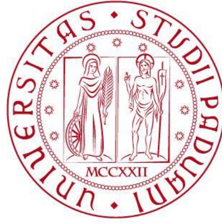


SISTEMI OPERATIVI

ARCHITETTURA E PROGRAMMAZIONE CONCORRENTE

(Versione 17/08/2017)

A cura di:
Stefano Ivancich

INDICE

1. Concorrenza nei sistemi 1

2. Sincronizzazione (Reti di petri) 3

3. Programmazione Concorrente 5

4. Il Nucleo 9

4.1. Routine di interruzione di primo intervento (First Level Interrupt Handler) 10

4.2. Il Dispatcher 10

4.3. Wait e Signal 10

4.4. Scheduling 11

5. Gestione della memoria 13

5.1. Tecniche di gestione della memoria 14

5.2. Memoria virtuale 18

5.3. Gestione della memoria virtuale 20

5.4. Multithreading 22

6. Gestione I/O 25

7. Gestione memoria secondaria 31

7.1. Schedulazione degli accessi al disco 31

7.2. Sistemi RAID 32

8. File System 35

8.1. File system UNIX 39

9. UNIX 47

10. Real Time OS 51

Note per l'esame 57

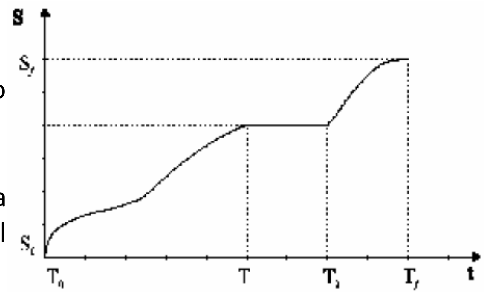
Lo scopo di questo documento è quello di riassumere i concetti fondamentali degli appunti presi durante la lezione, riscritti, corretti e completati facendo riferimento alle slide e al libro di testo "*M. Moro - Sistemi Operativi: Architettura e Programmazione concorrente. 2011. 3/e*". Non sono presenti esempi e spiegazioni dettagliate, per questi si rimanda al testo citato e alle slide.

Se trovi errori ti prego di segnalarli qui:

www.stefanoivancich.com
ivancich.stefano@gmail.com

Il documento verrà aggiornato entro 48h.

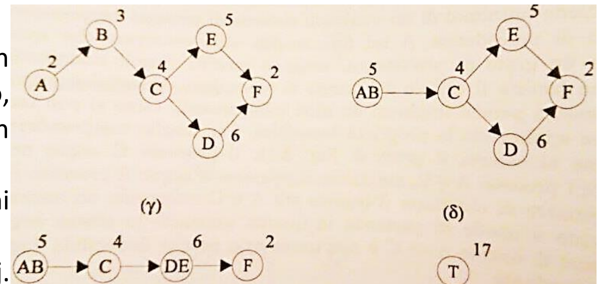
1. Concorrenza nei sistemi



Processo: una qualsiasi entità attiva che ha un suo percorso di sviluppo.
Stato di un processo: insieme dei valori di tutte le variabili di calcolo che vengono modificate durante le computazioni.
Diagramma temporale: ascisse tempo, ordinate valori degli stati.
Tempo di esecuzione: tempo finale - iniziale. Che dipende dalla complessità dell'algoritmo, velocità del processore, attesa del completamento di operazioni, (es. acquisiz. dati)
Evoluzione temporale è influenzata da:

- Fattori interni: disponibilità di risorse, la velocità di esecuzione.
- Fattori esterni: il processo deve attendere il completamento delle attività di altri processi

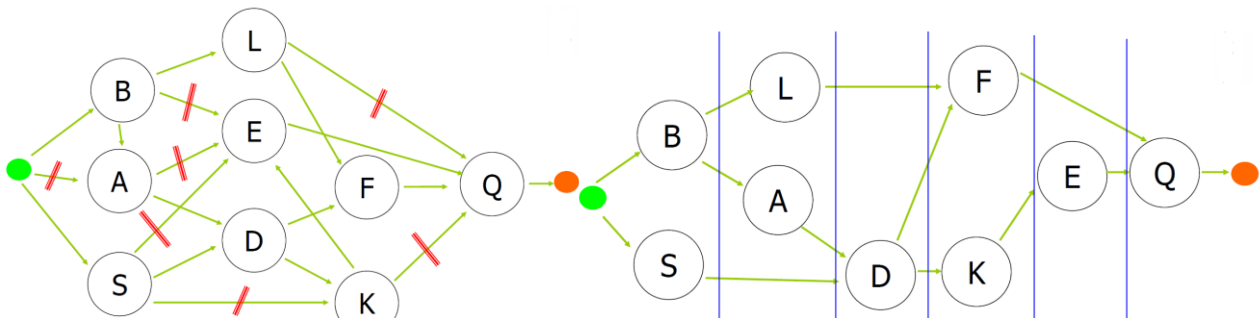
Scomposizione di un processo



Conveniente per motivi di analisi, non è univoca, mette in evidenza e raggruppa le attività significative del processo, non equivale alla scomposizione del programma in sottoprogrammi. Comporta dei vincoli di precedenza.
Grafo di precedenza: grafo diretto aciclico, in cui gli archi esprimono una relazione di precedenza tra 2 processi.
Simboli: •Pj: istante iniziale di Pj. **Pj•:** istante finale di Pj. **<•**: precede. **•>**: segue.
Sistema chiuso: se ha un unico processo iniziale ed un unico processo finale.
Un sistema aperto può sempre essere reso chiuso con l'aggiunta di due processi fittizi.
Calcolo durata complessiva sistema: serie=somma tempi, parallelo=massimo tempi.
Massimo (grado) parallelismo: la cardinalità del più grande sottoinsieme di processi per cui presi due qualsiasi nodi, questi non sono reciprocamente ne predecessore ne successore, diretto, indiretto, dell'altro.
Quindi esiste una partizione ordinata di nodi del grafo:

- $I_0 = \{\alpha\}$: radice del grafo (vera o fittizia)
- $I_{j+1} = \{nodi\ sucessori\ diretti\ solamente\ di\ nodi\ \in\ \cup_{k=0}^{j} I_k\}$: (U: nodi che sono già messi in qualche sottoinsieme)
- $I_{ultimo} = \{\omega\}$: nodo di chiusura che non ha successori.

Eliminare archi ridondanti: …Y_i<•…<•X<•…<•Y_i<•Z quindi Y_i<•Z è ridondante.



Se la durata dei processi è diversa, il sistema può avere un parallelismo più alto, e in caso non ci siano abbastanza processori si devono aggiungere dei vincoli di precedenza per ridurre il parallelismo.

Risorse: qualunque entità necessaria ad un processo per sviluppare la propria attività.

- **Condivisibili**: possono essere usate da più di un processo contemporaneamente.
- **Consumabili**: possono non essere riutilizzate.
- **Sottraibili**: possibilità di essere sottratte ai processi che ne stanno usufruendo. Preemtion: sottrazione forzata.

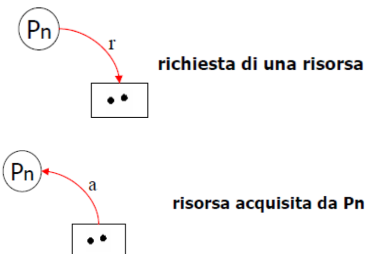
Stallo (deadlock): quando il sistema non può raggiungere il suo stato finale in quanto i processi si bloccano a vicenda.

Condizioni necessarie:

- **Mutua esclusione:** le risorse coinvolte non sono condivisibili.
- **Allocazione parziale**: un processo non richiede tutte le risorse necessarie in un unico step, ma in diversi momenti della sua evoluzione.
- **Non sottraibilità:** solamente il processo che sta usando la risorsa è in grado di rilasciarla.
- **Attesa circolare:** i processi sono in attesa di risorse occupate da altri processi a loro volta in attesa di risorse occupate dai primi. (P1 attende una risorsa da P2 che attende da … Pn che attende da P1).

Individuare lo stallo

Graficamente (Grafo di Holt):



- Definisco un ordinamento arbitrario dei processi i quali verranno esaminati nell'ordine prefissato.
- Se un processo può acquisire tutte le risorse di cui abbisogna si è certi che esso termina e si possono eliminare i legami con il magazzino.
- Itero il ciclo.

Se non riesco ad eliminare tutti i legami, il sistema è in stallo.

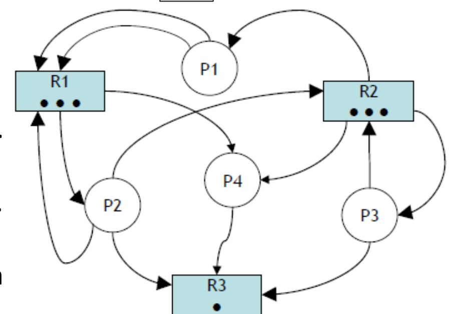


Tabelle:

- Tabella A= S: risorse totali, P1÷Pn: richieste massime di risorse. (Prima di iniziare)
- Tabella B= S: risorse assegnate, P1÷Pn: risorse ricevute. (Istante t)
- Tabella C=A-B. Cioè S: risorse libere, P1÷Pn: risorse ancora da acquisire.

Algoritmo: (Ridurre tabella C)

- Una riga può essere ridotta se risulta vettorialmente ≤ di riga S.
- Una riduzione incrementa S (tabella C) di riga eliminata nella tabella B.

A	R_1	R_2	R_3
S	3	3	1
P_1	2	1	0
P_2	2	1	1
P_3	0	2	1
P_4	1	1	1

B	R_1	R_2	R_3
S	2	3	0
P_1	0	1	0
P_2	1	0	0
P_3	0	1	0
P_4	1	1	0

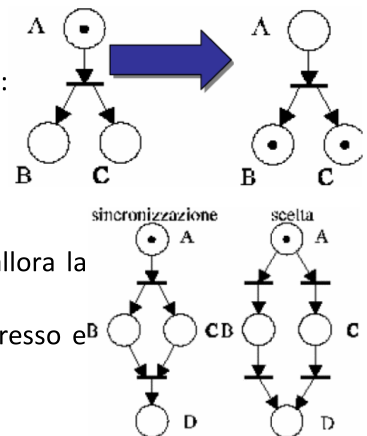
C	R_1	R_2	R_3
S	1	0	1
P_1	2	0	0
P_2	1	1	1
P_3	0	1	1
P_4	0	0	1

Prevenzione dello stallo: eliminando almeno una delle 4 condizioni necessarie per lo stallo.

- Allocazione globale delle risorse: applicabile quando sono note le risorse utilizzate dai processi durante la loro evoluzione (rimuove la condizione dell'allocazione parziale).
- Allocazione gerarchica delle risorse: viene stabilito un ordinamento nelle risorse.
  - Un processo che stia usando risorse, ne può ottenere altre solo di ordine più elevato.
  - Per ottenere risorse di ordine più basso, deve prima:
    - Rilasciare quelle di ordine maggiore
    - Richiedere la risorsa di ordine inferiore,
    - Richiedere di nuovo quelle superiori.

  Rimuovendo così l'attesa circolare simulando il preemption. Però richiede il rilascio volontario delle risorse.
- **Algoritmo del banchiere.** Le risorse vengono concesse ad un processo solo se le risorse residue sono sufficienti alla terminazione del processo. (Rimuove la condizione di attesa circolare)

2. Sincronizzazione (Reti di petri)

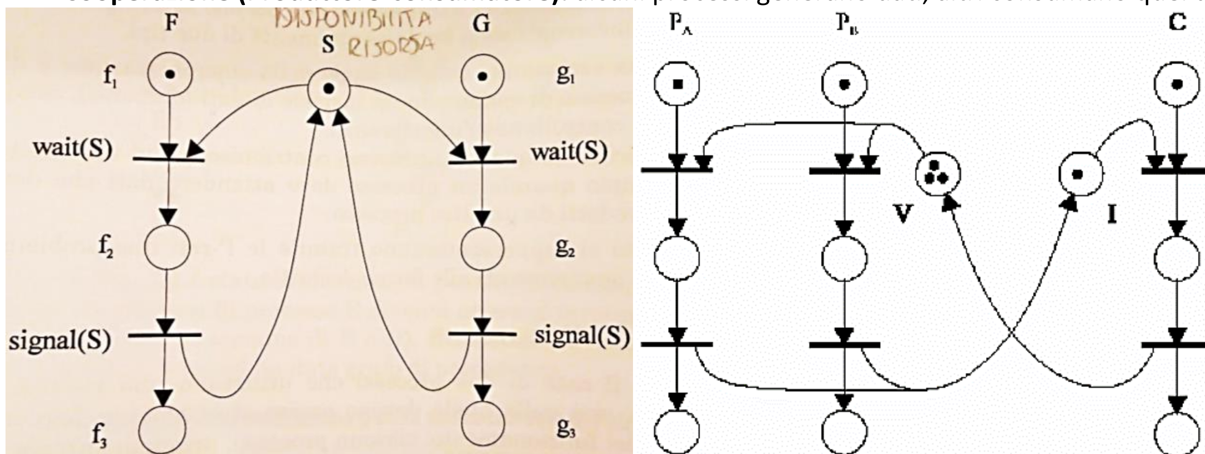


Rete di Petri: un grafo bipartito orientato in cui sono presenti 2 tipi di nodi: posti e transizioni.
Gli archi (orientati) vanno dai posti alle transizioni o dalle transizioni ai posti.
I posti hanno uno stato indicato con un intero N≥0 detto marcatura.
Regola di evoluzione:

- Se tutti i posti in ingresso ad una transizione sono marcati(N>0), allora la transizione può scattare.
- Lo scatto è istantaneo: viene tolta una marca da ogni posto di ingresso e messa una in ogni posto di uscita.
- Può scattare una sola transizione alla volta.

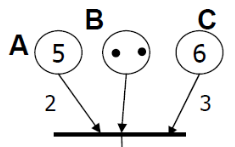
Interazioni tra processi:

- **Mutua esclusione:** prima di entrare nella zona in cui opera la risorsa (sezione critica), deve controllare se la risorsa è libera, se lo è entra nella sezione critica, altrimenti attende (sezione d'attesa). Sezione residua: il processo agisce solo su risorse private.
- **Cooperazione (Produttore-consumatore):** alcuni processi generano dati, altri consumano quei dati.

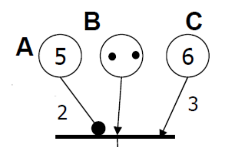


Estensioni:

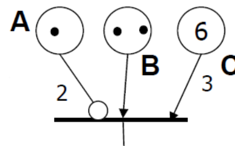
Rete pesata:
Ogni arco ha associato un peso non nullo.
In caso di scatto vengono tolte/aggiunte tante marche quanto il peso dell'arco



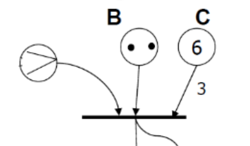
Arco abilitante:
La transizione è abilitata se la marcatura del posto è ≥ peso dell'arco.
In caso di scatto è ininfluente, equivale ad un 'cappio' sulla transizione.



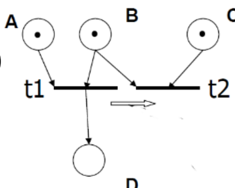
Arco inibitore:
La transizione è abilitata se la marcatura del posto è ≤ peso dell'arco.
In caso di scatto è ininfluente, equivale ad un 'cappio' sulla transizione.



Posto esterno:
È un posto sorgente e/o pozzo di marche, serve per immaginare un collegamento con un'altra parte della rete volutamente non specificata.

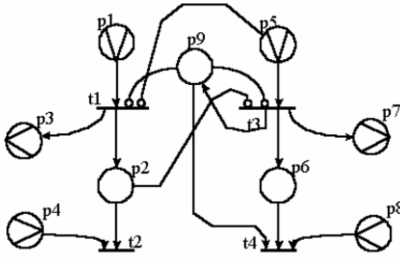


Priorità statica non Transitiva:
t1 → t2 (t1 è più prioritaria di t2) l'abilitazione di t1 induce l'inibizione di t2.

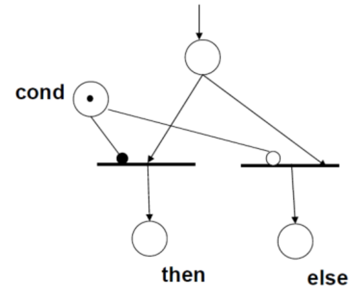


Esempi

Lettori-scrittori: Più processi lettori possono accedere contemporaneamente alla risorsa, uno scrittore ha accesso esclusivo alla risorsa.



If-else



Formalismo:

$\Phi = (P, T, F, M_0, W, Z)$

P insieme finito dei posti $|P|=n$

T insieme finito delle transizioni $|T|=m$

$F \subseteq ((P\times T\times\{r, e, i\}) \cup (T\times P))$ insieme finito degli archi con:

- (pi, tj, r) generico arco d'ingresso (normale)
- (pi, tj, e) generico arco abilitante
- (pi, tj, i) generico arco inibitore
- (tj, pi) generico arco d'uscita

$M_0 \in N^n$ marcatura iniziale

$W: F \rightarrow N - \{0\}$ funzione dei pesi

$Z \subseteq T\times T$ priorità statica (non transitiva)

La transizione tj risulta abilitata se:

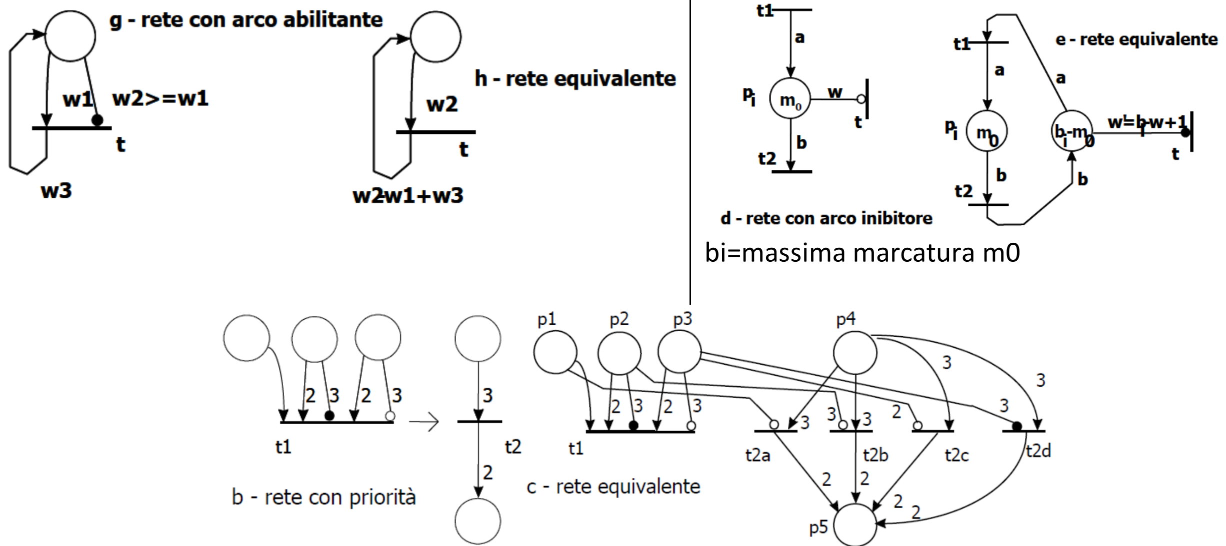
a) $\forall$(pi, tj, r)$\in$F Mk(pi) $\geq$ W(pi, tj, r)

b) $\forall$(pi, tj, e)$\in$F Mk(pi) $\geq$ W(pi, tj, e)

c) $\forall$(pi, tj, i)$\in$F Mk(pi) < W(pi, tj, i)

d) non esiste (tl, tj)$\in$Z tale per cui tl è abilitata da Mk

Trasformazioni equivalenti: per rappresentare le estensioni quando nel calcolatore c'è solo la rete di base.



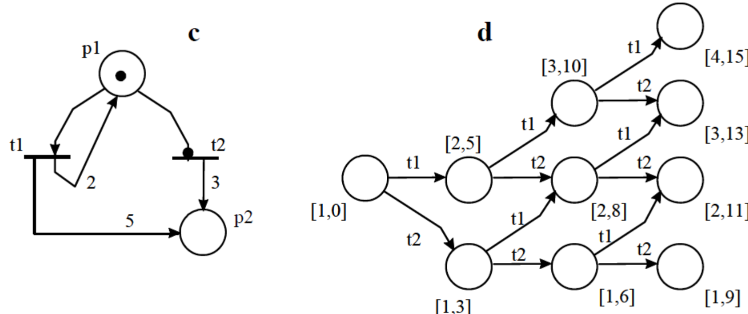
Grafo di raggiungibilità

$\Gamma\phi_0 = (Q = R(\phi_0), \Sigma = T, \delta)$

$Q = R(\phi_0)$: insieme, anche non finito, degli stati del grafo (marcature raggiungibili da quella iniziale)

$\Sigma = T$: insieme delle transizioni

$\delta: Q \times \Sigma \rightarrow$ Q: funzione di transizione di stato (condizione di attivazione, cioè scatto di una transizione)



3. Programmazione Concorrente

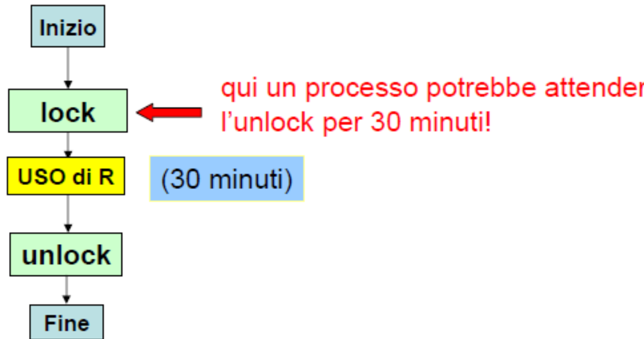
Mutua esclusione:

Sezione Critica: istruzioni con la quale un processo opera su una risorsa comune, le operazioni su variabili di stato della risorsa sono Sezioni critiche.

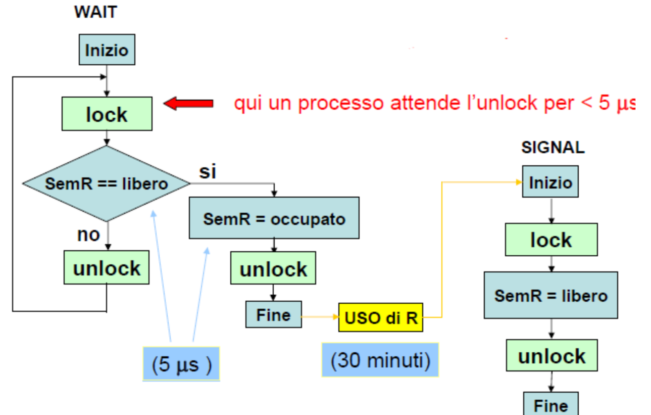
Problema: Se qualcuno verifica la disponibilità di una risorsa dopo che è già stato effettuato il controllo ma prima che venga effettuato il cambio della variabile di stato, la risorsa viene assegnata contemporaneamente a più processi.

Soluzioni:

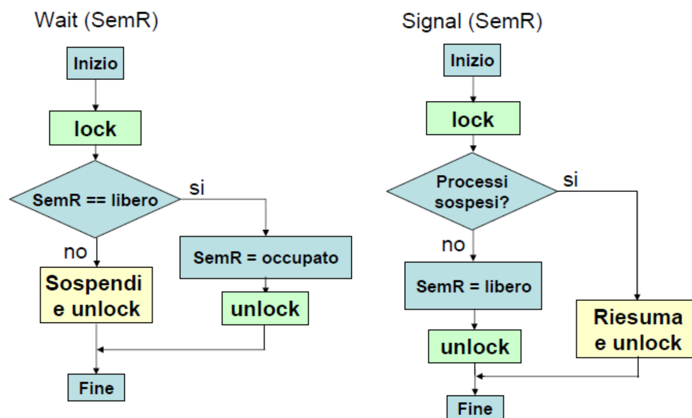
Lock e unlock: coppia di parentesi, che rendono indivisibile la sequenza di operazioni interne.



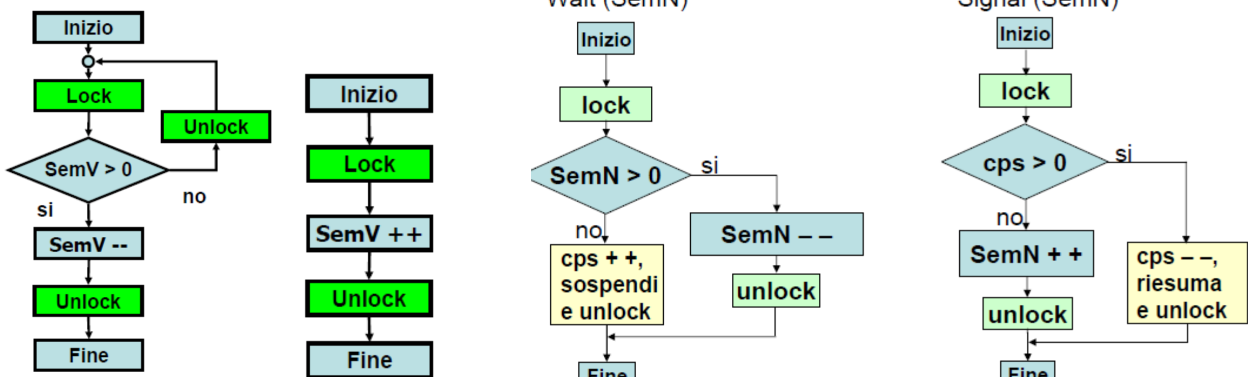
Wait e Signal: operazioni atomiche (hardware) associate al semaforo.



Semaforo con sospensione dei processi: Al semaforo va associata una coda di processi sospesi.



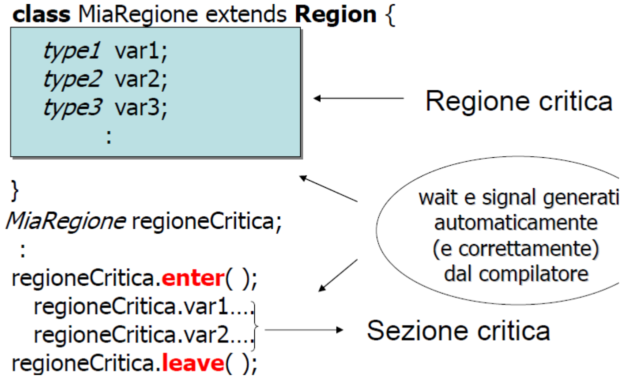
Semafori Numerici: variabile di stato intera:



Contatore dei processi sospesi (cps)

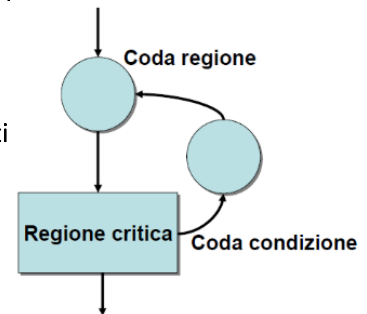
Semaforo privato (ad un processo): quando quel processo è l'unico che può eseguire l'operazione di wait sul semaforo. L'operazione di signal può essere eseguita da chiunque.
In sostanza: il processo attende la condizione di sincronizzazione eseguendo wait sul proprio semaforo privato poi il processo stesso (o un qualsiasi altro processo), quando verifica che la condizione di sincronizzazione è vera, esegue signal su quel semaforo.

I semafori consentono la soluzione di qualsiasi problema di sincronizzazione ma sono strumenti di basso livello. Altri modi (alto livello): regioni critiche e monitor.
Regioni critiche: Semplificano le sincronizzazioni assicurando la mutua esclusione nelle sezioni critiche.

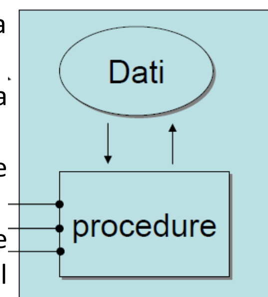


Regioni critiche condizionali: La mutua esclusione non sufficiente a risolvere i problemi di sincronizzazione, è spesso necessario verificare le condizioni di sincronizzazione.

- Coda regione: sospende tutti i processi che tentano di accedere alla sezione critica quando essa è già impegnata da un altro processo.
- Coda condizione: permette la sospensione di quei processi che, entrati nella sezione critica, sono dovuti uscire perché la condizione di sincronizzazione non era verificata.
- Quando un processo che completa la regione critica tutti i processi in attesa sulla Coda condizione passano alla coda regione.

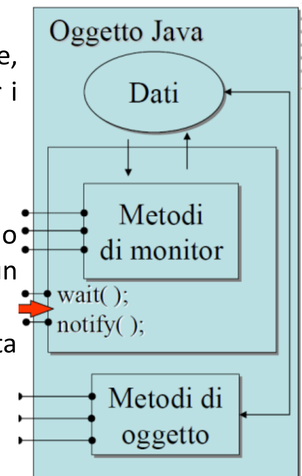


Monitor: permette di associare ad un insieme di variabili condivise da più processi un insieme di procedure, le uniche che possano agire su di esse. Il tutto in mutua esclusione.
Ogni processo può agire solo sulle variabile che gli appartengono, per accedere a variabili comuni deve eseguire determinate procedure.
Un processo che esegue le procedure di monitor lo fa in mutua esclusione (acquisisce e detiene il **lock** del monitor).
Un processo che utilizza il monitor può non completare la procedura (se non verificate le condizioni di sincronizzazione), può sospendersi (**wait**) in una coda all'interno del monitor.



Monitor di Hansen: l'operazione **signal** è l'ultima istruzione di un metodo del monitor (oltre a return).
In questo modo quando un processo che esegue signal sulla coda del monitor libera altri processi sospesi, altrimenti se la coda è vuota viene riabilitato l'accesso ai processi in coda fuori dal monitor.
Monitor di Hoare: il signal può essere richiamato più volte all'interno della stessa procedura, si dà priorità al risvegliato, Il risvegliante attende su una coda particolare detta **urgent** e riprenderà l'esecuzione all'uscita dal monitor del risvegliato.

Monitor di java:



Ad ogni oggetto viene associato un monitor, ovvero una regione critica condizionale, con 2 code, una per i thread che devono entrare (thread RUNNABLE), l'altra per i thread che non hanno trovato la condizione verificata.

public synchronized void NomeMetodoDiMonitor(){ … }

I metodi di monitor hanno la garanzia di essere eseguiti in mutua esclusione.

La JVM associa ad ogni oggetto un **lock** che un thread che vuole attivare un metodo di monitor cerca di acquisire prima eseguirne il codice. Se il lock è in possesso ad un altro thread deve aspettare in coda. while (! Condizione_attesa) { wait( );}

- Un thread, che possiede il lock, può doversi sospendere se non è verificata una condizione di sincronizzazione, mediante il metodo **wait():**
  - Rilasciando il lock
  - Inserendosi nella coda **wait set**.
  - Diventando **NOT RUNNABLE**.
- **notify()** libera un thread nella coda wait set, ma non si sa quale.
  Un thread reso RUNNABLE tramite notify() deve:
  - Riacquisire il possesso del monitor, in competizione con eventuali altri thread che siano in attesa di entrare nel monitor
  - Riverificare subito la condizione attesa, se essa non è verificata, riesegue wait() e si sospende di nuovo (NOT RUNNABLE) nel "wait set".
- **notifyAll()** estrae dal wait set e rende RUNNABLE tutti i thread in attesa.

Java consente di dichiarare **synchronized** anche blocchi di codice, utile quando:

- Solo una piccola parte del metodo manipola dati condivisi e va eseguita in mutua esclusione.
- Evitare che l'output si mescoli con quello di altri thread, si può rendere "mutuamente esclusivo" l'uso dell'oggetto System.out

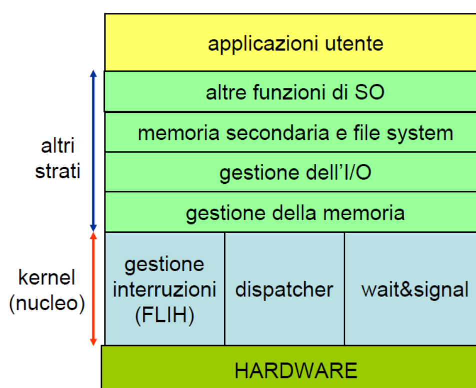
  synchronized(oggetto) { …codice… }

4. Il Nucleo

È il livello più basso del SO.

Caratteristiche essenziali hardware:

- Sistema di interruzioni: salvataggio del PC e del contesto. interrupt handler: determinala causa dell'interruzione. Attivazione della routine di servizio (ISR = Interrupt Service Routine) appropriata
- Interruzioni software: System Calls o Supervisor Calls (SVC)
- Meccanismo di protezione della memoria
- Istruzioni privilegiate (modo supervisore/modo utente)
- Orologio in tempo reale (real time clock).

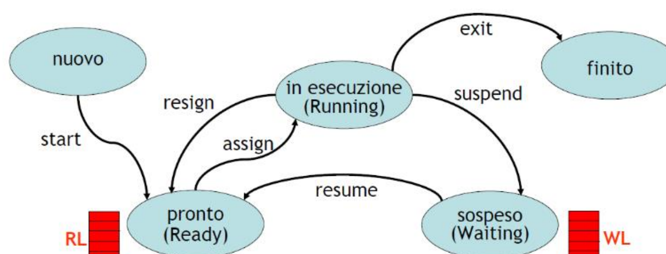


Caratteristiche del nucleo:

- Gestione interruzioni (FLIH)
- Gestione delle sincronizzazioni tra processi (semafori con primitive wait e signal)
- Funzioni di multiprogrammazione (scheduler / dispatcher)

Stati di un processo

- Running: un solo processo per processore
- Ready: una lista di processi (ready list)
- Waiting: una lista di processi (waiting list)

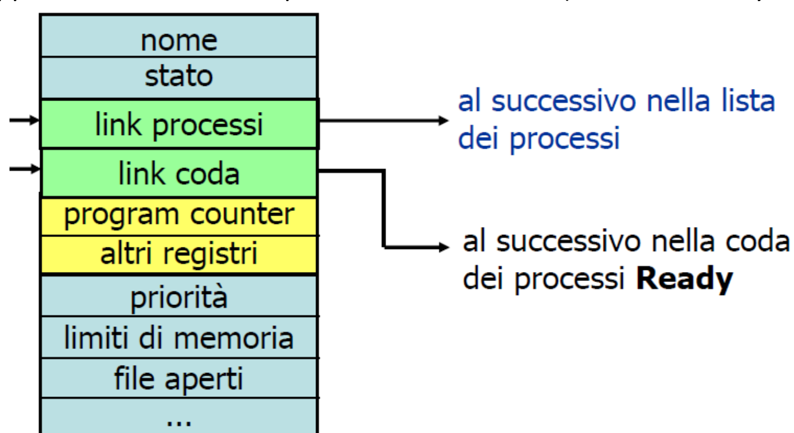


Rappresentazione dei processi

Il descrittore di processo (PD: Process Descriptor) struttura dati che contiene informazioni associate al processo: nome e/o id, stato (waiting, ready, running) contesto (registri del processore, altri registri, ecc.), altre informazioni per la gestione (risorse possedute, informazioni per la gestione della memoria, ecc.).

Quando un processo esce dallo stato Running, i registri del processore (contesto) vengono ricopiati nel suo PD. Quando un processo ritorna nello stato Running, il contesto viene ricopiato dal suo PD ai registri del processore.

Sono organizzati come liste concatenate: in questo modo le operazioni di inserzione e di estrazione sono rapide. Un PD può appartenere alla lista dei processi e ad una coda (ad es. alla Ready List)



4.1. Routine di interruzione di primo intervento (First Level Interrupt Handler)

- Determinare l'origine dell'interruzione
- Attivare il segmento di codice adeguato (ISR)

Le routine di servizio delle interruzioni fanno parte del nucleo del SO
Le interruzioni esterne (richieste da dispositivi di I/O) e quelle interne (SVC - SuperVisor Call o SC - System Call), sono gestite allo stesso modo.
Quando si verifica un'interruzione il processore si porta ad operare in modo supervisore: le routine di servizio delle interruzioni sono eseguite in modo supervisore.
In un **sistema non multitasking**, la routine di servizio di una interruzione (ISR):

- Esegue il servizio richiesto (ad es. acquisisce un carattere),
- Rimuove la causa che ha generato la richiesta di interruzione,
- Esegue l'istruzione di ritorno da interrupt (RTI), che ripristina il contesto del programma che era stato interrotto, il quale può quindi proseguire la sua esecuzione.

In un **sistema multitasking**:

- Il servizio di una interruzione può comportare la modifica dello stato di un processo (ad es. da Waiting a Ready);
- Le ISR terminano cedendo il controllo allo scheduler (anziché al processo interrotto);
- Lo scheduler decide (in base alla politica adottata) se:
  - Far proseguire il processo che era stato interrotto, o se
  - Rendere Running un altro processo (scelto tra quelli Ready, ponendo nello stato Ready quello interrotto).

4.2. Il Dispatcher

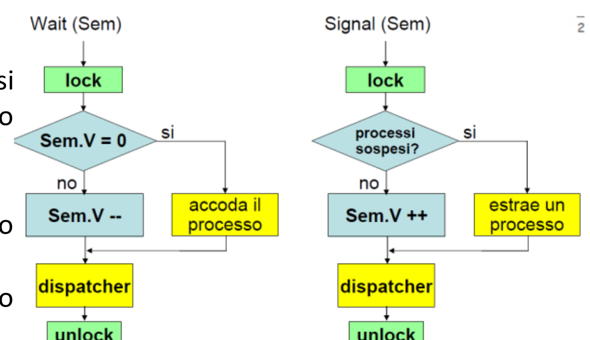
Il dispatcher (scheduler a basso livello) ha il compito di estrarre tra i processi nella Ready List quello da far diventare Running.

- Commuta il contesto (ricopia nel PD del processo Running i registri del processore e inserisce nei registri i valori estratti dal PD del processo selezionato per diventare Running).
- Porta il processore ad operare in modo utente.
- Inserisce nel Program Counter l'indirizzo della successiva istruzione del processo Running.

La scelta del processo da estrarre la fa lo **Scheduler ad alto livello** tramite modifica delle priorità dei processi Ready e Waiting. Lo scheduler ed il dispatcher interagiscono tra loro per ottenere il risultato voluto secondo la politica adottata.
Il dispatcher interviene, per selezionare il processo da rendere Running, nelle seguenti circostanze:

- Quando il processo Running passa nello stato Waiting (ad es. per una richiesta di I/O),
- Quando il processo Running passa nello stato Ready (ad es. per il verificarsi di una interruzione),
- Quando un processo Waiting passa nello stato Ready (ad es. per il completamento di una sincronizzazione), quando il processo Running termina.

4.3. Wait e Signal



Wait e Signal sono funzioni del kernel, attivate dai processi tramite System Call (SVC), indivisibili, cioè un solo processo alla volta può eseguirle.
La gestione della coda di processi associata al semaforo:

- Wait inserisce nella coda del semaforo il processo che non può proseguire.
- Signal estrae dalla coda del semaforo un processo (se c'è) e lo pone Ready.

Wait e Signal terminano attivando il dispatcher che decide quale processo sarà running ed esegue le eventuali commutazioni di contesto.
Semaforo realizzabile tramite: valore intero, puntatore alla coda, tipo di organizzazione della coda.

4.4. Scheduling

- A basso livello (dispatcher): componente del nucleo che si occupa di asegnare la CPU ad un processo.
- Ad alto livello: componente del S.O. che fissa i criteri di importanza tra i processi eseguibili.
- A medio termine: interviene in situazioni di sovraccarico.

Processi **I/O-bound**: Fasi di attesa lunghe, occupa la CPU per poco tempo.
Processi **CPU-bound**: occupa la CPU per molto tempo.
Overhead del SO: percentuale di tempo di CPU usato dal SO per svolgere le sue funzioni.
Scheduling non-preemptive: il processo Running rilascia volontariamente il processore.
Scheduling preemptive: il processo Running è forzato a rilasciare il processore (e messo nella Ready List) a vantaggio di un altro processo, evitando che un processo (CPU-bound) monopolizzi il processore.

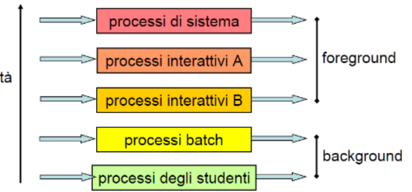
Criteri dello scheduler:

- **Utilizzo della CPU**: minimizzare i periodi di inattività.
- **Throughput**: massimizzare il n° di processi completati nell'unità di tempo.
- **Turnaround** time: minimizzare il tempo di completamento di un processo.
- **Reaction time**.
- **Deadline**: tempo massimo entro in cui il processo deve iniziare o terminare.
- Predicibilità.
- **Equità**: processi dovrebbero essere trattati allo stesso modo.
- Bilanciamento delle risorse.

Algoritmi di scheduling (monoprocessore):

- **FCFS - First Come, First Served**
  La CPU viene assegnata in ordine di richiesta. La Ready List (RL) è una coda FIFO.
  Algoritmo non-preemptive. Favorisce i processi CPU-bound.
  Effetto convoglio: mentre è in esecuzione un processo CPU-bound, tutti quelli I/O-bound diventano pronti e partono uno dopo l'altro per una breve esecuzione per risospendersi in attesa di I/O, rischio di avere la Ready List vuota.
- **RR - Round Robin**
  Time-slice: quanto temporale, allo scadere di esso il processo running viene messo in fondo alla Ready List.
  Algoritmo preemptive. overhead minimo, buon Throughput, trattamento equo a tutti i processi.
  Se Ts troppo grande: diventa uguale alla politica FCFS.
  Se Ts troppo piccolo: l'overhead di commutazione di contesto diventa inaccettabile.
- **PS - Priority Scheduling**
  Ad ogni processo è associato un indicatore di priorità, Ready List è ordinata per priorità decrescente.
  Algoritmo può essere sia preemptive che non-preemptive. L'overhead può essere alto.
  Rischio di starvation: i processi a priorità bassa possono rimanere in attesa indefinita, se ci sono sempre processi più prioritari pronti; per eliminare questo rischio è spesso adottata la tecnica di aging (aumenta la priorità col tempo d'attesa).
- **SPN - Shortest Process Next**
  È un caso particolare di PS in cui la CPU assegnata in base al tempo di esecuzione più piccolo.
  La Ready List è ordinata per tempo di esecuzione crescente, a parità di valore equivale a FCFS.
  Algoritmo non-preemptive. Favorisce i processi brevi. Buono throughput e tempo di risposta.
  Ma l'ordinamento della ready list comporta overhead. Rischio di starvation se entrano continuamente processi di breve durata.
- **SRT - Shortest Remaining Time**
  Forma Preemptive di SPN in cui il parametro valutato è il tempo di esecuzione rimanente.
  La Ready List è ordinata per tempo di esecuzione rimanente crescente.
  Può ridurre il tempo medio di attesa, ma l'overhead è ancora più alto di SPN.

- **HRRN - Highest Response Ratio Next**
  La Ready List (RL) è ordinata per fattore di risposta decrescente. $F_{r_i} = (a_i + e_i)/e_i$ con: a_i=attesa nello stato pronto, e_i=tempo di esecuzione stimato
  Algoritmo non-preemptive. Pur favorendo i processi brevi, favorisce anche quelli lunghi che attendono molto.
- **MS - Multilevel queue Scheduling**
  5 Ready List diverse, una per ciascun livello di priorità. Politiche di scheduling diverse: RR per il foreground, FCFS per il background. Un processo va in esecuzione solo se le code più prioritarie sono vuote. Algoritmo preemptive.



- **MF - Multilevel Feedback scheduling**
  Variante di MS in cui i processi possono migrare tra le code.
  Politica RR per ciascuna ready list: quando un processo ha esaurito il suo quanto temporale, subisce preemption e passa nella lista a priorità inferiore.
  I processi CPU-bound, scendono via via di priorità; quelli I/O bound conservano priorità elevata
  Rischio di starvation per prevenirlo:
  - La priorità aumenta con il tempo di attesa.
  - Quanti più grandi per le ready list di priorità inferiore.

Priority Inversion: problema che accade in un sistema con scheduling preemptive basato sulla priorità.

- Un processo a bassa priorità esegue wait(s) e poi subisce preemption a favore di uno a priorità più elevata che esegue wait(s) sullo stesso semaforo e quindi vi si accoda;
- Altri processi a priorità intermedia impediscono a quello di priorità bassa di avanzare ed eseguire signal(s);
- Di conseguenza i processi con priorità inferiore bloccano quello con priorità elevata (inversione della priorità);

Soluzione: il processo che impegna un semaforo eredita, temporaneamente, la priorità più elevata tra quelle dei processi in coda sullo stesso semaforo
L'adozione di questo meccanismo aumenta l'overhead.

5. Gestione della memoria

Gli obiettivi della gestione della memoria sono:

- **Rilocazione:** Il programmatore non sa dove il programma sarà caricato in memoria quando verrà eseguito; la posizione infatti dipende da: quali e quanti moduli compongono il programma, quali e quanti altri programmi sono presenti in memoria nel momento in cui esso verrà caricato. Inoltre, se la memoria occupata dal programma durante l'esecuzione dovesse servire per altri scopi il programma può essere temporaneamente ricopiato su disco (swap) e può venir successivamente collocato in una posizione diversa (rilocato) in memoria.
  Gli indirizzi di memoria contenuti nel codice e nei dati di un programma sono indirizzi logici, diversi da quelli fisici.
- **Protezione:** è necessario impedire che un processo acceda a locazioni di memoria di un altro processo. La verifica va fatta al momento della esecuzione e non dal compilatore perché molti indirizzi vengono calcolati al momento della esecuzione o i processi possono essere rilocati da una posizione di memoria ad un'altra. Le funzioni necessarie a questa verifica devono essere fornite dall'hardware.
- **Condivisione:** È necessario consentire che processi diversi accedano ad aree di memoria comuni. Nel caso che un insieme di subroutine sia utilizzato da più processi, conviene che di esse vi sia in memoria un'unica copia accessibile a tutti, piuttosto che averne tante copie replicate per ciascun processo.
- **Organizzazione logica:** I programmi sono costituiti da moduli, che vengono scritti e compilati separatamente. Il modo più naturale di vedere la memoria per il programmatore non è uno spazio di indirizzi lineare (da 0 fino all'indirizzo massimo), ma costituito da spazi di indirizzi separati (uno per ciascun modulo). La tecnica più appropriata per la gestione a moduli degli indirizzi è costituita dalla segmentazione.
- **Organizzazione fisica:** basata su due livelli: memoria centrale e memoria secondaria.
  La memoria centrale può non essere sufficiente a contenere un programma e i suoi dati: parte del programma deve stare in memoria secondaria. In un sistema multiprogrammato il programmatore non sa quanto spazio di memoria è disponibile per il programma. Trasferire programmi tra disco e memoria è compito del SO

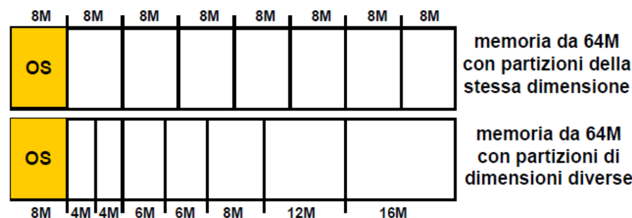
5.1. Tecniche di gestione della memoria

Partizionamento

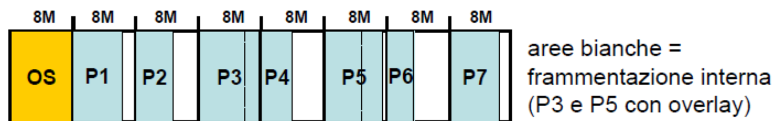
La memoria viene divisa in partizioni: di dimensioni fisse o variabili (partizionamento dinamico)
Per partizioni fisse anche i processi piccoli occupano una intera partizione (**frammentazione interna**).
Se non ci sono partizioni libere, il SO può liberarne una trasferendo su memoria secondaria (swap) il processo relativo.
Per partizioni variabili, a ciascun processo viene assegnata una partizione delle dimensioni richieste (nessuna frammentazione interna) però col passar del tempo si formano in memoria delle aree di dimensioni piccole non utilizzabili (**frammentazione esterna**).
La frammentazione richiede che periodicamente la memoria venga compattata, con impegno di CPU.

Partizionamento fisso:

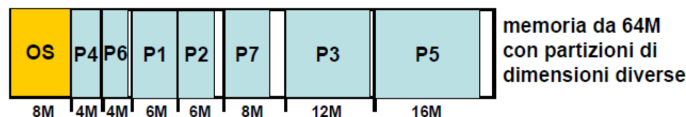
La memoria viene divisa in partizioni di dimensioni fisse: di dimensione tutte uguali o diverse.
Le dimensioni non possono essere modificate
Facile da realizzare, il grado di multiprogrammazione è limitato (un processo per partizione).



- Partizioni fisse di dimensioni uguali: un processo di dimensione inferiore o uguale a quella delle partizioni può essere caricato in una qualsiasi partizione libera.
  Se non ci sono partizioni libere, il SO può liberarne una trasferendo su memoria secondaria (swap) il processo relativo, se un processo è più grande di una partizione, il programma deve essere progettato con la tecnica di overlay. Anche i processi piccoli occupano una intera partizione: ciò provoca un uso inefficiente della memoria indicato come frammentazione interna.

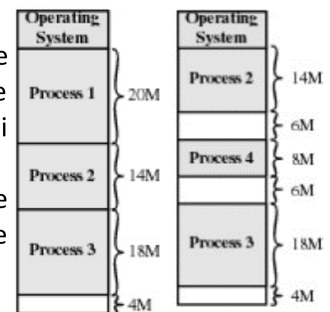


- Partizioni fisse di dimensioni diverse: processo collocato nella partizione più piccola in grado di contenerlo.
  Riduce la frammentazione interna: le partizioni piccole riducono lo spreco per processi piccoli, le partizioni grandi riducono la necessità di overlay, si può pensare di avere una coda di processi per ciascuna dimensione delle partizioni.



Partizionamento dinamico:

Il numero e le dimensioni delle partizioni sono variabili, a ciascun processo viene assegnata una partizione delle dimensioni richieste (nessuna frammentazione interna). Col passar del tempo si formano in memoria delle aree di dimensioni piccole non utilizzabili (frammentazione esterna).
Per recuperare queste aree non utilizzabili si spostano i processi in memoria e compattare le aree libere in un'unica area più grande. Però l'operazione richiede tempo di CPU.



Paginazione

La memoria fisica viene divisa in blocchi uguali, detti frame (o pagine fisiche), di dimensioni spesso piccole e uguali alla dimensione di un settore del disco.

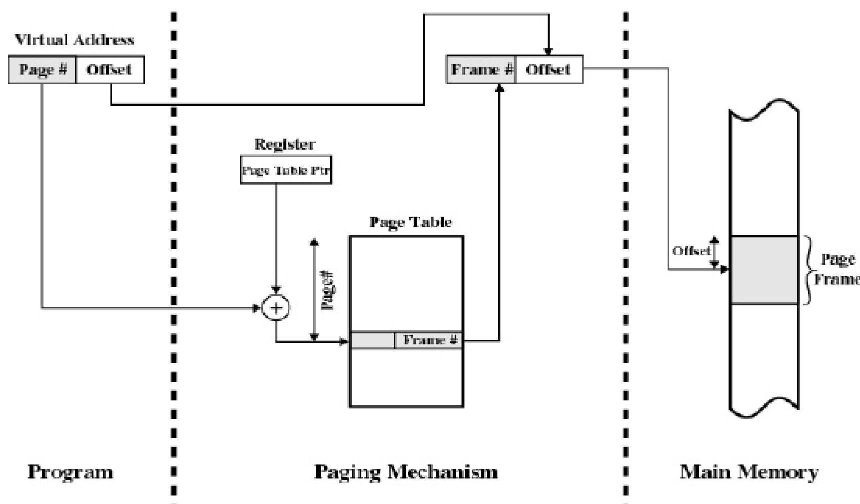
Anche lo spazio di memoria indirizzato da ciascun processo è diviso in blocchi delle medesime dimensioni, detti pagine (o pagine logiche): un indirizzo di memoria di un processo è diviso in due parti: indice di pagina e offset nella pagina.

Il SO mantiene una **Page Table** per ciascun processo: quando il processo è caricato in memoria, essa indica in quale frame si trova ciascuna sua pagina, i frame occupati da un processo possono non essere contigui.

Nella Page Table, l'elemento di indice i corrisponde alla i-esima pagina (dello spazio di indirizzi) del processo e contiene: l'indice del frame nella memoria fisica in cui è collocata la i-esima pagina del processo, alcuni bit di controllo: P = pagina presente nella memoria fisica, M = pagina modificata durante la permanenza in memoria fisica (se non è stata modificata non occorre ricopiarla su disco in caso di rimpiazzo), bit di protezione (sola lettura, sola esecuzione, accesso riservato al SO, …).

Page Table dei diversi processi hanno dimensioni diverse (dipende dalla dimensione del processo), processi molto grandi hanno page tabe con molti elementi.

La page table non è contenuta in registri appositi, ma è collocata in memoria, un registro (page table pointer) punta alla page table attiva.



Translation lookaside buffer (TLB):

Usando la page table (tenuta in memoria) per trasformare un indirizzo virtuale in un indirizzo fisico, c'è il problema che ogni accesso alla memoria ne richiede almeno due: uno per accedere alla page table, uno per accedere al dato. Quindi i tempi di accesso alla memoria raddoppiano.

Per limitare questo effetto, si usa una cache veloce in cui tenere gli elementi della page table usati più di recente, la Translation Lookaside Buffer (TLB) e ciascun suo elemento contiene: un page# (in memoria associativa, per ricerca parallela) e il corrispondente frame# (frame che contiene la pagina).

Uso del TLB: Dato un indirizzo virtuale (page#, offset), il page# viene cercato nella memoria associativa del TLB:

- Se trovato (hit), si estrae il corrispondente frame# e si costruisce l'indirizzo fisico (frame#, offset);
- Se non c'è (miss), si accede alla page table (indice=page#):
  - Se l'elemento della page table indica che la pagina cercata è presente in un frame nella memoria fisica: si costruisce l'indirizzo fisico (frame#, offset) e si aggiorna il TLB inserendo l'elemento trovato.
  - Altrimenti si ha un **page fault:** si porta in un frame la pagina cercata (swap in), si aggiorna la page table (nuova pagina presente), si costruisce l'indirizzo fisico (frame#, offset) e si aggiorna il TLB inserendo il nuovo elemento.

Segmentazione

Un indirizzo è costituito da: un indice di segmento e un offset. La segmentazione pone gli stessi problemi del partizionamento dinamico.

Ciascun processo è costituito da un insieme di segmenti di dimensioni diverse (uno per ciascuno dei moduli) e da una **Segment Table** contenente, per ciascuno dei suoi segmenti, le seguenti informazioni:

- L'indirizzo iniziale del segmento (inserito dal SO al momento in cui il segmento viene caricato in memoria)
- La dimensione del segmento
- Bit di controllo (presente P, modificato M, …)
- Bit di protezione (read only, permessi, …)

indirizzo base	lunghezza	P	M			

Da indirizzo virtuale a indirizzo fisico:

L'indirizzo virtuale definito dal codice è composto da un indice di segmento (#seg) e un offset (Off).

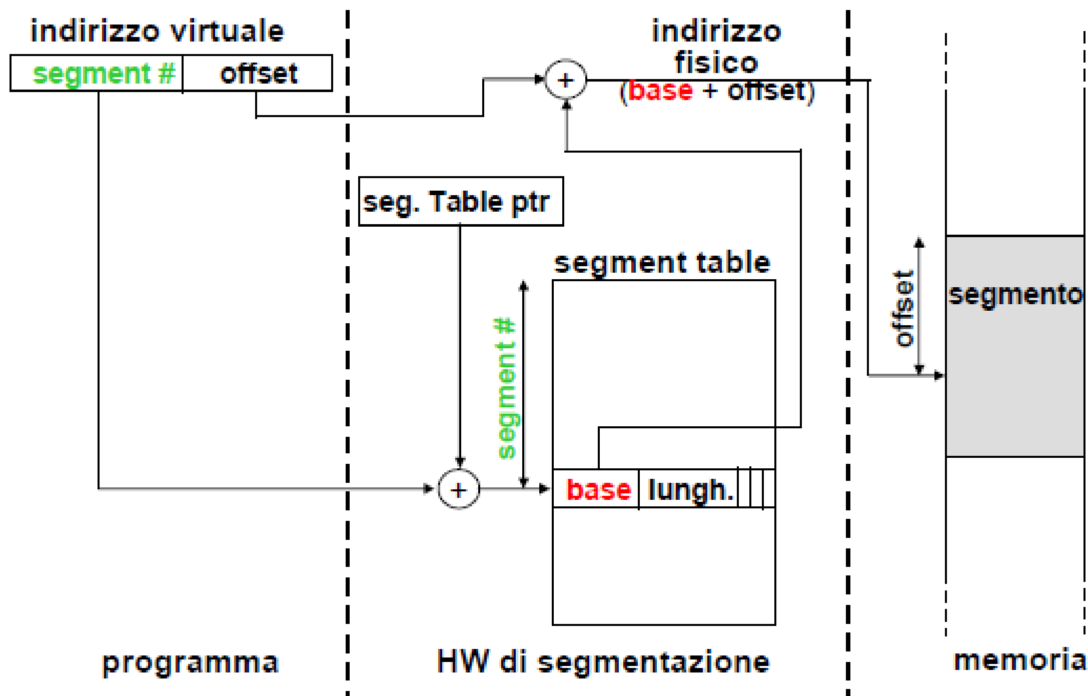
L'indice di segmento individua un elemento della segment table da cui si estrae: l'indirizzo iniziale del segmento (base) e la dimensione del segmento (Dim).

L'offset (Off) viene confrontato con la dimensione (Dim): se Off > Dim l'indirizzo è non valido (eccezione) altrimenti è valido.

L'indirizzo fisico è dato da base + Off (indirizzo iniziale del segmento, estratto dalla segment table, + offset, estratto dall'indirizzo virtuale).

Per realizzare un sistema di memoria virtuale (MV), è necessario che nell'hardware sia presente: un meccanismo di paginazione, o di segmentazione, oppure entrambi.

Sfruttando questi meccanismi il SO può gestire i trasferimenti delle pagine e/o dei segmenti tra memoria centrale e disco.



Segmentazione con paginazione

Combinando segmentazione e paginazione si combinano i vantaggi di entrambe: la paginazione è trasparente al programmatore, elimina la frammentazione esterna, facilita la gestione della memoria, la segmentazione è visibile al programmatore, consente la modularità, la protezione e la condivisione dei segmenti.
Ad ogni processo è associata una **segment table** ad ogni segmento è associata una **page table.**
Da indirizzo virtuale a indirizzo fisico:

A partire dall'indirizzo virtuale:

segment #	page #	offset

Il segment# individua un elemento nella Segment Table:

indirizzo base	lunghezza	P	M			

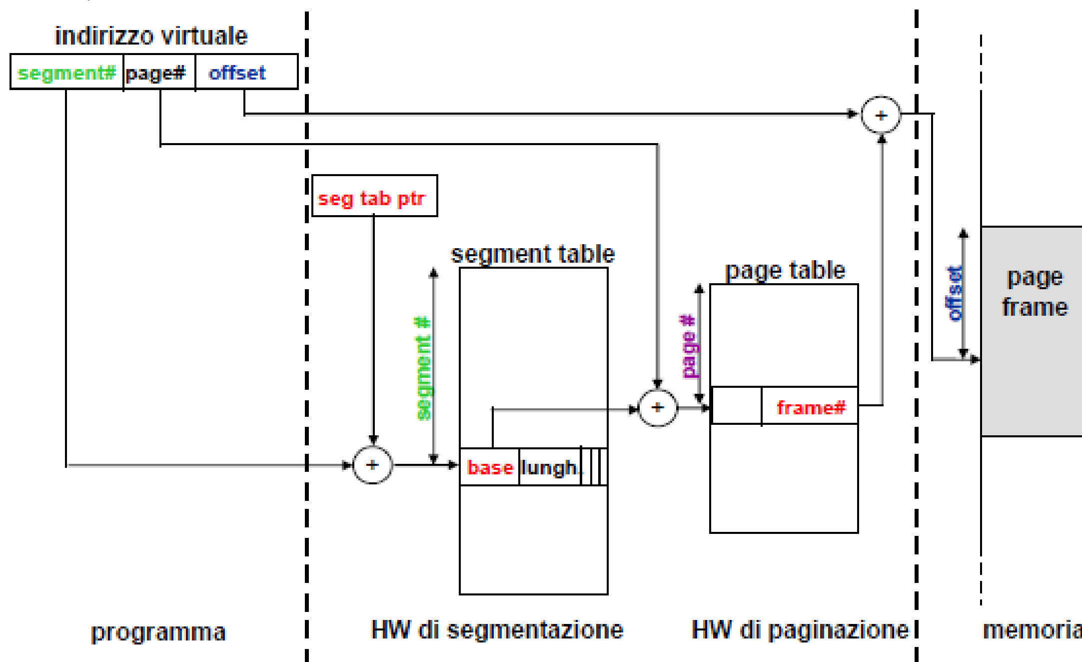
Se offset > lunghezza: eccezione di address error;

(page# + indirizzo base) individua elemento nella Page table:

frame #	P	M

(frame#, offset) fornisce l'indirizzo fisico:

frame #	offset



Caratteristiche importanti: gli indirizzi di memoria vengono tradotti in indirizzi fisici durante l'esecuzione, per cui i processi possono subire swap (out e in) dalla memoria e occupare aree di memoria diverse. I diversi blocchi (pagine o segmenti) di un processo possono essere collocati in aree non contigue della memoria di conseguenza: non è necessario che tutti i blocchi (pagine o segmenti) di un processo siano caricati in memoria durante l'esecuzione, il SO carica in memoria solo alcuni blocchi (**resident set**).

Esecuzione di un programma:

Quando il processo P richiede un indirizzo che non si trova nel resident set (memory fault): viene generato un interrupt e interviene il SO che: pone il processo nello stato "waiting", avvia la lettura da disco del blocco richiesto (swap in), rende running un altro processo Q.
Quando la lettura da disco del blocco richiesto è completata viene generato un interrupt e interviene il SO che riporta il processo P nello stato "ready".
A regime, quando la memoria fisica è tutta occupata, la lettura di un nuovo blocco in seguito a un memory fault, richiede di scegliere (secondo una politica di rimpiazzo opportuna) quale altro blocco deve essere rimpiazzato (swap out).

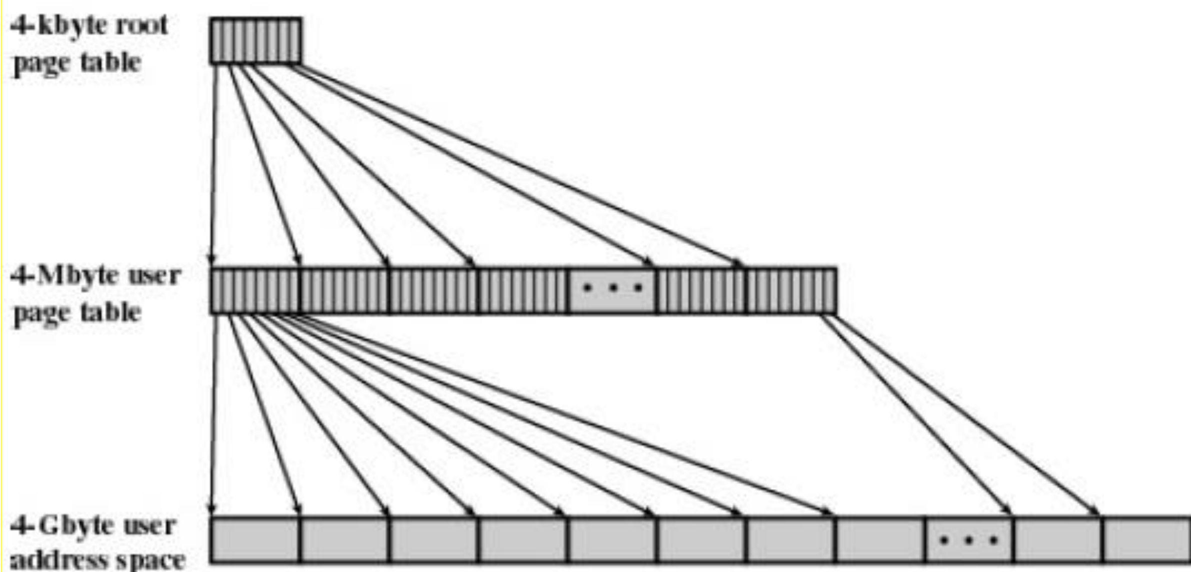
5.2. Memoria virtuale

Con la memoria virtuale il numero di processi presenti in memoria è maggiore (di ciascun processo è presente in memoria solo il resident set, rimane più memoria libera per altri processi), ciò produce maggior rendimento nell'uso del processore (più processi sono in memoria, meno probabile è che la ready list rimanga vuota). Ogni processo può indirizzare più memoria di quella fisica disponibile.
Memoria virtuale = spazio di indirizzi di memoria (maggiore della memoria fisica presente) disponibile a un processo.

Principio di località
In ogni breve intervallo di tempo, gli indirizzi di memoria richiesti da un processo tendono ad essere localizzati nelle medesime aree, ciò consente di fare previsioni abbastanza affidabili su quali blocchi (pagine o segmenti) del processo serviranno nel futuro (immediato).
Una volta portato in memoria fisica un blocco, il processo continuerà ad utilizzarlo per un po' di tempo, solo un accesso su N provoca un memory fault, probabilità di page fault: $pF = 1/N$ ($0 \leq pF \leq 1$),
Il sistema è tanto più efficiente quanto maggiore è N (quanto minore è pF).

Trashing
In un sistema di memoria virtuale (MV) può accadere che il SO decida di rimpiazzare (swap-out) un blocco poco prima che il processo cerchi di accedervi allora questo va riportato subito in memoria swap-in, con il conseguente swap-out di un altro blocco. Se questo succede troppo frequentemente, si parla di **trashing**: il processore passa gran parte del suo tempo ad eseguire swap, invece che ad eseguire le istruzioni dei processi.

Page table di grandi dimensioni: alcuni processi possono occupare uno spazio virtuale enorme:
Esempio: pagine da 4KB (offset di 12 bit), il numero di pagine virtuali è 220 = 1M, Una page table con 1M elementi, ciascuno da 4 byte, occupa 4MB (per un solo processo!)
In molti sistemi di MV le page table sono collocate nella memoria virtuale (sono paginate come le altre pagine) e sono organizzate a 2 livelli: una root page table (di primo livello) per paginare la page table (di secondo livello) del processo. Del processo running deve essere in memoria la root page table e la parte della page table che referenzia le pagine in esecuzione.



Page table invertita - PTI

Invece dei 2 livelli, alcuni SO usano una page table invertita che contiene un elemento per ciascun frame fisico, per cui la sua lunghezza è fissa e indipendente dal numero di processi e di pagine virtuali, ogni elemento di PTI contiene l'indice del frame (frame#) e l'indice della pagina virtuale (page#) presente nel frame.
Una hash table trasforma il page# in un indice nella PTI: se l'elemento della PTI individuato contiene il page# richiesto, si estrae il frame# e lo si concatena all'offset per ottenere l'indirizzo fisico, altrimenti si segue il link che porta agli elementi della PTI alternativi (usati in caso di collisione hash) e si verifica se contengono il page# richiesto, se il page# non viene trovato si ha un page fault.

Caricamento delle pagine con PTI:

La Hash Table serve a individuare l'elemento della PTI che (forse) contiene il frame in cui si trova una pagina, dato un indirizzo (page#, offset), dal page #, con la funzione hash H, si ottiene l'indice nella hash table: H(p) → i. essendo $0 \le p \le NPL$, $0 \le i \le NFF$, con NPL >> NFF. NPL = n. di pagine logiche; NFF = n. di frame fisici.
Quando il SO carica la pagina page# nel frame frame#, usando la H(p) trova l'elemento della HT che dovrà puntare all'elemento Z della PTI associato a quel frame: se l'elemento della HT è libero, lo fa puntare all'elemento Z della PTI e inserisce in quest'ultimo il page#, altrimenti, seguendo il link, aggiunge in coda alla linked list l'elemento Z della PTI e vi inserisce il page#.

Da indirizzi virtuali a indirizzi fisici con PTI:

Quando il SO deve verificare se una pagina è in memoria: dato l'indirizzo virtuale (page#, offset), dal page#, tramite la H(p) e la HT, ottiene l'indice di un elemento della PTI: se l'elemento individuato contiene il page# richiesto, estrae il frame# e lo concatena all'offset per ottenere l'indirizzo fisico (frame#, offset), altrimenti percorre la linked list che individua gli elementi della PTI che possono contenere il page# richiesto, se il page# non viene trovato si ha un page fault e il SO procede al caricamento della pagina in un frame.

Dimensione delle pagine

Per pagine di lunghezza inferiore ad un valore minimo diventa elevato il numero di pagine di ciascun processo e, quindi, la dimensione delle page table: aumenta la porzione di page table in memoria virtuale (su disco anziché in memoria) e aumenta la probabilità di page fault doppi (su page table e sulla pagina dati).
Per pagine di lunghezza superiore a questo valore minimo: quanto maggiore è la lunghezza delle pagine tanto maggiore è la frammentazione interna, tanto minore è il numero di pagine di ciascun processo in memoria fisica e maggiore è la probabilità di page fault.
Quindi: al di sopra di un valore minimo, all'aumentare della dimensione delle pagine aumenta la probabilità di page fault.

La Probabilità di Page fault

Aumenta all'aumentare delle dimensioni delle pagine (diminuisce lo sfruttamento della località: le pagine contengono dati lontani da quelli cui si accede). Torna a scendere, superato un valore limite, quando la dimensione della pagina si avvicina o supera quella dei processi. A parità di dimensione delle pagine, dipende molto dalla grandezza del **working set** (numero di frame allocati a un processo), diminuisce al crescere del working set.

Pagine di dimensioni variabili

Esistendo processori in grado di gestire pagine di dimensioni variabili, è stata considerata la possibilità di usare pagine di dimensione diversa, ad esempio: pagine grandi per il codice dei processi e piccole per gli stack dei thread. La gestione di pagine di dimensioni differenti è complicata: la gran parte dei SO prevede pagine di dimensioni uguali.

5.3. Gestione della memoria virtuale

Il SO interviene per attuare delle scelte, basate su opportune politiche:

- **Fetch policy:** quando trasferire una pagina da disco a memoria.
- **Placement policy:** in quale area della memoria collocare la pagina.
- **Replacement policy:** quale pagina (segmento) rimpiazzare.
- **Resident set management**: quante pagine di un processo tenere in memoria.
- **cleaning** (sgombero): quando riscrivere su disco una pagina da rimpiazzare.
- **Load control:** controllare il carico della CPU.

Fetch policy

- **demand paging:** la pagina viene portata in memoria solo quando il processore chiede di accedere ad un suo indirizzo.
- **prepaging**: oltre alla pagina richiesta, si portano in memoria anche le pagine che risiedono su settori adiacenti del disco. È più efficiente trasferire da disco più settori/pagine consecutivi piuttosto che un settore alla volta ma c'è il rischio di portare in memoria pagine che non servono

Placement policy

- **Best-Fit:** viene scelto il blocco libero più piccolo in grado di contenere il processo.
- **First-Fit:** viene scelto il primo blocco libero abbastanza grande da contenere il processo.
- **Next-Fit**: come first-fit, ma la ricerca parte dall'ultimo blocco assegnato, anziché dall'inizio della memoria.

Replacement policy

Decidono la pagina (o segmento) da rimpiazzare per far posto ad una pagina che deve essere trasferita in memoria. Le pagine del SO sono mantenute in memoria centrale in stato locked quindi non sono soggette a replacement.

Algoritmi di replacement:

- **Ottimo:** la pagina selezionata per il rimpiazzo è quella cui il processore in futuro chiederà di accedere più tardi di tutte, impossibile da realizzare ma serve come elemento di confronto per gli altri metodi.
- **FIFO**: la pagina selezionata per il rimpiazzo è quella presente in memoria da più tempo.
- **LRU**: la pagina selezionata per il rimpiazzo è quella che da più tempo non subisce accessi. Migliore di FIFO, ma complicato da realizzare e con grande overhead.
- **Clock**: (coda circolare) l'efficienza di LRU con l'overhead del FIFO
  A ciascun frame è associato uno use bit, che indica se la pagina è stata usata. Lo use bit viene posto a 1 quando la pagina viene caricata in memoria o quando la pagina subisce un accesso (viene usata). Il SO esamina, ad uno ad uno, gli use bit se lo use bit vale 1, lo azzera e prosegue l'esame, quando incontra il primo frame con use bit = 0, seleziona quel frame per il rimpiazzo.
  Variante: Oltre allo use bit, si tiene conto anche del modified bit che indica se la pagina è stata modificata. Il SO fa una scansione cercando un frame con use=0, mod=0 (senza azzerare gli use bit uguali a 1 che incontra). Se completa un intero giro senza averne trovato uno riscandisce il buffer cercando un frame con use=0, mod=1, questa volta azzerando tutti gli use bit=1 che incontra.
  Se, al termine del secondo giro, non ne ha trovato ancora nessuno, riinizia dal primo passo.
- **Page Buffering:** È una variante dell'algoritmo FIFO per aumentare l'efficienza.
  Il SO cerca di mantenere un piccolo numero di pagine candidate alla sostituzione in due liste concatenate: una contente pagine vuote o non modificate, l'altra pagine modificate. In caso di page fault si utilizza una pagina della prima lista, quando questa lista arriva sotto una certa soglia, alcune pagine vengono prese dalla seconda.

Gestione del resident set

Resident set: pagine di un processo che, in un dato istante, sono contenute in frame della memoria fisica.
Allocazione fissa: a ciascun processo è assegnato un numero fisso di frame. In seguito ad un page fault, viene rimpiazzata una pagina dello stesso processo.
Allocazione variabile: il numero di frame assegnati a ciascun processo varia. Il SO deve valutare il comportamento dei processi per decidere se aumentare o diminuire il numero di frame ad essi assegnati.
Dominio di rimpiazzo:

- **Locale**: quando si verifica un page fault, la pagina rimpiazzata è dello stesso processo.
- **Globale**: quando si verifica un page fault, la pagina da rimpiazzare può essere scelta tra tutte le pagine presenti in memoria.

Allocazione fissa con dominio locale: A ciascun processo è assegnato un numero fisso di frame: se troppo piccolo, può portare al trashing; se troppo grande, si ha uno spreco della memoria: si riduce il numero di processi presenti in memoria e aumenta la probabilità che la ready list si svuoti.
Allocazione variabile con dominio globale: l SO mantiene una lista di free frame (free list). Al verificarsi di un page fault: un frame viene tolto dalla free list e aggiunto al resident set del processo, se la free list è vuota, il frame può essere tolto ad un altro processo: usando una delle politiche di rimpiazzo (LRU, clock, …).
Allocazione variabile con dominio locale: quando un processo viene caricato in memoria, il SO gli assegna un certo numero di frame, le pagine possono essere assegnate con politica demand paging o prepaging.
Al verificarsi di un page fault, la pagina da rimpiazzare viene scelta tra quelle del processo stesso, periodicamente il SO rivaluta il numero di frame assegnati al processo, usando la **strategia del working set**.

Working Set di un processo: insieme di pagine usate effettivamente dal processo: è costituito dalle W pagine usate nelle ultime Δ unità di tempo in cui il processo è stato in esecuzione.
Si tratta di fare in modo che il Resident Set di un processo (le sue pagine presenti in memoria fisica) coincida con il Working Set: il SO periodicamente rimuove le pagine che non sono più nel working set, il processo può diventare running solo se tutto il suo WS è in memoria.
Problemi del working set: il passato recente non indica il comportamento futuro, tener traccia di W comporta overhead, il valore ottimo di Δ non è noto (e comunque varia nel tempo).
Approssimazioni della strategia WS:

- Page-Fault Frequency: al verificarsi di un page fault, si valuta il tempo trascorso dal page fault precedente: se è toppo piccolo si aggiunge un frame al WS, se è troppo grande si rimuove un frame con use bit=0.
- WS ad intervallo variabile: ad ogni intervallo di tempo, gli use bit del resident set sono azzerati, durante l'intervallo, ad ogni page fault il resident set cresce, al termine dell'intervallo, si rimuovono le pagine con use=0, la durata dell'intervallo (compresa tra un valore minimo e uno massimo) dipende dal numero di page fault (se elevato la durata è più vicina al minimo, se basso al massimo).

Cleaning policy

Decidono quando una pagina modificata deve essere ricopiata da memoria a disco:

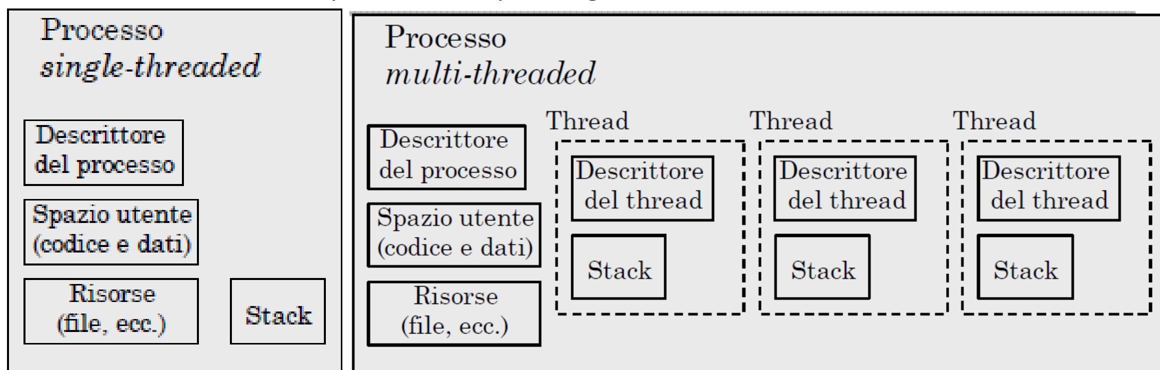
- **Demand cleaning:** la pagina viene ricopiata su disco quando viene selezionata per essere rimpiazzata da un'altra: il processo che ha subito il page fault deve attendere un tempo più lungo (2 accessi al disco) prima di ripartire
- **Precleaning:** la riscrittura su disco delle pagine modificate viene fatta ogni tanto, per gruppi di pagine: se avviene troppo presto, la pagina può venir rimodificata altre volte (e l'accesso al disco per la riscrittura diventa inutile), funziona bene associato all'algoritmo di page buffering: le pagine della modified list ogni tanto vengono riscritte su disco (a gruppi) e inserite nella free list; da qui o ritornano nel resident set (se riusate), oppure vengono rimpiazzate.

Load control
Controllano il livello di multiprogrammazione, decidendo quanti processi tenere in memoria: se sono troppo pochi, aumenta il rischio di ready list vuota, se sono troppi, le dimensioni dei resident set potrebbero diventare insufficienti e provocare il trashing, se è necessario ridurre il livello di multiprogrammazione, va individuato il processo da sospendere (swap-out di tutte le sue pagine)
Sospensione di un processo: La scelta del processo da sospendere per ridurre il livello di multiprogrammazione può essere fatta con diversi criteri:

- Il processo di priorità più bassa (coinvolge lo scheduler)
- Il processo che provoca il maggior numero di page fault: probabilmente il suo WS non è residente e si bloccherebbe presto
- Il processo con il resident set più piccolo: il successivo lavoro per ripristinarlo è il minimo
- Il processo con il resident set più grande: il numero di frame resi liberi è il massimo
- Il processo con il massimo tempo di esecuzione rimanente: per far terminare prima gli altri (che libereranno presto i relativi frame)

5.4. Multithreading

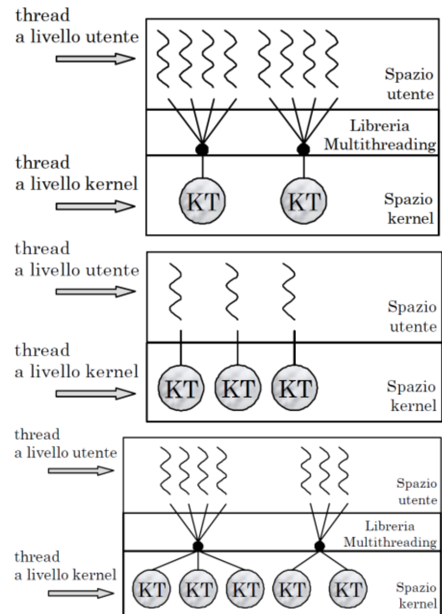
Processo: Uno spazio di indirizzamento virtuale atto a contenerne l'immagine**,** Detentore di risorse**,** Accesso controllato ai processori, ad altri processi, file, risorse di I/O.
Thread: è all'interno di un processo, flusso esecutivo, dotato di stato (running, ready ecc.) e di contesto salvabile, dotato di uno stack, possibilmente dotato di alcune proprie variabili statiche, ha accesso alla memoria e alle risorse del processo contenente.
Il possesso di risorse è riferito ai processi, il dispatching è riferito ai thread.



Benefici del multithreading: creazione e terminazione di un thread sono meno onerose di quelle di un processo, Meno oneroso il context switch tra due thread dello stesso processo, Poiché i thread nello stesso processo condividono automaticamente memoria statica e file, possono più facilmente comunicare.
Problemi del multithreading: Sospendere un processo vuol dire sospendere tutti i suoi thread, i meccanismi di sincronizzazione e comunicazione dovrebbero essere sostanzialmente gli stessi di quelli dei processi.

Modelli realizzativi:
User-level multithreading: I thread sono gestiti da un'apposita libreria, La gestione avviene nello spazio di indirizzamento utente (il S.O. non è consapevole dell'esistenza dei thread)
Kernel-level multithreading: Il multithreading è supportato dal S.O.

Modello molti-a-uno: Molti thread a livello utente vengono fatti corrispondere ad un solo kernel thread, quando il thread kernel si sospende, vengono automaticamente sospesi tutti i thread utente associati.



Modello uno-a-uno: ogni thread a livello utente corrisponde ad un solo kernel thread, le stesse routine del kernel possono essere realizzate in forma multithreaded, overhead di sistema dovuto al passaggio user/kernel mode.

Modello molti-a-molti: un sottoinsieme di thread a livello utente viene fatto corrispondere ad un sottoinsieme, spesso più limitato, di kernel thread, talvolta si vincola qualche thread utente ad essere associato ad uno e un solo thread kernel (modello a due livelli).

Scheduling dei thread: Nei modelli molti-a-uno e molti-a-molti la libreria organizza la competizione dei thread all'interno di un processo, assegnandoli ad uno o più thread kernel, se lo scheduling all'interno di un processo è a priorità con preemption, all'interno di un certo livello di priorità il time-sharing non può essere garantito con precisione dalla libreria. Comunque la competizione effettiva della CPU è globale e coinvolge tutti i thread kernel creati.

Multithreading su multiprocessore: L'assegnazione di un processore ad un thread avviene:

- **Load sharing:** una sola ready list: quando un processore si libera, gli viene assegnato il primo thread ready della coda, problemi sulla gestione di una coda condivisa e sulla coerenza delle cache dei processori.
- **Gang scheduling:** un gruppo di thread correlati viene schedulato nello stesso momento su un sottoinsieme di processori liberi, produce una riduzione delle sincronizzazioni bloccanti e quindi un aumento di performance.
- **Assegnazione ad un processore dedicato:** è un gang scheduling estremo (uno-a-uno), interessante per sistemi ad elevato parallelismo.
- **Scheduling dinamico:** il numero di thread che compongono un processo varia per ottenere un controllo più fine del carico.

6. Gestione I/O

Tecniche di I/O

- I/O programmato: il processore attende in busy waiting che il dato sia stato trasferito
- I/O gestito ad interrupt: il processore comanda una operazione di I/O e poi prosegue ad eseguire un altro processo o thread, il dispositivo interrompe il processore quando il dato è stato trasferito.
- Direct Memory Access (DMA): il processore comanda il trasferimento di un blocco di dati, il dispositivo trasferisce i dati (cycle stealing), il dispositivo interrompe il processore quando tutti i dati sono stati trasferiti.

Obiettivi di gestione:

- **Efficienza**: le operazioni di I/O sono lente (perché lo sono i dispositivi) e il processore non deve perdere tempo per esse.
- **Indipendenza** dal particolare dispositivo: per semplificare la vita al programmatore e per poter usare lo stesso programma con (modelli di) dispositivi diversi.

Efficienza nella gestione

I dispositivi di I/O sono molto più lenti della CPU, la gestione centralizzata di tutto l'I/O da parte del SO (ad interrupt o via DMA) è il presupposto per il multitasking (che consente di usare in modo efficiente la CPU, mandando in esecuzione un altro processo quando un altro richiede una operazione di I/O).
L'I/O può diventare un collo di bottiglia (ready list vuota). Per aumentare il numero di processi nella ready list, spesso vengono effettuati degli swap (in e out), che sono anch'essi operazioni di I/O.
L'efficienza negli accessi al disco è di grande importanza e ad essa viene dedicata particolare attenzione.

Indipendenza dal dispositivo

Si vuole un trattamento uniforme dei dispositivi in riferimento al modo in cui i processi vedono i dispositivi e al modo in cui i dispositivi sono gestiti dal SO. Ma non è facile ottenere questa uniformità, a causa della diversa natura e delle diverse modalità di funzionamento dei dispositivi.
Si cerca di usare un approccio modulare in modo da: nascondere nelle routine di basso livello gran parte dei dettagli dei dispositivi, distinguere i dispositivi fisici, cui sono associate le routine di basso livello, dai dispositivi logici (canali), cui sono associate funzionalità generali di alto livello (open, read, write, close), associare un dispositivo fisico a uno logico e consentire ai processi di interagire con i dispositivi logici.

Driver di I/O

Il software del SO che realizza ciò si chiama driver. Un driver controlla un tipo di dispositivo, ne nasconde i dettagli e consente le operazioni di I/O ai processi tramite comandi di alto livello.
Un driver è solitamente costituito da due parti, che operano in modo asincrono, sincronizzate tramite semafori:

- Una parte, eseguita dal processo utente, nei fatti una subroutine di interfaccia (API), che il processo invoca per richiedere una operazione di I/O (OPEN, CLOSE, READ, ...); le subroutine di interfaccia attivano una routine (DOIO) del SO specificandone i parametri (eventualmente tramite SVC).
- Una seconda parte, eseguita dal SO con l'ausilio di: una routine di servizio delle interruzioni (ISR) e un processo di sistema (Device Handler) che, per rendere veloce l'esecuzione della ISR, la affianca e svolge le operazioni non strettamente urgenti.

Richieste di I/O dei processi

Il processo che intende usare un dispositivo (*disp*) deve:

- Aprire un canale (ch) associato al dispositivo, chiamando una subroutine di interfaccia, ad es. OPEN (*disp*, *ch*, ...), questa subroutine porta il SO ad associare al dispositivo fisico *disp* il dispositivo logico *ch* ed a consentirne l'uso al processo.
- Richiedere le operazioni di I/O, chiamando una subroutine di interfaccia, ad es. WRITE (*ch*, *dati*, ...), questa subroutine porta il SO ad eseguire o avviare la corrispondente operazione;

- Chiudere il canale *ch*, al termine dell'uso del dispositivo, tramite una subroutine di interfaccia ad es. CLOSE(*ch*, ...), il SO elimina le associazioni fatte dalla OPEN.

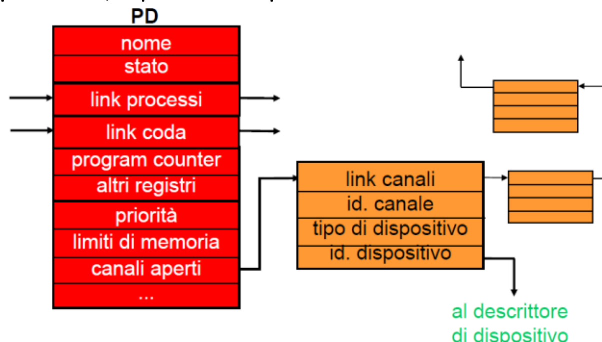
Le subroutine di interfaccia contengono eventualmente una SVC: questa attiva una routine del SO (**DOIO**) che esegue quanto richiesto e può sospendere il processo richiedente.

Attivazione della DOIO:
La DOIO è attivata direttamente (o indirettamente tramite SVC) dalla subroutine di interfaccia, ha il compito di predisporre le strutture di dati che servono per eseguire l'operazione richiesta, sulla base dei parametri ricevuti:
DOIO(ch, mode, count, pointer, sem)

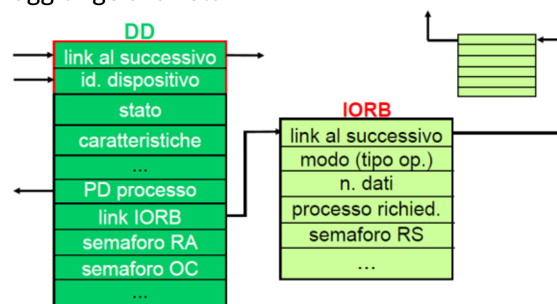
- ch = canale (identificatore di dispositivo logico)
- mode = tipo di operazione (open, read, write, …)
- count = numero di dati da trasferire
- pointer = indirizzo di memoria di origrine o di destinazione del trasferimento
- sem = semaforo di supporto (RS - Richiesta Servita) per segnalare quando l'operazione di I/O sarà completata

Strutture dati usate dal driver
Per eseguire le operazioni di I/O richieste da un processo, **il driver deve disporre di queste informazioni:** il processo richiedente, il dispositivo logico (canale), il dispositivo fisico, l'operazione da eseguire e i semafori per le sincronizzazioni.
Le strutture di dati che contengono queste informazioni sono: il **descrittore di processo** (PD), il **descrittore di dispositivo** (DD) e il **descrittore della richiesta** di I/O (IORB).

Descrittore di processo - PD
OPEN (disp, ch, ...): il SO crea l'associazione tra canale e dispositivo e la lega al descrittore del processo, il processo opera utilizzando ch:



Descrittore di dispositivo e IORB
Il DD contiene informazioni che definiscono le caratteristiche specifiche del dispositivo.
DOIO (ch, mode, ...): completa un IORB e lo aggiunge alla lista:



Funzioni della DOIO

- La routine accede al descrittore PD del processo in esecuzione e, tramite il link *canali aperti*, trova il dispositivo associato a ch e il relativo descrittore DD.
- Verifica la validità dei parametri passati, confrontandoli con le caratteristiche contenute nel DD: se i parametri non sono validi, termina con un'eccezione d'errore (che può provocare la terminazione del processo), altrimenti usa i parametri per costruire l'IORB (I/O Request Block) che descrive la richiesta del processo e lo inserisce nella lista degli IORB (contenuto nel DD).
- Esegue un signal sul semaforo RA (Richiesta Attiva, contenuto nel DD) per indicare che c'è un nuovo IORB nella lista del dispositivo: questo signal sincronizza il Device Handler del dispositivo.
- Esegue un wait sul semaforo RS (Richiesta Servita, contenuto nell'IORB), che provoca la sospensione del processo fino a che quella richiesta non è stata servita.

Struttura della DOIO

```
DOIO (ch, mode, count, pointer, RS) {
    . . .  /* verifica la compatibilità dei parametri con le  caratteristiche
              del dispositivo (DCB); */
    . . .  /* se i parametri non sono validi, abortisce l'operazione
              segnalando l'errore; */
    . . .  /* costruisci un IORB con i parametri ricevuti;
              inserisci l'IORB nella coda di richieste al dispositivo (link
              IORB nel DCB); */
    signal (RA);         // segnala il nuovo IORB al Device Handler
    wait (RS);           // il processo richiedente si sospende ...;
    return               // l'operazione è terminata
```

Struttura del Device Handler

```
while (true) {
   wait (RA)    // attende un signal dalla DOIO (IORB presente)
   . . .        /* preleva un IORB dalla lista associata al
                dispositivo (DD) e avvia l'operazione richiesta */
   wait (OC)    // attende, sul semaforo OC Op Completa (nel DD),
                // che l'operazione sia completata (signal dalla ISR)
   . . .        /* verifica eventuali errori,
                esegue le altre operazioni non eseguite dalla ISR */
   signal (RS)  // il processo sospeso ritorna ready
   . . .        /* restituisce l'IORB alla lista degli IORB liberi */
}
```

Struttura della ISR

```
ISR:
   . . .        // gestione del dispositivo – elimina l'interruzione
inp (pointer)   // effettua l'input, copia il dato nella posizione voluta
pointer ++      // incrementa il puntatore
count  --       // decrementa il contatore
If count =0 then signal (OC)          // simula un DMA
                // avvisa il DH che l'operazione di I/O è completata
terminate       // restituisce il controllo allo scheduler
                /* (che deciderà se riprendere il processo interrotto
                   o mandare in esecuzione il Device Handler) */
```

Struttura del driver

La parte del driver eseguita dal processo utente è costituita dalle subroutine di interfaccia e la routine DOIO attivata da esse.

La parte del driver eseguita dal processo di I/O del SO (sincrona con le operazione del dispositivo) è costituita da: il Device Handler (processo di sistema che gestisce l'I/O del particolare dispositivo) e la routine di servizio delle interruzioni (ISR) del dispositivo.

Le due parti del driver (tra loro asincrone) si sincronizzano tramite i semafori RA ed RS e comunicano tramite le strutture di dati condivise (in particolare la lista concatenata di IORB).

Processo utente

OPEN, READ, CLOSE sono le subroutine di interfaccia (API) del driver del dispositivo D1

```
// processo utente:
. . .
OPEN (D1, ch1);
. . .
READ (ch1, data);      subroutine READ
. . .
CLOSE (ch1);
. . .
```

Subroutine di interfaccia

SVC() è una interruzione software di chiamata a sistema e provoca l'attivazione della routine di servizio DOIO

```
READ (ch1, data);
/* sistema i parametri */
. . .
. . .
SVC();        // chiede al SO di effettuare il READ      DOIO
. . .
return;
```

Routine DOIO

Si sincronizza con il Device Handler (processo di sistema) tramite i semafori: RA (richiesta attiva) e RS (richiesta servita)

```
DOIO (ch, mode, ...);
/* costruisce un IORB e lo inserisce nella lista */
signal (RA);    // segnala il nuovo IORB al Device Handler
wait (RS);      // il processo si sospende in attesa
. . .
return;         // eventualmente terminazione della SVC
```

Device Handler

Effettua le operazioni più complesse (transcodifica, controlli etc.) specifici per il dispositivo, costituisce la parte più importante del driver.

Il DH e la ISR si sincronizzano tramite il semaforo OC

```
while (true) {
      . . .
      wait (RA);       // attende un signal dalla DOIO
      . . .
      wait (OC);       // attende un signal dalla ISR
      . . .
      signal (RS);     // segnala il completamento alla DOIO
}
```

Gestione dell'interruzione - ISR

```
ISR:
   . . .

If count=0 then signal (OC);
                  // avvisa il DH che l'operazione è
  completa
terminate         /* restituisce il controllo allo scheduler */
```

UTENTE	SISTEMA OPERATIVO		
Processo che richiede I/O	**Procedura I/O**	**Device Handler**	**Routine di interruzione**

DOIO(canale, modo, quantità, posizione, semaforo);
------- 	
------- 	
------- 	
------- 	
------- 	
------- 	
------- 	
------- 	
------- 	
richiesta_servita. wait();
controlla posizione di errore;
	identifica dispositivo;
	effettua controlli di errore;
	costruisci un IORB;
	inserisci IORB nella coda di richieste del dispositivo;
	richiesta_attiva. signal();
	uscita.
		loop
		richiesta_attiva. wait();
		preleva IORB;
		inizia operazione I/O;
		operazione_completa.wait();
		effettua controlli di errore;
		completa funzioni associate ad I/O appena concluso;
		metti ordine;
		richiesta_servita. signal();
		cancella l'IORB;
		end loop
			Individua interruzione;
			localizza descrittore di dispositivo;
			operazione_completa.signal()

confine SO

Flusso di controllo

Sincronizzazioni (mediante semafori)

Lancio operazione di I/O (al termine il dispositivo provoca una interruzione)

Sincronizzazione differita mediante interruzione

Driver più snello

Per i dispositivi più semplici, per i quali le operazioni da eseguire a seguito di una interruzione sono leggere, si può rendere più snella la struttura del driver eliminando il Device Handler e facendo svolgere le sue funzioni in parte alla DOIO e in parte alla ISR.

In questo caso la DOIO si sincronizza direttamente con la ISR (basta il solo semaforo RS):

- La DOIO, costruisce l'IORB, lo mette a disposizione della ISR, avvia anche l'operazione; poi attende: wait(RS)
- La ISR verifica eventuali errori, esegue tutte le operazioni richieste e poi segnala: signal(RS)

Bufferizzazione

Per aumentare l'efficienza del sistema di I/O il SO prevede alcune aree di memoria (buffer) in cui trasferire i dati di input provenienti dai dispositivi (prima di trasferirli ai processi utente) e i dati di output (prima di trasferirli ai dispositivi):

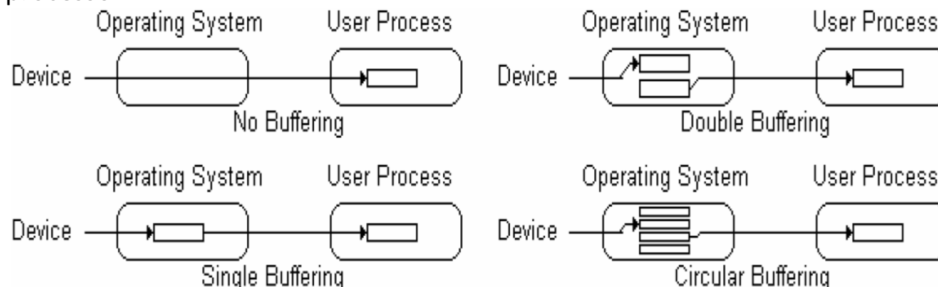
- Si riduce l'overhead dovuto alle sincronizzazioni.
- Disponendo di dispositivi DMA le operazioni di I/O sono più veloci.
- Con buffer multipli i dispositivi operano con maggior efficienza, il dispositivo può trasferire un blocco di dati mentre il processo ne può elaborare un altro in parallelo.
- Il processo può subire uno swap out, perché il trasferimento coinvolge un frame del sistema: lo swap out di un processo mentre è in corso un trasferimento in una sua pagina creerebbe problemi seri.
- Il SO deve tener traccia dei buffer assegnati ai processi.

Tipi di bufferizzazione:

- A blocchi: i buffer sono di dimensione fissa, ad es. di un frame, i trasferimenti avvengono per blocchi, viene usato per dischi, nastri, …
- Stream: i trasferimenti avvengono per flussi di caratteri, usato per terminali, stampanti, porte seriali, mouse.

Numero di buffer:

- Buffer singolo: il processo può elaborare un blocco di dati mentre l’altro viene trasferito
- Doppio buffer: il processo può trasferire dati da un blocco mentre il SO sta svuotando o riempiendo l’altro, diminuiscono i tempi di attesa del processo.
- Bufferizzazione circolare o un buffer multiplo (pool): ciascun buffer è un elemento di una coda circolare o di una lista, consente di far fronte a picchi (temporanei) di operazioni di I/O effettuate da un processo



Bufferizzazione per input a caratteri

Talvolta la ISR gestisce un dispositivo di input a caratteri (byte o interi) il cui rate di produzione è irregolare (ad esempio tastiera): logicamente di tipo stream, il buffer può essere organizzato come una coda circolare di caratteri, se la ISR riempie il buffer, vengono persi caratteri (gli ultimi oppure i più vecchi), la ISR NON può sospendersi, quindi non può attendere lo svuotamento del buffer da parte del processo utente o del DH.

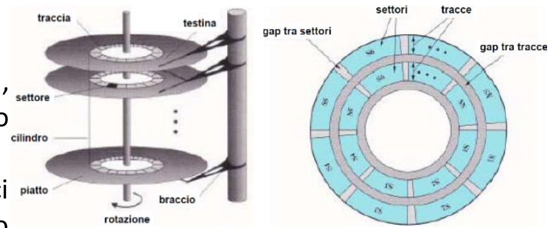
Spooling

Un processo che richieda l’uso di un dispositivo non condivisibile deve attendere che si liberi, nel caso esso sia già in uso

Per i dispositivi di I/O (principalmente di output), per evitare queste attese, il SO utilizza la tecnica di spooling (Simultaneous Peripheral Operation On Line), che consiste nell’assegnare, ad ogni processo che lo richieda, un dispositivo virtuale, realizzato da un file su disco:

- Ciascun processo ottiene, senza attesa, l’accesso al file che rappresenta il dispositivo (non condivisibile)
- Le richieste dei processi di trasferire dati al dispositivo vengono trasformate in trasferimenti sul file assegnato
- Essendo i dischi veloci, l’I/O è spesso più rapido
- Quando un processo chiude il proprio dispositivo virtuale, il suo file viene inserito nella coda dei file che saranno trasferiti al dispositivo fisico, uno alla volta, da un processo di sistema (Spooler)
- La coda dei file si mantiene da un’attivazione all’altra del sistema

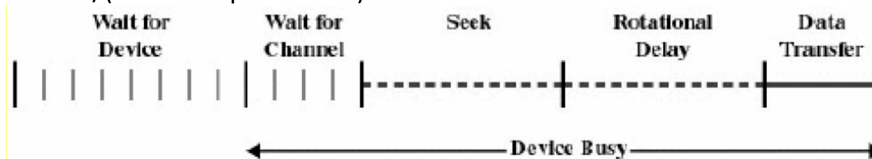
7. Gestione memoria secondaria



Piatto: un disco rigido si compone di uno o più dischi paralleli, di cui ogni superficie, detta "piatto" e identificata da un numero univoco, è destinata alla memorizzazione dei dati.
Traccia: ogni piatto si compone di numerosi anelli concentrici numerati, detti tracce, ciascuna identificata da un numero univoco.
Cilindro: l'insieme di tracce alla stessa distanza dal centro presenti su tutti i dischi o piatti è detto cilindro. Corrisponde a tutte le tracce aventi il medesimo numero, ma diverso piatto.
Settore: ogni piatto è suddiviso in settori circolari, ovvero in "spicchi" radiali uguali ciascuno identificato da un numero univoco.

Caratteristiche

I dischi sono molto più lenti della memoria centrale: tempo di accesso tipico per i dischi: 10ms, per la memoria: 10ns
Gli accessi al disco sono spesso molto frequenti: richieste dei processi richieste del SO (swap di pagine della memoria virtuale). Quindi le prestazioni complessive del sistema dipendono dall'efficienza dell'I/O su disco.
Tempi di accesso al disco è la somma di: l'attesa che il dispositivo venga assegnato al processo, l'attesa che il canale (se usato da più dischi) sia disponibile, il tempo di seek t_S(tempo per posizionare la testina sulla traccia), il tempo di rotazione t_R, il tempo per il trasferimento dei dati, mentre il settore scorre sotto la testina: per un settore tsett= TR/(n° settori per traccia):



7.1. Schedulazione degli accessi al disco

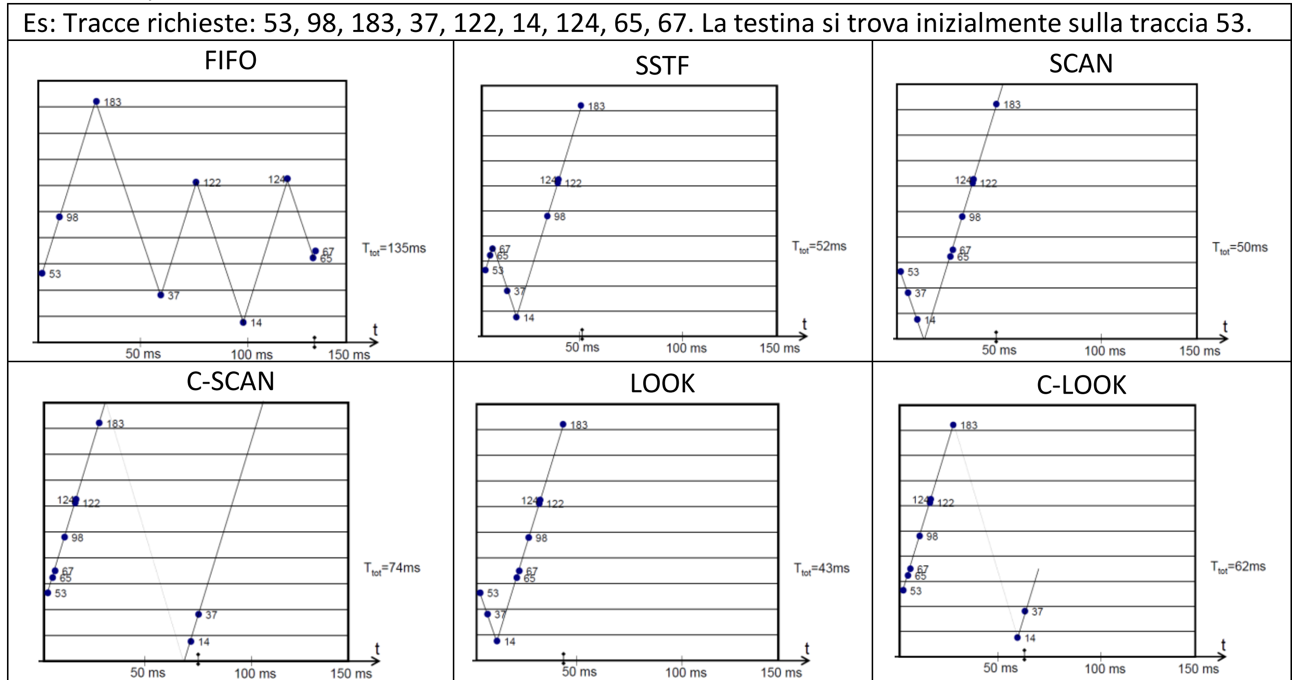
Nei sistemi multitask, le richieste di accesso al disco sono di tipo casuale, a causa dei tempi di seek questo sono molto più lente di quelle sequenziali, quindi ci sono politiche che trasformano le sequenze di richieste casuali, in sequenze di accessi ordinati.

- FIFO
- SSTF (Shortest Seek Time First)
- SCAN (C-SCAN)
- LOOK (C-LOOK)

FIFO: le richieste di accesso vengono evase nell'ordine con cui sono pervenute. Prestazioni molto scarse.
SSTF: viene scelta ogni volta come richiesta successiva da evadere, quella situata sulla traccia più vicina alla posizione attuale della testina.
Tempo complessivo è più breve del FIFO. Però, se continuano a pervenire richieste per tracce vicine alla attuale, può provocare starvation delle richieste più lontane.
SCAN: le richieste vengono evase secondo l'ordine con cui vengono incontrate dalla testina, che si sposta nella stessa direzione da una estremità all'altra del disco e inverte la direzione solo quando ha raggiunto l'estremità.
Evita il rischio di starvation ed ha una buona efficienza. Però favorisce le richieste che interessano le tracce situate vicino alle estremità delle superfici del disco (vengono scandite due volte, avanti e indietro, in tempi ravvicinati), e anche le richieste arrivate per ultime.
Il primo dei due problemi è risolto con la variante **CSCAN:** le richieste vengono evase solo mentre la testina sta procedendo in una direzione (il ritorno all'altra estremità avviene senza accessi al disco) ma è anche meno efficiente.
Il secondo con la variante **N-step SCAN**: le richieste sono ripartite in sottocode di lunghezza N, viene elaborata, una sottocoda alla volta: se N = 1 si ricade nel caso FIFO, se N è grande le prestazioni sono simili allo SCAN. Le nuove richieste sono inserite in una sottocoda diversa da quella in elaborazione, per cui non possono passare avanti.

LOOK e C-LOOK: l'ordine di evasione delle richieste è lo stesso del caso SCAN, ma la testina inverte la direzione quando ha raggiunto l'ultima richiesta in quella direzione. Stesse caratteristiche dello SCAN e dello C-SCAN ma più efficienti.

Es: Tracce richieste: 53, 98, 183, 37, 122, 14, 124, 65, 67. La testina si trova inizialmente sulla traccia 53.

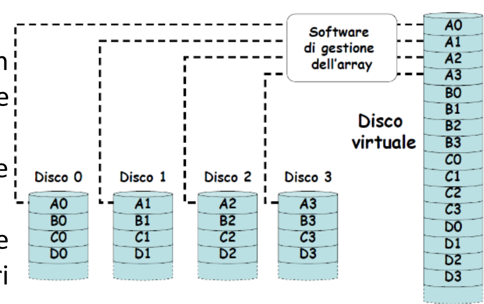


7.2. Sistemi RAID

RAID (Redundant Array of Inexpensive Disks) è uno standard per memorizzare dati su più dischi: un insieme di dischi fisici viene visto dal SO come un disco logico unico (meno costoso, più affidabile), i dati sono distribuiti su più dischi fisici, si sfrutta l'elevata capacità per memorizzare informazioni di ridondanza (parità) che consentono di recuperare i dati in caso di guasto.

Lo standard RAID prevede 7 livelli diversi (da 0 a 6), che indicano organizzazioni diverse con diverse funzionalità. RAID 2 e 4 non sono usati nei sistemi commerciali.

RAID 0 (non ridondante):



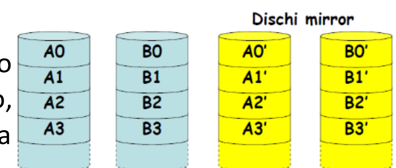
Usa la tecnica di disk striping: i dati di ciascun file sono suddivisi in stripe (strisce: che possono essere byte, settori o blocchi), stripe consecutivi sono collocati, a rotazione, su dischi diversi.

Una richiesta di I/O è gestita accedendo in parallelo ai diversi stripe collocati su dischi diversi.

Per applicazioni che richiedono alte velocità di trasferimento: serve un system bus veloce e conviene avere stripe di dimensioni inferiori alle richieste, per gestire queste ultime in parallelo su più dischi.

Per applicazioni che richiedono tempi di risposta brevi: una richiesta riguarda pochi dati: i tempi di I/O sono dominati dai tempi di seek e di rotazione, conviene avere stripe di dimensioni superiori alle richieste, così ciascuna richiesta è soddisfatta da un singolo disco.

RAID 1 (mirroring):

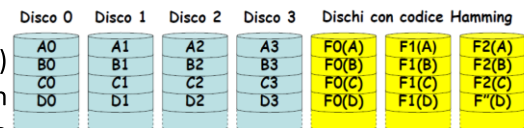


Consiste nella duplicazione dei dati (mirroring), le operazioni di lettura di uno stripe sono effettuate su uno qualsiasi dei due dischi che lo contengono, quelle di scrittura sono effettuate in parallelo sui due dischi (va aggiornata la copia) e quindi limitate dalla velocità del più lento dei due.

In caso di guasto di un disco, l'esecuzione del processo può continuare usando la copia:

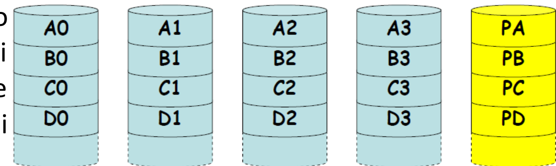
È una forma di backup in tempo reale, raddoppia il costo, se le richieste sono in maggioranza di lettura, presenta anche vantaggi rispetto al RAID 0: la velocità di trasferimento può raddoppiare. La frequenza delle richieste può raddoppiare perchè c'è il doppio di dischi disponibili per gli accessi. Se le richieste sono di scrittura, le prestazioni sono le stesse del RAID 0.

RAID 2 (codice di Hamming):



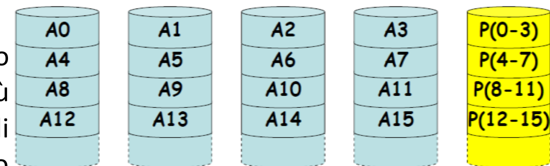
Usa un ECC (Error Correcting Code) di N bit (codice di Hamming) per rilevare/correggere gli errori negli stripe registrati in posizioni corrispondenti, servono N dischi in più per contenere questo codice. Consente la correzione di singoli bit errati e il rilevamento di errori su più bit.
Non è più usato nei sistemi commerciali.

RAID 3 (bit di parità):



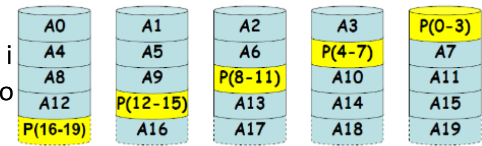
La dimensione degli stripe è piccola (byte o word), l'accesso parallelo ai dischi è sincronizzato (le testine di tutti i dischi si trovano nella stessa posizione), c'è un disco in più che contiene i bit di parità calcolati sui bit nella stessa posizione di tutti gli altri dischi.
Se un disco va fuori uso, il suo contenuto può essere ricostruito calcolando l'XOR degli stripe corrispondenti degli altri dischi. Non può eseguire richieste simultanee di I/O. Estremamente raro in pratica.

RAID 4 (parità a livello di blocco):



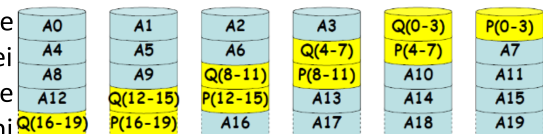
Ciascun disco opera indipendentemente (le testine non sono nella stessa posizione), per cui possono essere servite più richieste di lettura in parallelo, gli stripe sono di grandi dimensioni (blocchi di byte) e per ciascun blocco viene creato un blocco contenente i bit di parità, memorizzato su un disco di parità.
Ogni operazione di scrittura richiede 4 accessi: due di lettura e due di scrittura, il disco di parità subisce 2 accessi per ogni operazione di scrittura. Non è usato nei sistemi commerciali.

RAID 5 (parità distribuita):



A differenza del RAID 4, gli stripe di parità sono distribuiti su tutti i dischi (tipicamente secondo uno schema circolare) evitando il collo di bottiglia sul disco di parità presente nel raid 4.

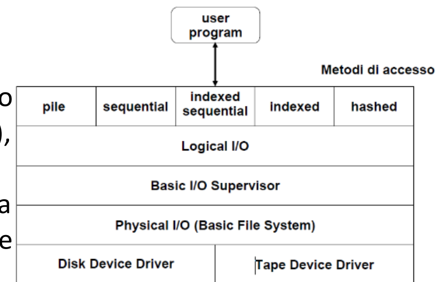
RAID 6 (parità doppia):



Aumenta la ridondanza rispetto ai livelli precedenti, usa due diversi algoritmi di parità: P (basato sull'OR esclusivo usato nei livelli 4 e 5) e Q (un algoritmo di check diverso), i risultati dei due algoritmi sono memorizzati in due blocchi diversi su due dischi diversi.
Consente di rigenerare i dati anche nel caso di due dischi fuori uso, l'operazione di scrittura deve aggiornare anche due blocchi di parità. Adatto per le applicazioni in cui è importante una elevata affidabilità dei dischi.

8. File System

Obiettivi: fornire gli strumenti di memorizzazione e gestione dei dati richiesti dalle applicazioni, ottimizzare le prestazioni, gestire dispositivi di memoria secondaria, evitare la perdita dei dati, fornire di I/O, consentire l'accesso contemporaneo a più applicazioni.



Architettura di un file system

Device drivers: comunicano direttamente con i dispositivi, gestiscono l'avvio delle operazioni di I/O sui dispositivi (API, DOIO, Device Handler), intervengono alla fine delle operazioni di I/O (ISR, Device Handler).
I/O fisico: gestisce il trasferimento di blocchi fisici (settori) di dati tra disco e memoria: dove collocare i blocchi su disco: (d,c,t,s) ed eventuale bufferizzazione dei blocchi in memoria centrale.
Supervisione dell'I/O fisico: traduce gli indici dei blocchi logici (0..N) negli indirizzi fisici (d,c,t,s) dei corrispondenti blocchi su disco, gestisce i blocchi liberi e li assegna ai file e opera la schedulazione degli accessi per migliorare le prestazioni.
I/O logico: consente alle applicazioni utente di accedere ai record di un file organizzandone la divisione in blocchi.

Organizzazione logica del disco

Partizionamento: consente la coesistenza di file system diversi sullo stesso disco fisico (SO diversi), ciascuna partizione si presenta al SO come fosse un disco fisico diverso.
Formattazione: in ciascun settore del disco viene scritta un'intestazione contenente il numero del settore stesso, viene anche predisposto lo spazio per l'ECC (error correcting code): al momento della scrittura di un settore, l'hardware di controllo del disco inserisce nell'EEC un valore calcolato come funzione del valore di ciascun byte scritto nel settore stesso; al momento della lettura del settore, l'ECC viene ricalcolato e, se il valore è diverso, viene rilevato un errore (il settore potrebbe essere danneggiato e i dati letti non sono affidabili).
Cluster: sono l'unità minima di spazio su disco (costituita da N settori contigui) assegnata ad un file.
FAT (File Allocation Table): è una tabella usata dal S.O. per localizzare i cluster assegnati a ciascun file.
FAT con elementi da 32-bit (FAT32) consente partizioni da 2TB (2^{41}) con cluster piccoli (8 KB) (sono usati solo 28 dei 32 bit: $2^{41}/2^{28} = 2^{13}$= 8K). NTFS usa cluster da 0.5 a 4 KB (1..8 settori).
Blocco di boot: all'avvio di un calcolatore, viene eseguito il bootstrap che serve a: inizializzare le diverse componenti del sistema, dai registri della CPU ai controller dei dispositivi, caricare in memoria da disco il SO e ad avviarlo.
Il bootstrap è costituito da: una piccola parte situata su ROM, una seconda situata su disco in una posizione prefissata, spesso a partire dal blocco 0 (blocchi di boot).
Il codice contenuto nella ROM comanda il trasferimento su RAM dei blocchi di boot e ne avvia l'esecuzione, mentre il codice su disco è più complesso ed è in grado di caricare l'intero SO e di avviarne l'esecuzione.

Gestione dello spazio di swap

Nei SO con MV, una parte del disco è utilizzata come area di swap: in alcuni sistemi in essa viene memorizzata copia dell'intera immagine dei processi in esecuzione, nei sistemi paginati essa è destinata a contenere le pagine scaricate dalla memoria centrale.
La dimensione dello spazio di swap incide notevolmente sulle prestazioni del sistema, alcuni sistemi (UNIX) prevedono più aree di swap situate su dischi separati, in modo da distribuire su di essi il carico di I/O dovuto alle operazioni di paginazione
L'area di swap può essere situata in un file e gestita tramite il file system o situata in una partizione apposita (più veloce e efficiente).

Disk cache
Il meccanismo è simile al cache di memoria: mantiene in memoria centrale copia dei settori del disco usati più frequentemente, se i dati si trovano nel cache, vengono usati lì, altrimenti si leggono da disco (e se ne mette copia nel cache), poiché le operazioni hanno tempi enormemente più lunghi, gli algoritmi di rimpiazzo (LRU – Least Recently Used, LFU - Least Frequently Used) sono più sofisticati e complessi.
Il disk cache può essere ricavato usando parte della memoria centrale (è in sostanza la bufferizzazione dell'I/O su disco): in questo caso è gestita dal SO, oppure può essere ricavato usando chip di memoria situati presso il drive del disco e gestiti dall'hardware di controllo del disco: in questo caso è trasparente al SO.

Metodi di accesso: forniscono una interfaccia standard tra le applicazioni utente e il file system, quello più usato è quello con indice ovvero un file indice per ogni tipo di campo da ricercare nei record.
File directory (cartelle): sono file cui è proprietario il S.O. che contengono informazioni sui file memorizzati in un disco, i processi utente di solito possono accedervi solo tramite apposite system call. Costituiscono un indice che consente di accedere ai file a partire dai loro nomi simbolici.
Operazioni sui directory (tramite apposite system call): la ricerca, creazione, cancellazione di un file e l'ottenimento della lista dei file contenuti.
Strutture dei directory:

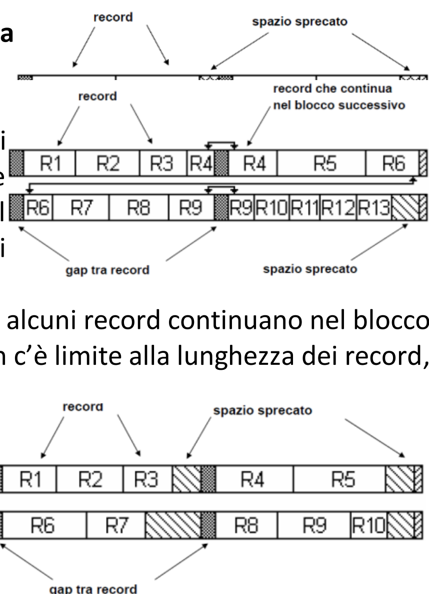
- A un livello: un file sequenziale che contiene una lista di record (uno per ogni file contenuto nel directory), usato nei primi SO per PC, difficile gestire un numero elevato di file o più utenti
- A due livelli: un directory per ciascun utente, puntati da uno principale ma non consente agli utenti di organizzare bene i propri file
- **Ad albero:** un directory principale (root), ciascun directory può contenere file (normali) o altri directory (subdirectory)

Pathname nei directory ad albero: unicità dei nomi solo all'interno di ciascun directory: pathname relativi (al directory di lavoro) e pathname assoluti (a partire dal directory root).
Condivisione dei file da più processi:
Diritti di accesso al file che possono essere assegnati ai diversi processi:

- Nessuno: l'esistenza del file non è nota (ad altri processi)
- Semplice conoscenza: il file può essere visto
- Esecuzione: il file (programma) può essere eseguito
- Lettura: il contenuto del file può essere letto
- Modifica: il contenuto del file può essere scritto
- Modifica dei diritti di accesso
- Cancellazione

Va garantita la mutua esclusione (lock) nell'accesso dei processi (a un singolo record, o all'intero file).

Bloccaggio dei record (Non attuale, si usa suddivisione a settori, ma utilizzato nei database):
I record vanno inseriti nei blocchi fisici (settori o cluster):
Bloccaggio a lunghezza fissa: Possibile con record di lunghezza fissa, ogni blocco contiene un numero intero NR di record, spreco di spazio alla fine del blocco (frammentazione interna). È semplice trovare l'indice IB del blocco che contiene un record, dato il suo indice IR (IB = parte intera di IR/ NR).
Bloccaggio a lunghezza variabile con superamento del limite di blocco: alcuni record continuano nel blocco successivo, l'accesso a questi record richiede 2 accessi (ai 2 blocchi), non c'è limite alla lunghezza dei record, spreco di spazio solo alla fine del file.
Bloccaggio a lunghezza variabile senza superamento del limite di blocco: tutti i record sono interamente contenuti in un solo blocco, se lo spazio libero in un blocco non è sufficiente, il record viene messo tutto nel blocco successivo, spreco di spazio nella gran parte dei blocchi, la dimensione di un record non può superare quella dei blocchi.



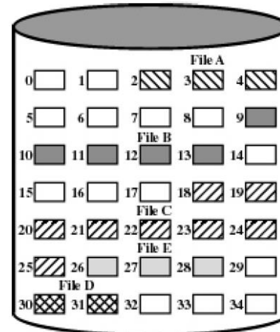
Allocazione dei blocchi

Un file è costituito da un insieme di blocchi del disco, due alternative per la dimensione dei blocchi:

- Lunghezza variabile e grande: efficienza negli accessi contigui, la File Allocation Table (FAT) è piccola, frammentazione esterna nel riuso dei blocchi.
- Lunghezza fissa e piccola: nessuna contiguità, la File Allocation Table (FAT) è grande, più semplice allocare e riusare (frammentazione interna).

Metodi per allocare un blocco a un file:

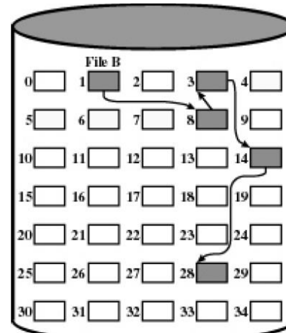
Allocazione contigua: lo spazio su disco viene assegnato ai file quando sono creati, metodo usato nei CD-ROMs (ISO 9660). Vantaggi: FAT semplice: un elemento per ciascun file, contiene: la posizione del blocco iniziale del file e il numero di blocchi del file, è semplice trovare la posizione di un blocco, efficiente la lettura di blocchi contigui (disk cache). Svantaggi: la dimensione massima del file va dichiarata prima, frammentazione esterna nel riuso dei blocchi liberi, necessità di compattarli.



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Allocazione concatenata: i singoli blocchi del disco vengono assegnati quando servono. Vantaggi: FAT semplice: per ciascun file c'è un link al primo blocco (ciascun blocco contiene un link al successivo), è facile aggiungere blocchi, non c'è frammentazione esterna. Svantaggi: per trovare un blocco bisogna percorrere tutta la catena, metodo adatto ai file sequenziali, non ci sono i vantaggi delle letture contigue, per migliorare l'efficienza, periodicamente si consolidano i file rendendoli contigui. La FAT 12/16/32 di MSDOS è una variazione di questo metodo.

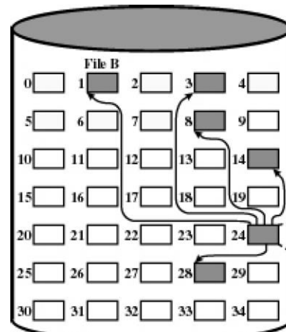


File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Allocazione indicizzata: risolve sia i problemi del metodo contiguo (frammentazione esterna e pre-dichiarazione delle dimensioni), sia quelli del metodo concatenato (scarsa efficienza nell'accesso diretto).

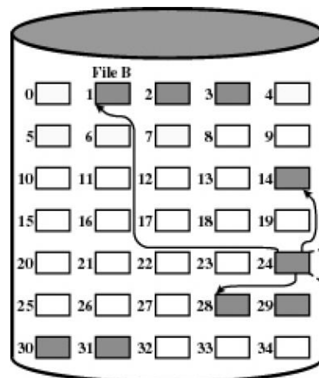
I singoli blocchi del disco vengono assegnati quando servono, possono essere di dimensioni fisse o variabili, la FAT contiene, per ciascun file, un link ad un blocco-indice: il blocco-indice contiene i puntatori ai blocchi del file. Adatto sia ad accessi sequenziali sia ad accessi diretti, è la forma di allocazione più comune, UNIX usa una variante di questo metodo.



File Allocation Table

File Name	Index Block
• • •	• • •
File B	24
• • •	• • •

1
8
3
14
28



File Allocation Table

File Name	Index Block
• • •	• • •
File B	24
• • •	• • •

Start Block	Length
1	3
28	4
14	1

Gestione dei blocchi liberi
Per allocare blocchi ai file il SO deve sapere quali sono i blocchi liberi: serve una DAT (Disk Allocation Table), oltre a una FAT.
Le tecniche comunemente usate sono:

- Tabella di bit (bit table): ad ogni blocco corrisponde un bit (1=occupato, 0=libero), occupa poco spazio su disco (copia in memoria), ricerca esaustiva per trovare i blocchi da allocare, il disco può essere diviso in sezioni con una tabella sommario (n. max blocchi liberi contigui, ...) per ciascuna.
- Lista concatenata: in ogni gruppo contiguo di blocchi liberi ci sono 2 campi: il numero di blocchi liberi del gruppo e un link al gruppo successivo, non occupa spazio su disco (in memoria link alla testa), l'overhead di tempo può essere pesante.
- Tabella indice: i blocchi liberi sono visti come appartenenti ad un file, nella FAT c'è il link al suo blocco indice che ne individua i blocchi (di dimensione variabile).
- Lista dei blocchi liberi: ciascun blocco su disco è individuato da un indice (di 24 o 32 bit a seconda della capacità del disco), su disco viene tenuta una lista degli indici dei blocchi liberi (24 o 32 volte più grande della bit table), in memoria viene tenuta solo una piccola parte della lista: i blocchi liberati più recentemente (LIFO) e quelli liberi da più tempo (FIFO), un processo di background mantiene ordinata la lista per consentire la individuazione dei blocchi contigui

Per gestire i blocchi liberi e assegnarli ai file il SO usa 2 tabelle: DAT (Disk Allocation Table) per rintracciare i blocchi liberi e FAT (File Allocation Table) per i blocchi assegnati ai file.
Le tabelle possono essere tenute:

- In memoria: le tabelle potrebbero essere troppo grandi, in caso di crash le informazioni verrebbero perdute
- Su disco: per allocare un blocco ad un file è necessario accedere più volte al disco, per leggere e modificare DAT e FAT il sistema rallenta in modo che può diventare intollerabile.

Gestione dei crash
Per gestire i crash di sistema senza perdita di dati, bisogna: operare un lock sulla DAT per accedervi in mutua esclusione, trovare (nella sua copia in memoria) i blocchi liberi richiesti, impegnare i blocchi nella DAT e aggiornare la copia su disco, aggiornare anche la FAT, sia in memoria, sia su disco, operare l'unlock della DAT per consentirne l'uso ad altri.
Questa serie di operazioni può rallentare molto il sistema, per migliorare l'efficienza si può: preallocare i blocchi liberi a lotti (impegnandoli nella DAT), assegnarli ai file quando vengono richiesti (aggiornando la FAT in memoria e su disco), senza accedere alla DAT, esaurito il lotto, se ne prealloca un altro e in caso di crash possono risultare in uso (nella DAT) alcuni blocchi non assegnati e allora sarà necessario ripulire la DAT (verificando nella FAT se i blocchi sono davvero in uso).

8.1. File system UNIX

Gestione dei file in Unix

Unix vede tutti i file come una sequenza di byte (la struttura logica è gestita dalle applicazioni) e distingue 4 tipi di file:

- File ordinari
- File **directory** (per ciascun file ne contengono il nome e un puntatore al relativo i-node)
- File **speciali** (associati ai dispositivi di I/O: gestiti a caratteri o a blocchi)
- **Pipe** con nome (buffer circolari in memoria che consentono la comunicazione tra processi con il meccanismo del produttore-consumatore)

Un file può avere più nomi (link), che possono essere riferiti al directory corrente (pathname relativi es: subd/pippo.txt) oppure essere riferiti al directory radice (pathname assoluti es: /usr/local/pippo.txt).
Ogni file ha un utente possessore che appartiene ad un determinato gruppo.
Si possono impostare i permessi di accesso (r-read w-write x-execute) riservati ai processi che appartengono al possessore, ad un altro utente del gruppo del possessore, a tutti gli altri utenti.

Chiamate Posix per la gestione di file e directory

File e directory	Descrizione
`fd=creat(name, mode)`	Crea un nuovo file
`fd=open(file, how)`	Apre un file per la lettura e/o scrittura
`s=close(fd)`	Chiude un file aperto
`n=read(fd, buffer, nbytes)`	Legge i dati da un file nel buffer
`n=write(fd, buffer, nbytes)`	Scrive i dati da un buffer nel file
`pos=lseek(fd, offset, whence)`	Muove il puntatore nel file
`s=stat(name, &buf)`	Restituisce l'informazione relativa al file
`s=mkdir(name, mode)`	Crea una nuova directory
`s=link(name1, name2)`	Da un nome aggiuntivo ad un file esistente
`s=rmdir(name)`	Cancella una directory vuota
`s=unlink(name)`	Rimuove un nome ad un file
`s=chdir(dirname)`	Cambia la directory di lavoro
`s=chmod(name, mode)`	Cambia i bit di protezione del file

i-node (64 bytes):

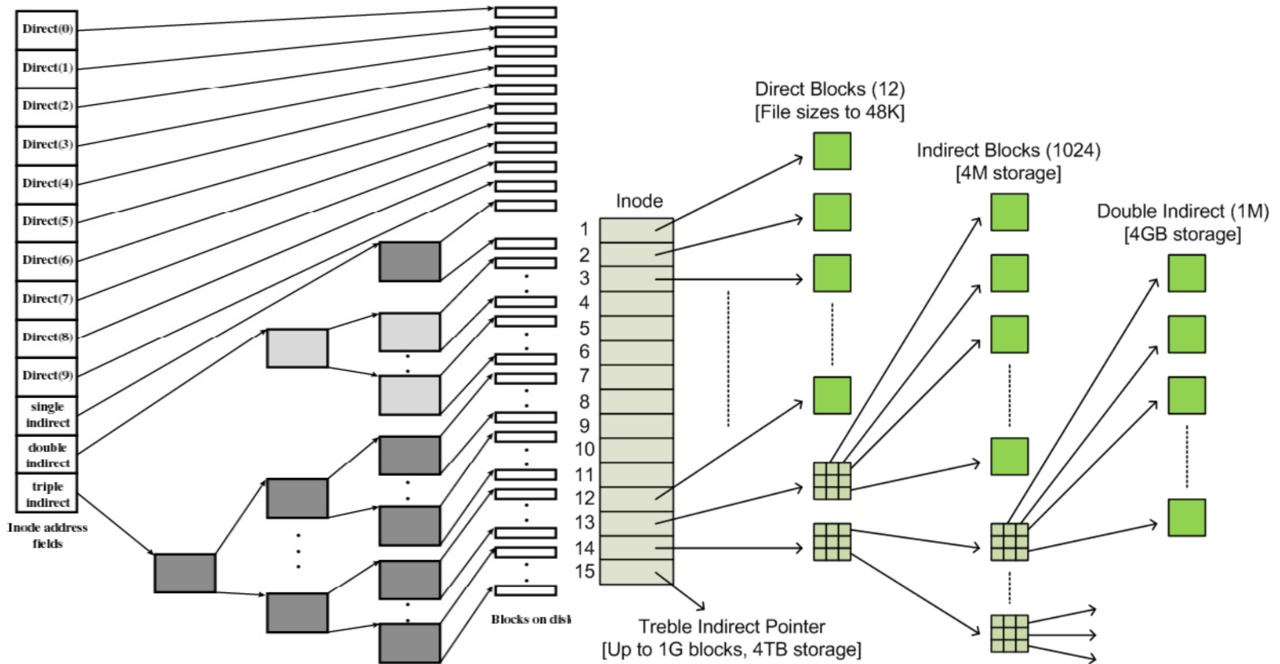
In Unix ad ogni file è associato un i-node (index node) che contiene le informazioni per la gestione del file:

- **mode** (tipo di file e diritti di accesso)
- **link count** (numero di directory che hanno un link all'i-node)
- **owner id** (chi è il proprietario)
- **group id** (gruppo associato al proprietario)
- **file size** (dimensione del file, in byte)
- **last accessed** (data e ora dell'ultimo accesso al file)
- **last modified** (data e ora dell'ultima modifica al file)
- **i-node modified** (data e ora dell'ultima modifica all'i-node)
- **13 elementi** dell'indice di allocazione

Nell'i-node non c'è il nome del file: un file può avere più nomi che sono contenuti nei directory.
In Unix **l'allocazione dei blocchi ai file è di tipo indexed**, la parte iniziale del blocco-indice è contenuta direttamente nell'i-node, il resto in altri blocchi con indici a più livelli:

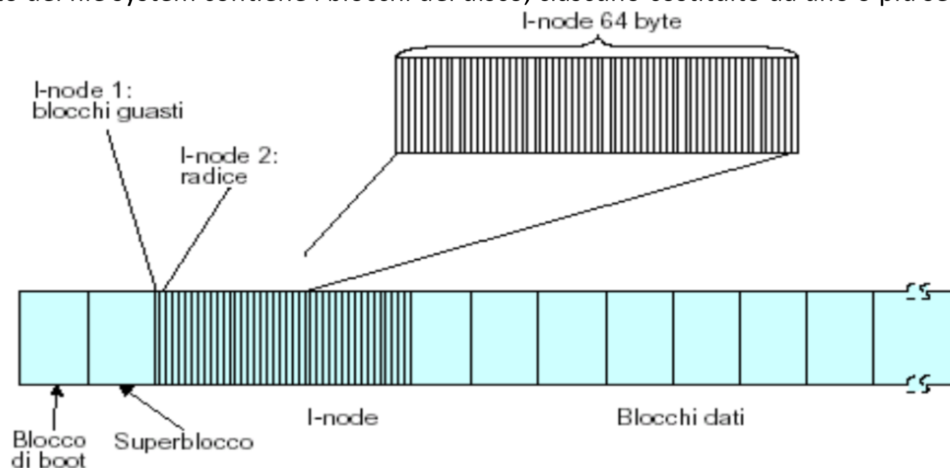
- **I primi 10 elementi** dell'indice di allocazione contengono gli indirizzi diretti di altrettanti blocchi (se il file non supera i 10 blocchi, il suo indice è tutto contenuto nell'i-node).
- **L'11°** elemento punta ad un blocco che contiene il resto dell'indice (indice indiretto): con blocchi da 1KB e indici da 4B, questo blocco-indice individua altri 256 blocchi (256KB).
- Per file ancora più grandi, il **12°** elemento punta a un blocco di indicizzazione indiretta doppia (punta a 256 blocchi di indicizzazione indiretta semplice) per altri 64K blocchi (64MB).
- Per file ancora più grandi, il **13°** elemento punta a un blocco di indicizzazione indiretta tripla (punta a 256 blocchi di indicizzazione indiretta doppia) per altri 16M blocchi (16GB).

Ai file piccoli (la maggior parte) si accede con poche indirezioni. La dimensione massima gestibile è sufficiente a soddisfare tutte le applicazioni.



Struttura del file system

- Il primo blocco è il blocco di boot
- Il blocco successivo contiene il superblocco, con la (prima parte della) lista dei blocchi liberi e la cache degli i-node liberi
- I successivi blocchi contengono gli i-node (uno per ciascun file)
- Il resto del file system contiene i blocchi del disco, ciascuno costituito da uno o più settori



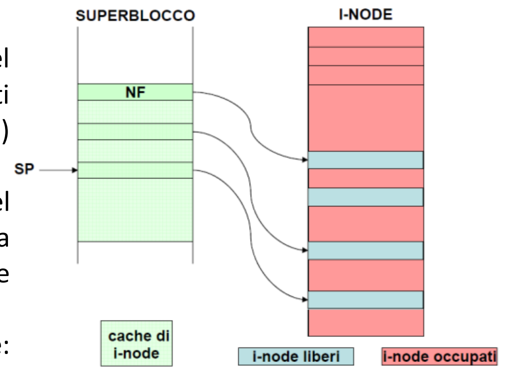
Superblocco

È un blocco che contiene le informazioni che servono per allocare i-node e blocchi:

- La dimensione del file system (n. di blocchi e n. di i-node)
- Il numero di blocchi liberi
- Il primo blocco della lista dei blocchi liberi
- L'indice del blocco successivo della lista dei blocchi liberi
- Il numero di i-node liberi
- Una cache di indici di i-node liberi, gestita a stack
- L'indice del successivo i-node libero non presente nella cache degli i-node liberi

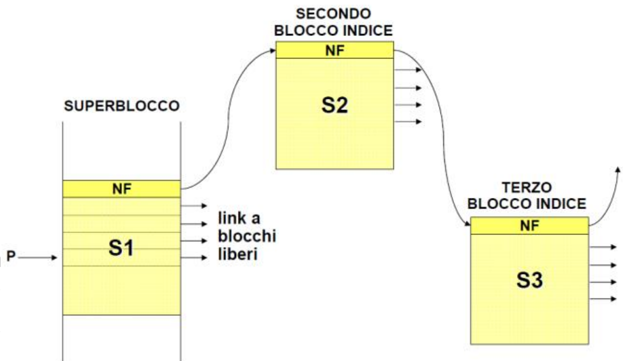
Il kernel mantiene il superblocco in memoria e ne aggiorna periodicamente (ad es. ogni 30 s) la copia su disco.

Allocazione degli i-node



Per allocare un i-node, il SO ne estrae l'indice dalla cache del superblocco: questa cache è organizzata a stack: vengono inseriti (push) gli indici degli i-node che diventano liberi ed estratti (pop) quelli che vengono allocati.
Se la cache è vuota: segue l'indice NF (situato nel superblocco) del successivo i-node libero e ricerca (nella sequenza di i-node situata su disco di seguito al superblocco) nuovi i-node liberi fino a riempire di nuovo la cache.
NF individua sempre il primo i-node libero nella sequenza di inode: quelli inseriti nella cache vengono preallocati (marcati "in uso").
Per liberare un i-node, il S.O. ne inserisce l'indice (push) nella cache; se non c'è posto ne confronta l'indice con NF e, se minore, pone NF uguale a quell'indice.

Allocazione dei blocchi



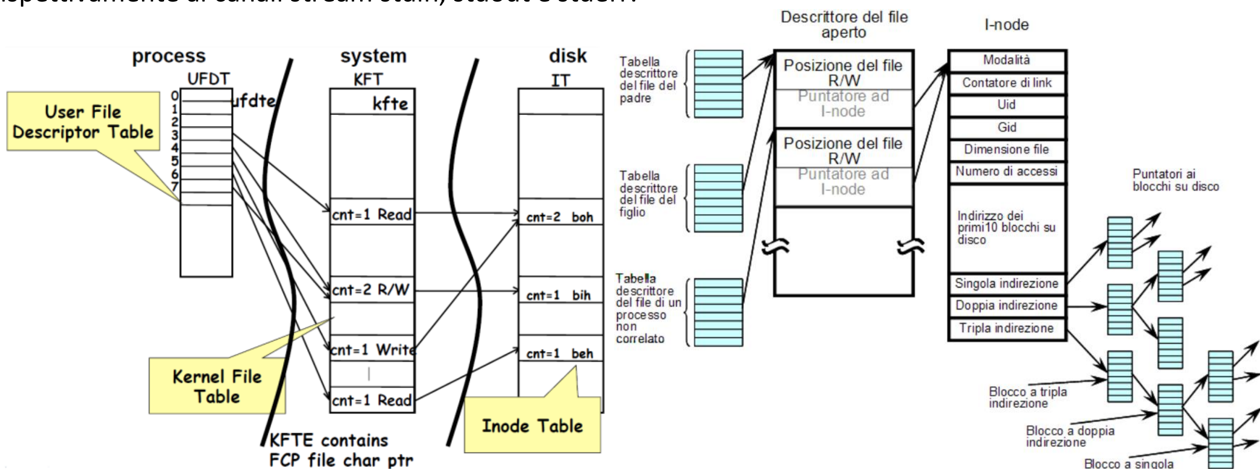
Unix gestisce i blocchi liberi tramite una lista contenuta in una sequenza di blocchi:
Ciascun blocco di questa sequenza contiene un elenco di indici di blocco: il primo è un link al blocco successivo, tutti gli altri individuano blocchi liberi allocabili.
Il primo blocco della sequenza (S1) si trova nel superblocco ed è gestito a stack: un pointer (P) individua l'indice del prossimo blocco allocabile (inizialmente P punta all'ultimo indice di S1 e poi viene decrementato ad ogni allocazione).
Per allocare un blocco, il SO seleziona quello indicato da P e decrementa P: se P raggiunge l'inizio di S1, utilizza il link al blocco successivo per caricare quest'ultimo in S1.
Per liberare un blocco il SO ne inserisce l'indice (push) in S1, (pre-incrementa P): se non c'è posto (S1 è pieno), svuota S1 ricopiandone il contenuto nel nuovo blocco libero, e inserisce in S1, come primo elemento, l'indice di questo blocco.

Accesso ai file

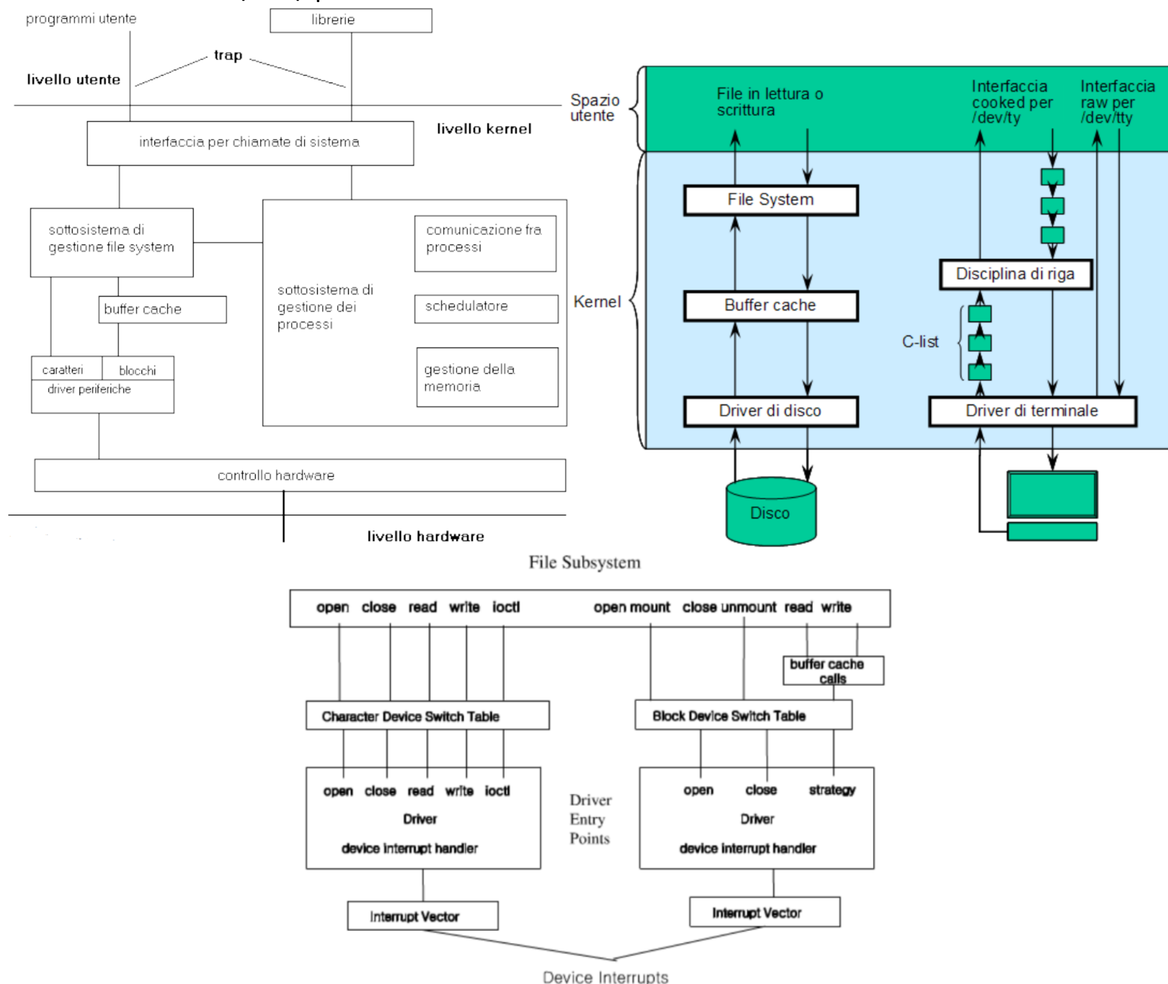
Ogni processo accede ai suoi file aperti attraverso il suo UFDT - User File Descriptor Table; Ogni entry (UFDTE) è un descrittore di file che consente di accedere al file aperto di riferimento (KFTE) nel sistema KFT - Kernel File Table. Un KFTE contiene:

- Un flag di accesso ai file.
- Un contatore di riferimento (numero di ufdte attivo).
- Un puntatore all'inode che identifica un file.
- Un FCP (file char pointer) all'attuale posizione del file.

I primi 3 UFDTE (0, 1 e 2) vengono automaticamente aperti dal sistema per ciascun processo e collegati rispettivamente ai canali stream stdin, stdout e stderr.



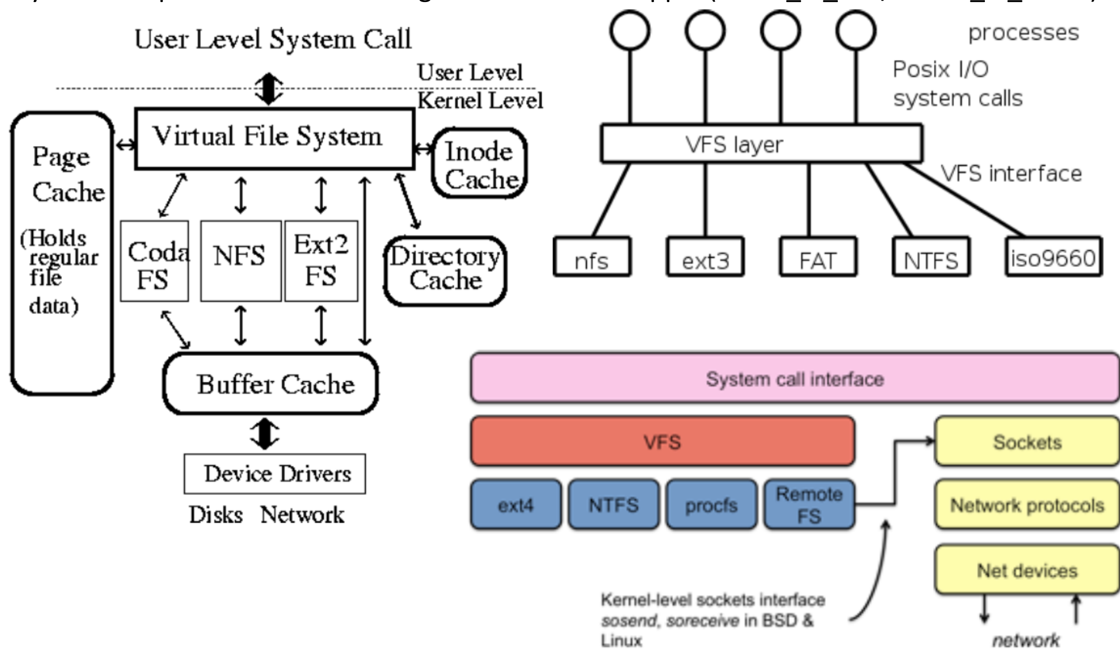
Bufferizzazione
A livello utente: letture/scritture avvengono su buffer nello spazio di indirizzamento utente, chiamate: fopen, fclose, fseek, fread, fwrite, getc, putc, gets, puts, printf.
Buffer caching del disco: i blocchi dei file vengono mantenuti un un buffer in memoria con funzioni di cache (trasparente), problema del crash.
Dispositivi
Ogni device fisico è rappresentato da un file speciale (di tipo device): questi file sono solitamente raggruppati nel directory */dev*, all'interno vi sono le informazioni che identificano il dispositivo: major number codice della classe del dispositivo, minor number codice del dispositivo nella classe.
Ad una classe è associato un device driver
Le usuali operazioni su file (open, close, read, write + ioctl) si 'collegano' al device driver attraverso il file device descrittivo. Es: ls>/dev/lp.



Memory mapped files
I blocchi di file vengono messi su uno spazio di indirizzi virtuale e quindi il loro contenuto può essere letto / scritto come se fosse in elementi di memoria normale, evitando così le chiamate di file system.
In questo modo VM supporta anche la memorizzazione dei file di disco. Allo stesso modo per la condivisione di più processi. UNIX e Linux utilizzano la chiamata mmap () per la mappatura.
Crash recovery
In caso di crash il FileSystem deve essere in grado di rilevare le incongruenze e correggerle il più possibile.
Un altro approccio è un file system logging modifications.
Una transazione è considerata eseguita (commit) dal punto di vista dell'utente quando alcuni metadati relativi sono memorizzati in un file sequenziale di log. Ogni operazione elementare elencata in questo file di registro viene eseguita in background.

Quando tutti i componenti di una transazione vengono eseguiti correttamente, le loro annotazioni vengono rimosse dal buffer.
In caso di crash, per una transazione eseguita è sufficiente eseguire ciò che indica il file di log (non sono ancora azioni eseguite). Se prima del crash la transazione fosse stata interrotta, è necessario eliminare gli effetti dei componenti realmente precedentemente eseguiti, ponendo nuovamente il FileSystem in uno stato coerente.

VFS (Virtual File System) in Linux
Gestisce file che possono essere: file memorizzati su disco, stream di dati da o verso la rete, stream di dati da o verso un processo (proc file), stream di dati da o verso un device driver.
Progettato secondo i principi OO, 3 oggetti principali:i-node, file, file-system.
I directory sono file particolari che contengono una lista di coppie (nome_di_file / indice_di_inode).



Ext2fs: File system in Linux

È lo standard file system su disco di Linux.

Per diminuire la frammentazione interna usa blocchi di piccole dimensioni (1 KB di default, ma anche 2 o 4 KB). Lo spazio su disco è ripartito in gruppi di blocchi contigui. Per non compromettere l'efficienza, cerca di accedere il più possibile a sequenze di blocchi all'interno dello stesso gruppo:

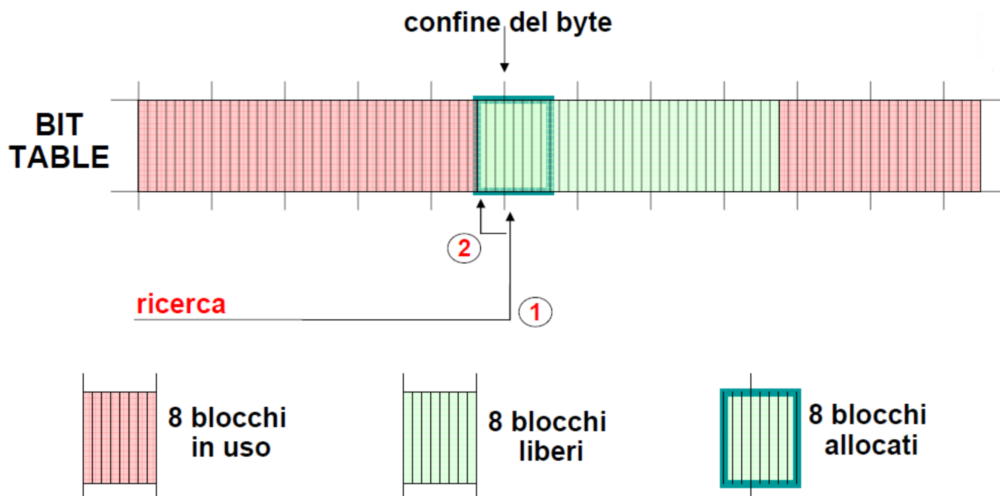
- Usando una politica di allocazione che cerca di collocare i blocchi logici consecutivi di un file in blocchi fisici contigui
- Cercando di tenere i directory, i relativi i-node e i blocchi dei corrispondenti file in posizioni fisicamente vicine (nello stesso gruppo) per diminuire i tempi di seek
- Cercando di collocare directory diversi (e i relativi i-node e file) in gruppi diversi

Allocazione dei blocchi

La gestione dei blocchi liberi è fatta usando una bit table (un bit per blocco), tenendo conto della ripartizione in gruppi. Per allocare il primo blocco di un nuovo file, cerca un blocco libero a partire dall'inizio del gruppo che contiene l'i-node. Per allocare un nuovo blocco ad un file esistente, la ricerca parte dall'ultimo blocco assegnato allo stesso file.

La ricerca (nella bit table) è fatta in due passi:

- Prima viene cercato un byte con tutti 8 i bit nulli (8 blocchi consecutivi liberi), se non lo trova, cerca un bit nullo
- Se trova un byte nullo, esplora all'indietro i bit (del byte precedente) nella bit table, fino a trovarne uno =1 (corrispondente ad un blocco già in uso), a partire dal bit successivo prealloca 8 o più blocchi così evita di lasciare buchi vuoti (frammentazione)



NTFS: File system in Windows NT (10)

- Possibilità di recupero da crash di sistema: le transazioni sui file sono atomiche: vengono memorizzate definitivamente solo se si completano (commit/rollback) e vengono tenute due copie dei dati critici del file system.
- Sicurezza (definizione dei diritti di accesso ai file).
- Gestisce efficientemente dischi di grande capacità e file di grandi dimensioni.
- Per migliorare le prestazioni (diminuendo il numero di accessi al disco) viene usata una disk cache (in memoria) con tecniche di lazy write (scrittura ritardata).
- Possibilità di usare gli attributi dei file come indici.

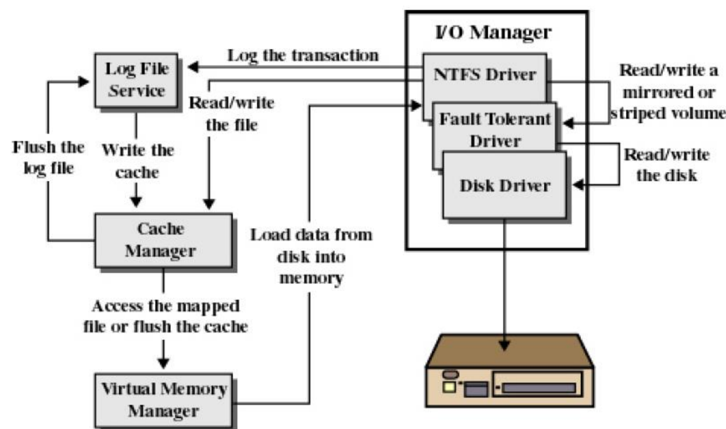
Gestione dello spazio su disco

- Settore: il più piccolo elemento cui accedere (spesso 512 byte)
- Cluster: uno o più settori contigui (1, 2, 4, 8, …, 128 settori)
- **Volume**: partizione logica di un disco: può essere un intero disco fisico, o una parte di esso, o estendersi su più dischi (RAID), dimensione massima 264 byte, struttura di un volume:

boot sectors	master file table	system files	file area

Struttura di un volume

- Settore(i) di boot (da 1 a 16): contiene informazioni sulla partizione e sul file system.
- Master File Table (MFT), situata di seguito ai settori di boot: tabella di record di lunghezza variabile che contengono informazioni relative ad un file o ad un directory (folder), se il file è piccolo il record contiene anche tutto il file.
- File di sistema (contenuti in una regione di circa 1 MB di seguito alla MFT), tra cui:
    - MFT2: copia dei primi 3 record della MFT (in caso di crash)
    - File di log: in cui sono registrati i passi intermedi di ogni transazione (per il commit-rollback)
    - Bit map: bit table che indica i cluster liberi e quelli in uso
    - Tabella di definizione degli attributi: contiene gli attributi supportati nel volume
- Altri file



Recupero da crash

L'obiettivo e di consentire il recupero, dopo un crash del sistema, dei dati riguardanti il file system, non dei dati dei file. L'utente deve quindi aspettarsi di poter recuperare, dopo un crash, il volume e la struttura dei directory di una applicazione, non il contenuto dei file.
Il file di log consente di disfare (rollback) o rifare (commit) le modifiche fatte al file system.
Le modifiche al file system richiedono 4 passi:
● vengono registrate sul log file (nella cache)
● vengono apportate (nella cache)
● viene fatto il flush del log file (le modifiche al log file fatte sulla cache in memoria vengono ricopiate su disco)
● viene fatto il flush delle modifiche al volume

9. UNIX

Creazione dei processi

Ogni processo (figlio) è creato da un altro processo (padre), ciò crea una gerarchia che collega padri e figli.
Ad ogni processo sono associati 7 identificatori:

- Il PID (process ID), assegnato alla creazione, che lo identifica univocamente nel sistema
- Il PID del processo padre
- User ID reale (utente che esegue il processo)
- User ID effettivo (può essere diverso da precedente, vedi flag set-uid)
- Group ID reale (gruppo dell'utente che esegue il processo)
- Group ID effettivo (può essere diverso da precedente, vedi flag set-gid)
- Process group ID (gruppo di processi che condividono lo stesso contesto di standard I/O)

Un descrittore di processo è diviso in 2 parti:

- La prima risiede sempre in memoria centrale (struttura proc) e contiene: stato, priorità, utilizzo tempo di CPU (RCU), segnali pendenti, PID proprio, del padre e PGID, puntatore a struttura user, …
- La seconda che può stare anche in memoria secondaria (struttura user) contiene: altri ID, puntatore a struttura proc, inode directory corrente, directory root, UFDT, …

Ogni processo dispone di un PPRT (Per Process Region Table) con puntatori (logici) alle regioni (segmenti) del processo, di norma ci sono 3 regioni iniziali: text, data, stack (contiene stack utente e heap).
Ogni regione è descritta da un elemento della RT (Region Table) e può avere la propria tabella della pagine.

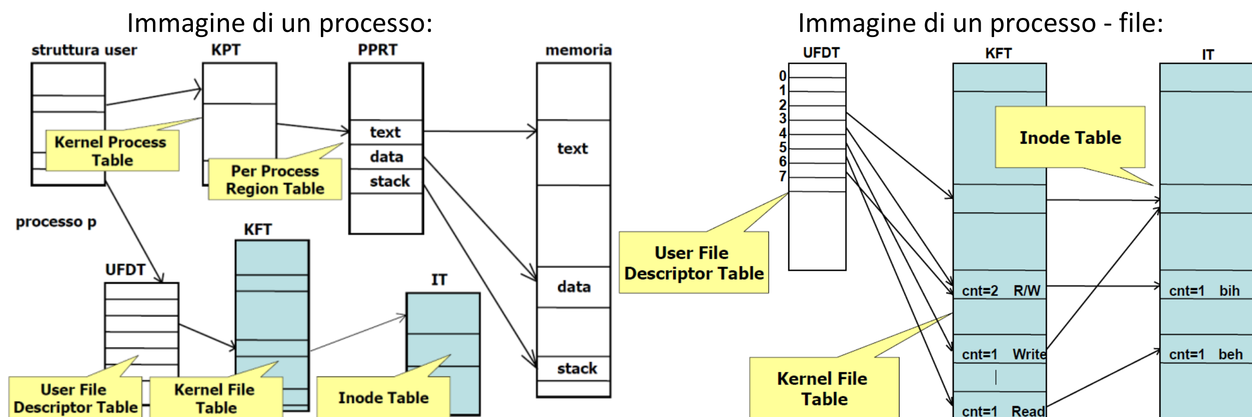
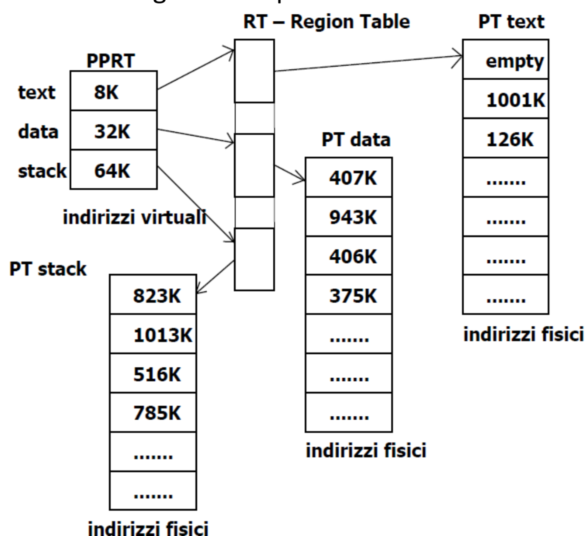


Immagine di un processo - memoria:

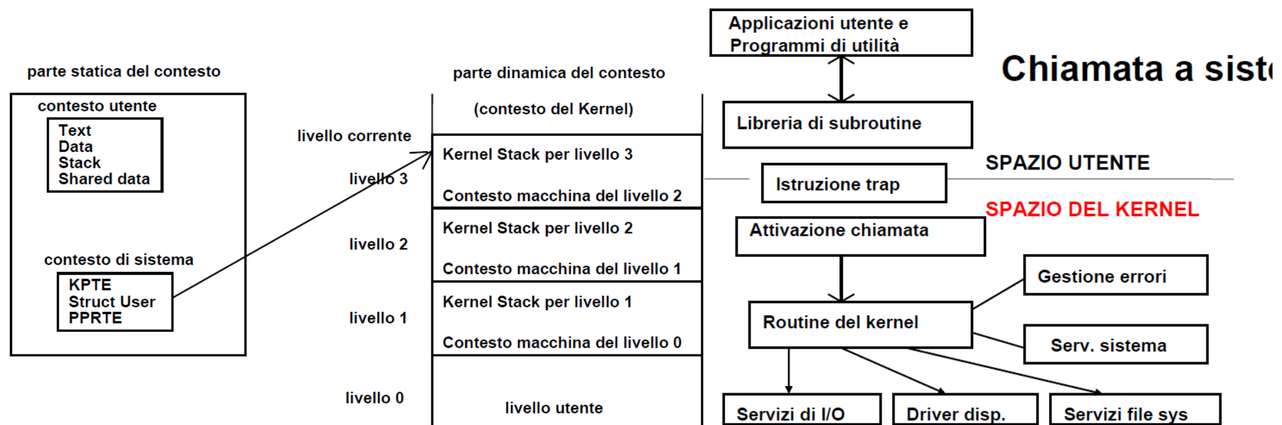


Creazione processo - fork():

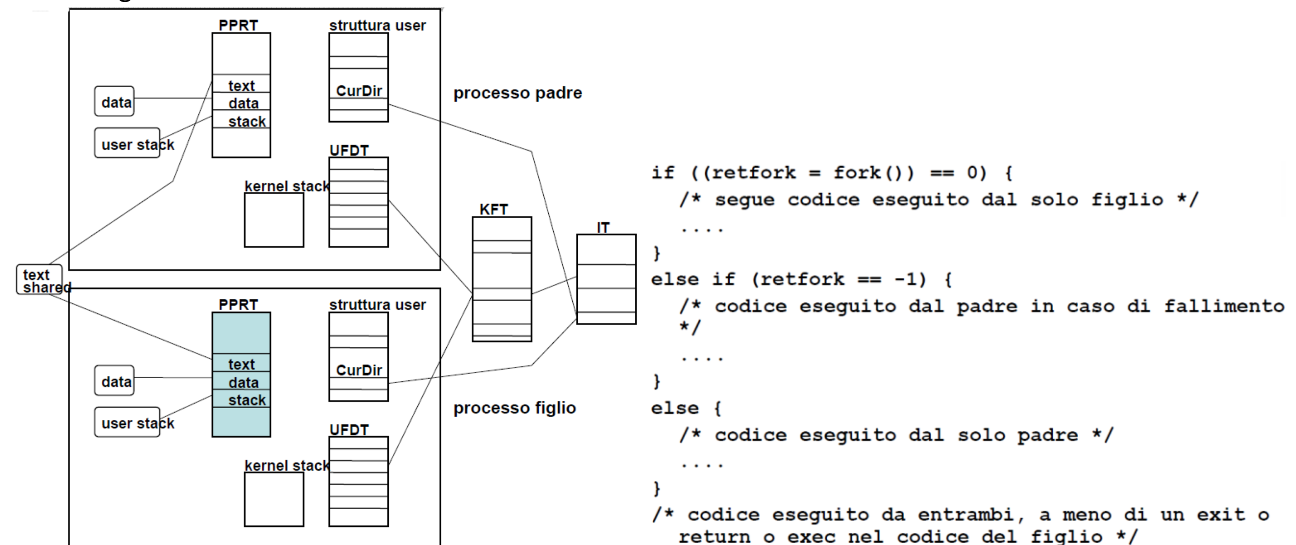
```
pid=fork();        /* if fork succeds, the parent's pid>0 */
if (pid <0) {
                   /* the fork fails: full memory or table */
} else if (pid>0) {
                   /* parent's code*/
} else {
                   /* child's code */
}
```

Contesto di un processo
Il processo può operare in modo utente (esegue il codice utente) o in modo kernel (esegue codice del SO in modalità supervisore).
Gli ingressi al kernel sono 3: Chiamata a sistema, Interrupt, Eccezione.
In modo utente utilizza lo user stack allocato nella regione stack, in modo kernel usa il kernel stack (allocato nella struttura user). Quando, essendo in modo kernel, il SO stabilisce un cambio di processo, viene salvato il contesto del processo corrente e ripreso quello del nuovo processo da eseguire.
Il contesto del processo è dato da:

- Contesto utente: le regioni del processo
- Contesto macchina: registri macchina
- Contesto di sistema: strutture user e proc, kernel stack



La primitiva fork
Crea un nuovo processo (figlio) assegnandovi un nuovo PID e allocando le strutture di sistema.
Adotta un modello a 'clonazione': alloca spazio per le regioni data e stack del processo figlio e vi copia l'attuale contenuto delle regioni del padre, opzionalmente può farlo anche per la regionetext, altrimenti questa risulta condivisa (read-only).
Al padre ritorna come valore di ritorno della funzione fork() il PID del figlio, al figlio il valore 0 (di per sé è un PID speciale di un processo di sistema).
Ritorna –1 in caso di fallimento della clonazione (es: non c'e' spazio di sistema).
Entrambi i processi si trovano ad eseguire l'istruzione (macchina) che segue quella di chiamata alla fork().
Il figlio si vede clonata anche la UFDT e pertanto eredita i file aperti dal padre.
Padre e figlio NON hanno aree dati in memoria condivise.



Primitive collegate:

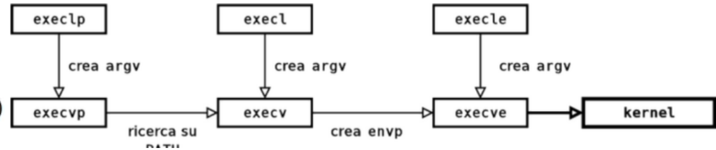
```
int wait(int *retval);
```

Attende la terminazione di un figlio
in *retval è contenuto il motivo della terminazione
e il valore di terminazione fornito dal figlio

```
void exit(int status);
```

Terminazione del processo con relativo valore di terminazione

Famiglia di primitive `exec`
(`execl, execv, execle, execve, execlp, execvp`)
Carica per il processo un nuovo eseguibile

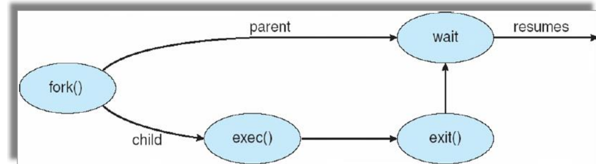


Chiamate di sistema: (s:codice di errore, pid:ID processo, residual: tempo rimanente dal alarm precedente)

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

POSIX Shell

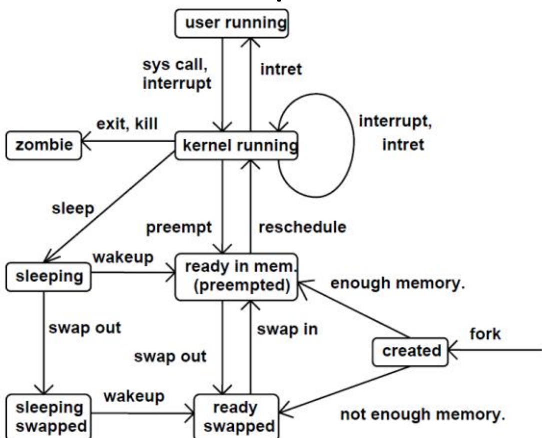
```
while (TRUE) {                              /* repeat forever /*/
    type_prompt( );                         /* display prompt on the screen */
    read_command(command, params);          /* read input line from keyboard */

    pid = fork( );                          /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);           /* error condition */
        continue;                           /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);           /* parent waits for child */
    } else {
        execve(command, params, 0);         /* child does the work */
    }
}
```



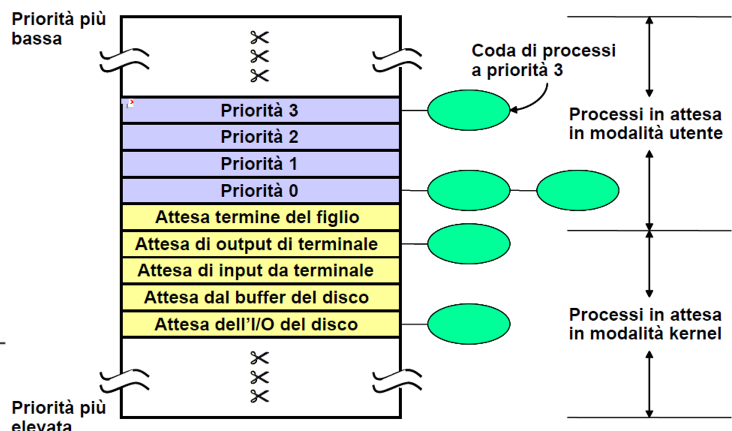
Shell Pipe*: sort < f | head*

La shell crea due processi, sort e head, collegati da un pipe: l'output standard del sort è collegato all'input standard di head. In questo modo tutti i tipi di dati che stanno scrivendo sul suo stdout vanno direttamente a head, leggendo dalla sua stdin, invece di riempire un file. Se il pipe si riempie l'esecuzione di sort si ferma.

Stati di un processo



Code dei processi



Pthread

Il POSIX thread è un API standard (IEEE standard 1003.1c) che comprende decine di funzioni per la sincronizzazione di thread, divise in 5 categorie: thread, mutex, condition variable, barrier, read/write.
Un thread è un flusso di esecuzione all'interno del contesto di un processo. Tutti i thread all'interno dello stesso processo condividono le risorse non duplicate quando viene creato un nuovo thread.
Alla creazione del processo (o all'attivazione di un nuovo eseguibile) il processo ha un thread che esegue l'inizio della funzione main().
Se un thread invoca fork() sistemi differenti utilizzano soluzioni diverse:

- Alcuni duplicano il thread che esegue solo la chiamata.
- Altri permettono entrambe le possibilità: Se la chiamata exec() segue immediatamente la fork(), la duplicazione di tutti i thread che sono stati definiti finora nel processo parent, non è ragionevole perché saranno sostituiti da thread eventualmente creati dal nuovo eseguibile.

I segnali

Sono un meccanismo di sincronizzazione elementare di tipo diretto e asincrono.
Ogni processo ha associata una variabile includente la memoria per n segnali binari (segnali NON cumulabili) e può decidere con una maschera quali abilitare e quali no.
Ogni processo può rispondere ad un segnale abilitato: Ignorando il segnale, Terminando, Attivando una funzione con prefissati parametri come 'routine di servizio' (ASR – Asynchronous Service Routine).
Una ASR va registrata al SO che la attiverà quando il relativo segnale risulta pendente ed abilitato, nel momento in cui il processo passa a running.
Alcuni segnali sono generati dal SO a fronte di eventi hardware o eccezioni (es: Ctrl-C su tastiera invia un segnale SIGTERM a tutti i processi del process group collegato al terminale o alla finestra attiva).
Per inviare un segnale ad un processo: int kill(int pid, int signum);
Per registrare una ASR: void (*signal(int signum, void (*action) (int sig [,int subcode, struct sigcontext *scp])))(int sig);
Quando un processo è bloccato si risveglia se riceve un segnale abilitato (fanno eccezione le code di attesa di massima priorità).
Per realizzare primitive con timeout si accende un timer in modo che generi, alla scadenza, un segnale che potrà risvegliare il processo sospesosi nel frattempo su qualche chiamata sospensiva interrompibile (dal segnale); se arriva in tempo l'evento atteso dal processo (un signal sul semaforo, un messaggio sul mailbox o altro), il processo deve allora spegnere il timer.

10. Real Time OS

Real-time computing

La correttezza di un sistema non dipende solo da una gestione ottimale delle risorse e dalla correttezza logica dell'esecuzione ma anche dal tempo in cui il risultato delle computazioni viene prodotto.

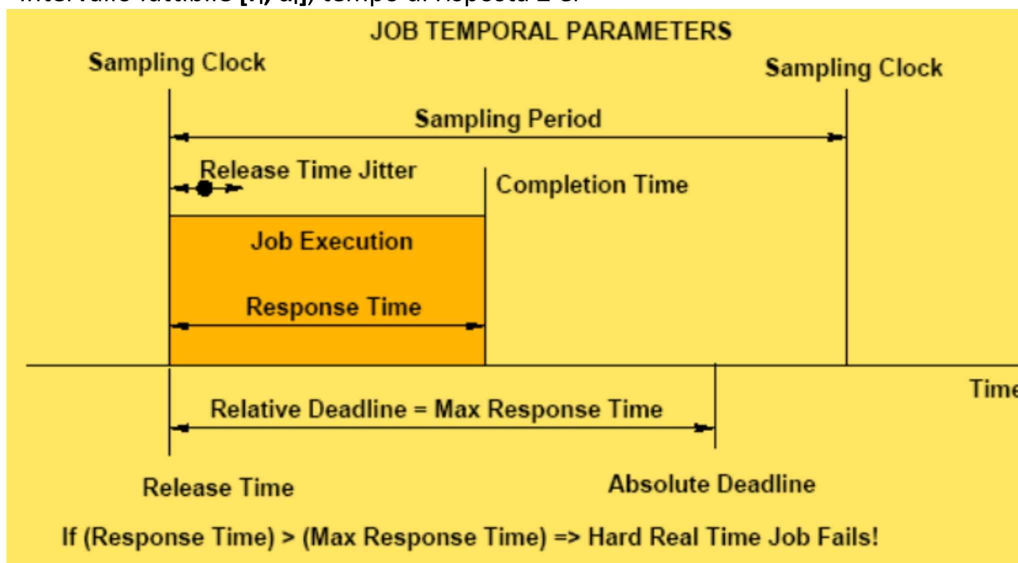
Processi (suddivisi in [sub]task) controllano o reagiscono ad eventi real-time provenienti dall'esterno (che tipicamente producono interruzioni hardware). Tipici sistemi di questa classe sono: di controllo di laboratorio, di impianti, della navigazione, di telecomunicazione, sistemi robotici, controllo/comando militari.

Classificazione dei sistemi Real-time

Ad ogni task real-time viene associato una scadenza (**deadline**) che è l'istante prima del quale è richiesto che l'esecuzione del task abbia inizio (o termini).

- Hard real-time task sono task con deadline vincolante: il non soddisfacimento della scadenza comporta condizioni inaccettabili o un errore fatale.
- Soft real-timetask sono task con deadline non vincolante: il non soddisfacimento della scadenza non pregiudica il funzionamento del sistema, anche se può costituire una situazione degradata.
- Real-time task aperiodico: il deadline riguarda l'avvio o la terminazione del task o entrambi.
- Real-timetask periodico: il vincolo del deadline è posto una volta per ogni periodo. Parametri temporali di un task periodico:
  - $\mathbf{T_i}$ è un task periodico, una sequenza di attività(job).
  - Periodo $\mathbf{p_i}$: intervallo di tempo minimo tra due rilasci di attività successive
  - Tempo di esecuzione $\mathbf{e_i}$: tempo massimo di esecuzione della sua (una delle) attività.
  - Fase $\mathbf{\Phi_i}$: tempo di rilascio della prima attività
  - Deadline relativa $\mathbf{D_i}$: deadline comune delle attività

  Parametri temporali di un'attività(job):
  - Data l'attività $\mathbf{J_i}$: tempo di rilascio **ri**, tempo di esecuzione $\mathbf{e_i}$, deadline assoluta $\mathbf{d_i}$
  - Deadline relativa $\mathbf{D_i}$ ≡ Tempo massimo di risposta consentito (associato col task, $d_i = r_i + D_i$)
  - Intervallo fattibile $\mathbf{[r_i, d_i]}$, tempo di risposta ≥ ei



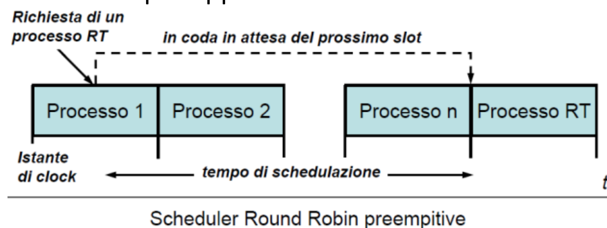
Requisiti specifici dei sistemi Real-time

Questi sistemi non gestiscono solo le risorse di sistema ma cercano di garantire i deadline dei task real-time. I requisiti specifici riguardano le seguenti aree:

- **Determinismo**: Le attività hanno luogo in precisi istanti o predeterminati intervalli di tempo. Il grado di determinismo dipende dalla velocità di risposta alle interruzioni (rapidità di acknowledge) e dalla capacità complessiva di gestire tutte le richieste entro il tempo previsto. Nessun sistema reale è completamente deterministico.
- **Capacità di reazione (responsiveness).**
- Rapidità nel servire un'interruzione: dipende dall'overhead dovuto al salvataggio del contesto e riconoscimento dell'interrupt, dalla durata della ISR, dal nesting di interruzioni di diverso livello.
- Affidabilità: Una degradazione per malfunzionamento anche solo temporaneo, tollerata nei sistemi tradizionali, nei sistemi real-time può essere molto dannosa.
- Recupero dei malfunzionamenti:
    - Il cosiddetto **fail-soft operation** è l'abilità di un sistema che fallisce di recuperare le operazioni in modo tale da preservare la maggior parte delle capacità e dei dati possibili.
    - Un sistema RT è detto **stabile**, nel caso non sia possibile soddisfare tutti i deadline, se è almeno in grado di garantire il soddisfacimento dei deadline dei task più critici e di quelli a maggior priorità.

RT – RR preemptive

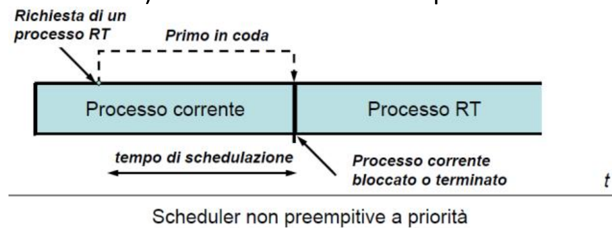
Tempo di scheduling 10 s: in genere non accettabile per applicazioni RT



Scheduler Round Robin preemptive

RT – Nonpreemptive a priorità

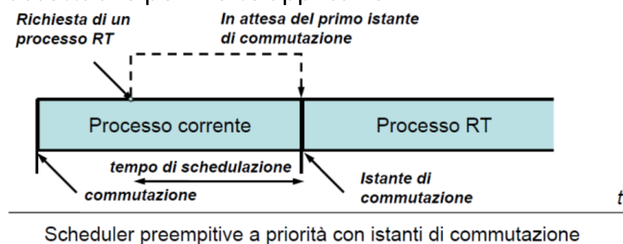
Tempo di scheduling più breve, spesso (processi CPU-bound) non essere accettabile per RT



Scheduler non preemptive a priorità

RT – Preemptive a priorità

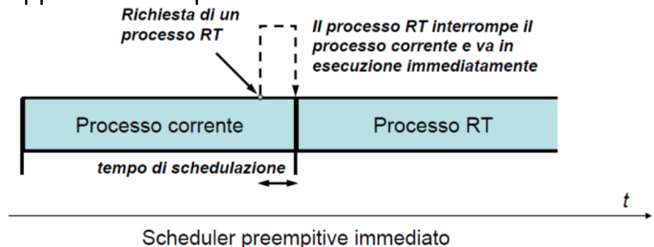
Il preemption può aver luogo allo scadere di quanti temporali. Tempo di scheduling limitato a pochi ms, accettabile per molte applicazioni RT.



Scheduler preemptive a priorità con istanti di commutazione

RT – Preemptive immediato

A fronte di una richiesta più prioritaria, il preemption è immediato. Tempo di scheduling limitato a 100 μs o meno, accettabile per le applicazioni RT più critiche.



Scheduler preemptive immediato

Trattamento dei deadline
È molto difficile gestire direttamente i deadline, si cerca di garantire che un RT task venga rapidamente schedulato all'approssimarsi del deadline. Viene solitamente richiesta una risposta deterministica (nel range frazioni di ms - decine di ms) in un ampio spettro di situazioni.
Conviene utilizzare almeno uno scheduling con punti regolari di preemption (basati su RTC) e assegnare priorità più elevate ai RT task.
Meglio ancora uno scheduling con preemption immediato (salvo regioni 'veramente' critiche ove non consentirlo). Con tempi di risposta alle interruzioni rapidi e preemption immediato si riesce a contenere i ritardi di scheduling sotto i 100 µs.
Scheduling real-time:

- **Statico**, basato su **tabelle preventive**: viene eseguita una analisi statica preliminare che fornisce lo schema che determina quando eseguire i task in modo da garantire i deadline. Applicabile a task periodici di cui si conoscono i parametri significativi (tempo di avvio periodico, tempo di esecuzione, deadline periodico, priorità relativa).
- **Statico**, basato su **priorità precalcolate**: l'analisi è usata SOLO per assegnare le priorità relative (es: l'algoritmo 'rate monotonic').
- **Dinamico**, basato su **planning**: all'arrivo di un nuovo task si determina run-time se accettarlo o meno a seconda che il suo inserimento consenta di soddisfare i suoi vincoli e mantenga soddisfatti gli altri se sì, viene anche stabilito quando schedularlo, in occasione dell'arrivo di un nuovo task, il piano di schedulazione viene ricalcolato in quel momento.
- **Dinamico**, **miglior risultato**: non c'è un'analisi, il sistema abortisce tutti i task attivati il cui deadline è scaduto, metodo implementato da molti sistemi: all'avvio di un task generalmente non periodico gli viene assegnata una priorità (basata sulle caratteristiche del task); viene applicata una qualche forma di scheduling basato su deadline (es: a deadline più vicino), finché non scade il deadline o il task non termina non è dato sapere se il deadline sarà onorato.

Deadline scheduling
La maggior parte degli RTOS moderni si concentra sull'avvio il più rapido possibile dei RT task e sulla rapidità di risposta alle interruzioni.
Una misura ragionevole della qualità di un RTOS è il grado di precisione di attivazione (e terminazione) dei task al tempo giusto, a prescindere da altre questioni in gioco (richieste di risorse, conflitti, ecc.).
Il semplice concetto di priorità è in tal senso insufficiente.
Per uno scheduling RT efficace servono informazioni aggiuntive sui task: ready time, tempo di esecuzione (opzionale), deadline di avvio, deadline di completamento, risorse necessarie, priorità relativa, struttura di sottotask.
Earliest deadline scheduling

Table 10.2 Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
• • •	• • •	• • •	• • •
B(1)	0	25	50
B(2)	50	25	100
• • •	• • •	• • •	• • •

Scheduling ogni 10 ms

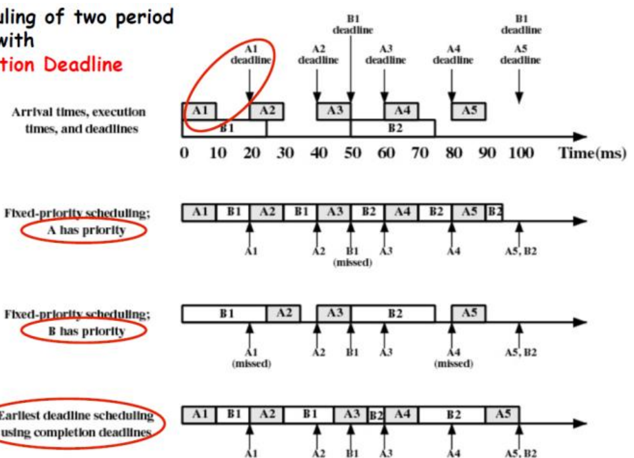
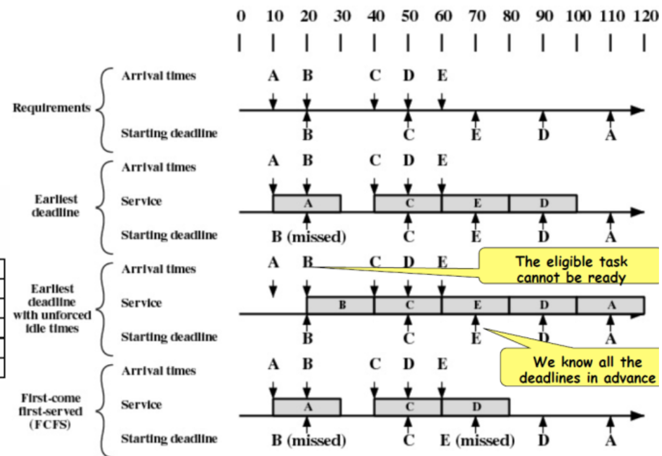


Table 10.3 Execution Profile of Five Aperiodic Tasks

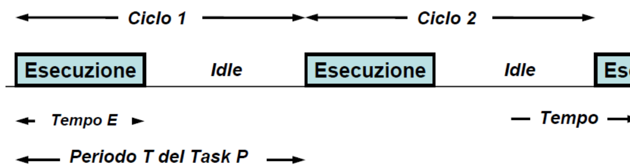
Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70



Scheduling di task aperiodici: starting deadline

Rate Monotonic scheduling

- Alcune applicazioni Real-Time sono costituite da processi periodici dei quali si conoscono le caratteristiche:
- R - Ready Time (istante in cui è pronto ad iniziare),
- S - Starting Deadline (istante entro cui deve iniziare),
- C - Completion Deadline (istante entro cui deve completare),
- E - Tempo di esecuzione,
- T - Periodo di attivazione.



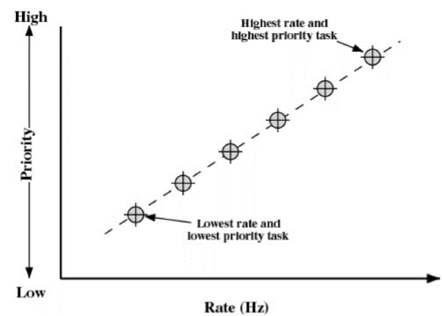
Per task periodici, assegna la priorità sulla base del valore del periodo, le priorità più elevate sono assegnate ai task di periodo inferiore (la priorità cresce con il crescere della frequenza).
Occorre verificare se e con quale margine tutti i deadline sono soddisfatti.
In generale, per **qualsiasi algoritmo** di schedulazione di **n task periodici**, **condizione necessaria** perché i **deadline** siano **soddisfatti** è la seguente:

$\Sigma_i C_i / T_i \leq 1$

C_i = tempo di esecuzione in ciascun periodo del task i

T_i = periodo del task i

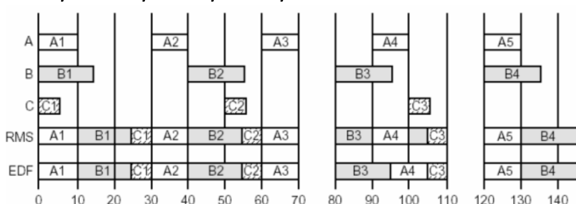
C_i / T_i = percentuale di utilizzazione della CPU



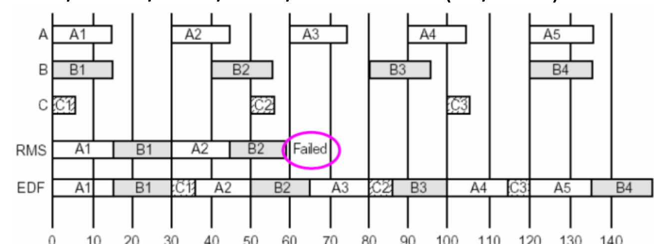
Il valore 1 corrisponde alla piena utilizzazione della CPU nel caso di un algoritmo di schedulazione idealmente perfetto (si trascurano gli overhead di sistema); la relazione limita il numero di task schedulabili
Nel caso **RMS** la **condizione** da soddisfare è più stringente all'aumentare di n: **$\Sigma_i C_i / T_i \leq n(2^{1/n} - 1)$**
Il termine a destra tende a ln 2=0.693 per n→∞

Confronto tra RMS e EDS

TA=30ms TB=40ms TC=50ms → prioA=33 prioB=25 prioC=20
CA=10ms CB=15ms CC=5ms (execution time)
$\Sigma_i C_i / T_i$ = 1/3 + 3/8 + 1/10 = 0.808



TA=30ms TB=40ms TC=50ms → prioA=33 prioB=25 prioC=20
CA=15ms CB=15ms CC=5ms (execution time)
$\Sigma_i C_i / T_i$ = 1/2 + 3/8 + 1/10 = 0.975 3(21/3 – 1) = 0.779

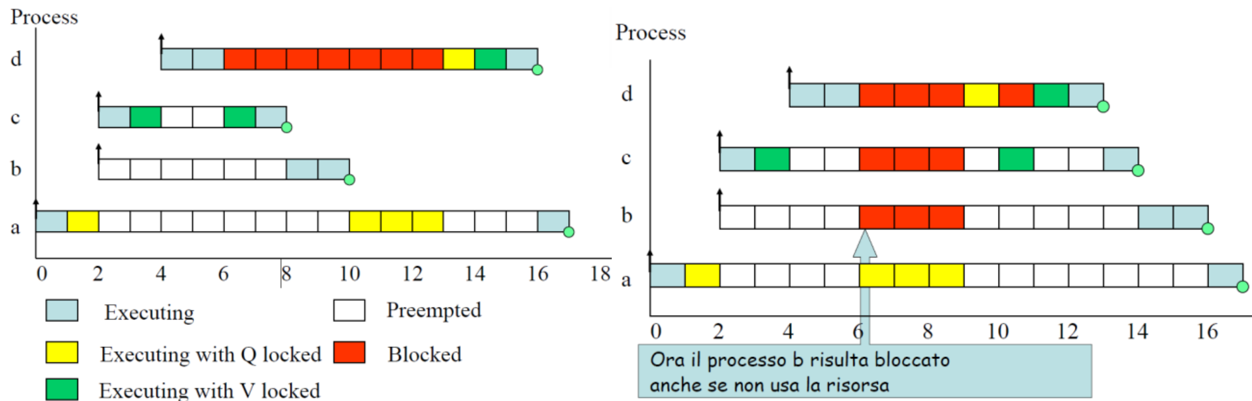


Priority Inversion

In un sistema con scheduling preemptive (preemptive priority scheduling) c'è il rischio di priority inversion:

- Un processo a bassa priorità esegue positivamente una wait(s) su un semaforo di mutua esclusione, successivamente un processo ad alta priorità vi si accoda.
- Altri processi a priorità intermedia per preemption impediscono al processo a bassa priorità di avanzare ed eseguire signal(s) per liberare il semaforo.
- Di conseguenza i processi con priorità inferiore bloccano quello con priorità elevata (inversione della priorità).

Una soluzione: **priority inheritance** (il processo che impegna un semaforo eredita, temporaneamente, la priorità più elevata dei processi in coda sullo stesso semaforo), però l'adozione di questo meccanismo aumenta l'overhead.



Proprietà tipiche dei RTOS:

- Switch molto veloce del contesto (di processo o thread).
- Dimensioni ridotte.
- Risposta rapida agli interrupt esterni.
- Multitasking con IPC (semafori, segnali, eventi).
- File contigui per accessi veloci ai dati memorizzati.
- Riduzione al minimo dei periodi ad interrupt disabilitati.
- Primitive per ritardare i task a tempo.
- Primitive di sospensione/ripresa di un task.
- Allarmi speciali e timeout.

Molti RTOS commerciali implementano tutte/alcune queste caratteristiche.

Note per l’esame

L’esame si divide in 2 parti: una di domande di teoria e una di programmazione, con un intervallo di 5 min tra di esse. Bisogna essere sufficienti in entrambe per superare l’esame, se la parte di teoria è insufficiente non correggere la programmazione.

La parte di teoria è composta da 10 domande di cui se ne devono scegliere 6. Di solito solo 3 domande sono esercizi. Si hanno 60min di tempo.
Gli esercizi sono facili, in un pomeriggio imparate a farli tutti, però alcuni (come quello degli archi) sono lunghi da svolgere e sono soggetti a facili errori di distrazione, quindi consiglio all’esame di partire facendo le domante teoriche e di non perdere troppo tempo su una domanda.
Sotto trovate le domande di teoria uscite in questi anni (si ripetono o sono molto simili).

Per la parte programmazione si hanno 120min. E’ chiesto di risolvere l’esercizio in un determinato modo (Monitor hoare, semafori, java o regioni), gran parte del codice è già scritto e si devono solo completare i metodi, il main ed eventualmente il metodo run() di un altro paio di classi.
Di solito non chiede per 2 appelli di fila di risolvere l’esercizio nello stesso modo, cioè se ad un appello ha chiesto i semafori la volta dopo è poco probabile che lo richieda.

Domande di teoria:

- Descrivere l’evoluzione di un processo tramite diagramma a stati utilizzando un esempio
- a) Definire le condizioni necessarie per lo stallo.
  b) Spiegare, in base a tali condizioni, il motivo per cui l’algoritmo del banchiere costituisce un corretto algoritmo di prevenzione dello stallo
- Allocazione globale delle risorse: definizione, vantaggi e svantaggi di questo metodo.
  Esemplificare i concetti esposti mediante una rete di Petri nel caso di due processi che richiedano contemporaneamente l’uso di 2 risorse non condivisibili.
- Rete di petri 1produttore-1consumatore, e dimostrare perché non va in stallo.

- Descrivere cosa è un semaforo, le sue caratteristiche e le sue estensioni.
- Cose’è un monitor
- Descrivere il costrutto di Monitor nelle due forme presentate (Brinch Hansen e Hoare) e spiegare la differenza tra Monitor e Regione critica.
- a) Spiegare il ruolo del semaforo urgent nella realizzazione a semafori del Monitor di Hoare uaa02.3
- Definire il Monitor di Java. Spiegare il motivo della necessità che siano disponibili sia il metodo notify() che notifyAll(). Evidenziare le differenze tra il Monitor di Java e una regione critica, e tra Monitor di Java e Monitor di Hoare.

- Spiegare in base a quali parametri si possono caratterizzare i diversi algoritmi di scheduling e confrontare RR con PS
- Dare un esempio di priority inversion

- Descrivere il meccanismo della paginazione
- Spiegare in che modo il meccanismo della paginazione consente che due pagine fisiche distinte e compresenti in memoria centrale corrispondano alla stessa pagina logica (stesso indice di pagina nell’indirizzo logico) di due processi utente del sistema, uno correntemente in stato ready e uno waiting. Sotto che aspetto lo strato di gestione della memoria viene interessato, nel caso in esame, quando il processo waiting passa a running?
- Descrivere il meccanismo della segmentazione
- Descrivere il meccanismo della segmentazione con riferimento ad un sistema multitasking che consente la condivisione di segmenti di codice e/o dati da parte di più processi. Il registro limite è

strettamente necessario solo nel caso di segmenti la cui lunghezza effettiva non sia un multiplo di una potenza di 2? Motivare la risposta.
- Spiegare i termini segmentazione interna ed estrena e riguardo la segmentazione.
- Strategia del working set nella gestione della memoria virtuale.

- Ruolo del Device Handler nella gestione dell'I/O

- Spiega Raid2
- Descrivere e dare un esempio di SSTF
- Descrivere e dare un esempio di C-LOOK
- Confrontare SCAN e C-SCAN

- Ruolo i-node in Unix
- Gestione dei blocchi liberi in un file System
- Spiegare tecnica dello spooling
- Effetti della duplicazione durante la chiamata fork()
- Immagine e contesto di un processo UNIX

- Rate Monotic scheduling

Esercizi di teoria:

- Togliere archi, grado parallelismo. (Slide pag 55italiano)
- Grafi holt e sue tabelle, banchiere
- Reti di petri grafo raggiungibilità
- Algoritmi Scheduling
- Algoritmi replacement policy (08.d, slide pag 345italiano)
- Schedulazione degli accessi al disco 08.d
- Esercizio fatto in classe con roba di probabilità