("%s",comando);
        pid=fork();
        if (pid==0){ //shell figlio
            execlp(comando, comando, (char*)0);
            perror("...");exit(1);
        }else if(pid>0){ //shell padre
            if(<comando lanciato in foreground>){
                wait(&status); ...
            }

        }
    }while(!EOF)
<LOGOUT>
End loop
```

```
#!/bin/bash
```