Sistemi operativi semplici (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione

Si definisce *sistema operativo* un insieme di utilità progettate per:

1) Offrire all'utente un'astrazione più semplice e potente della macchina assembler, dunque una macchina virtuale, definita come ambiente virtuale dove eseguire applicazioni e originariamente per sistemi multiutente. Essa deve essere semplice da usare (senza bisogno di conoscenze di microprogrammazione) e potente (es., usando la memoria secondaria per realizzare una più ampia memoria principale virtuale).
2) Gestire in maniera ottimale le risorse fisiche e logiche dell'elaboratore (definendo l'ottimalità come minimizzazione dei tempi di attesa e la massimizzazione dei lavori svolti per unità di tempo).

Un *processo* è un programma in esecuzione e corrisponde a:

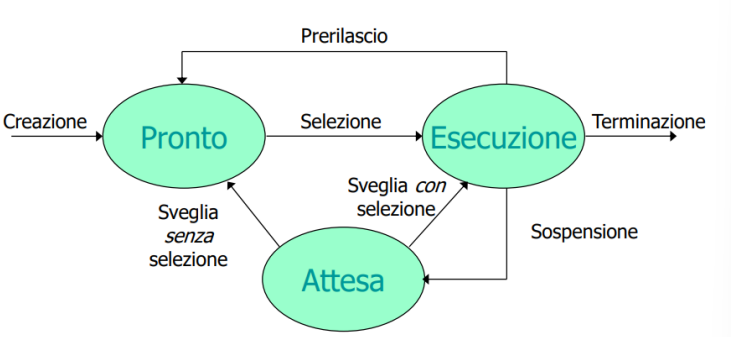
1) L'insieme ordinato di stati assunti dal programma nel corso dell'esecuzione (sulla sua macchina virtuale)
2) L'insieme ordinato delle azioni effettuate dal programma nel corso dell'esecuzione (sulla sua macchina virtuale)

Esso ha un suo spazio di indirizzamento logico, in quanto possiede una parte di memoria della macchina virtuale che può leggere e scrivere, organizzata ad esempio in pagine o segmenti e rappresentano i dati del programma e le sue aree di lavoro.
In un sistema coesistono processi utente e di S/O e possono cooperare tra loro ma hanno privilegi diversi. Avanzano concorrentemente, dato che il S/O assegna loro le risorse necessarie secondo diverse politiche di ordinamento (es. a divisione di tempo/priorità). Spesso hanno bisogno di dover comunicare tra loro e il S/O deve essere in grado di fornire i meccanismi e i servizi necessari.

Un processo può creare processi "figli" (ad esempio, un processo interprete di comandi (shell) lancia un processo figlio per eseguire un comando di utente) ed essi vengono:

- creati per eseguire un lavoro
- sospesi per consentire l'esecuzione di altri processi
- terminati al compimento del lavoro assegnato (un processo figlio che sopravvive alla terminazione del processo padre è detto "orfano" e può essere molto dannoso)

Riporta questi stati di avanzamento:



Il *gestore dei processi* costituisce il cuore o nucleo del S/O (kernel) e gestisce ed assicura l'avanzamento dei processi (ognuno col suo stato di avanzamento come si vede sopra) e la scelta del processo da eseguire ad un dato istante si chiama ordinamento (scheduling).
Il gestore decide il cambio di stato dei processi, in particolare:

- per divisione di tempo
- per trattamento di eventi (es. risorsa libera/occupata

Esso inoltre ha il compito di gestire il nucleo del S/O e di:

- gestire l'avanzamento dei processi, registrando ogni transizione nel loro stato di attivazione
- gestire le interruzioni esterne (all'esecuzione corrente) causate da eventi di I/O o situazioni anomale rilevate da altri processi o componenti del S/O
- consentire ai processi di accedere a risorse di sistema e di attendere eventi

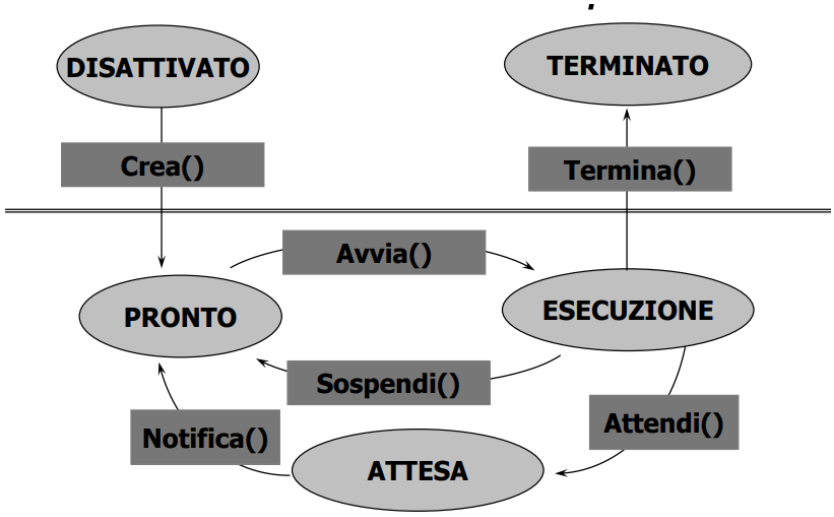
La politica di ordinamento deve essere equa (fair → fairness), perché i processi pronti per eseguire devono avere l'opportunità di farlo e processi in attesa di risorse devono avere l'opportunità di accederle.
I meccanismi e servizi di comunicazione e sincronizzazione devono essere efficaci, in quanto il dato (o segnale) inviato da un processo mittente deve raggiungere il destinatario in un tempo breve e in modo sicuro.

Viene definita *risorsa* un qualsiasi elemento fisico (hardware) o logico (realizzato a software) necessario alla creazione, esecuzione e avanzamento di processi.
Le risorse possono essere
– Durevoli (es., CPU)
– Consumabili (es., memoria fisica)
– Ad accesso divisibile o indivisibile
  • Divisibile se tollera alternanza con accessi di altri processi
  • Indivisibili se non tollera alternanza durante l'uso
– Ad accesso individuale o molteplice
  • Molteplicità fisica o logica (virtualizzata)

La *risorsa per la CPU* è risorsa indispensabile per l'avanzamento di tutti i processi.
- A livello fisico (hardware) corrisponde alla CPU
- A livello logico (sotto gestione software) può essere vista come una macchina virtuale (offerta dal S/O alle sue applicazioni)

La *risorsa per la memoria*, invece, in scrittura è una risorsa ad accesso individuale e in lettura è risorsa ad accesso multiplo. La gestione software la virtualizza (usandola insieme alla memoria secondaria) attribuendone l'accesso ai vari processi secondo particolari politiche. Se virtualizzata, diventa riutilizzabile e prerilasciabile e altrimenti consumabile e indivisibile. Si ha una gestione velocizzata con l'utilizzo di supporto hardware.
Le risorse di I/O sono risorse generalmente riutilizzabili, non prerilasciabili, ad accesso individuale.
La gestione software ne facilita l'impiego nascondendone le caratteristiche hardware e uniformandone il trattamento. L'accesso fisico ha bisogno di utilizzare programmi proprietari e specifici (es. BIOS).

Per quanto riguarda il S/O può risiedere
• Permanentemente in ROM
  – Soluzione tipica di sistemi di controllo industriale e di sistemi dedicati
• In memoria secondaria per essere caricato (tutto o in parte) in RAM all'attivazione di sistema (bootstrap)
  – Adatto a sistemi di elevata complessità oppure predisposti al controllo (alternativo) da parte di più istanze di S/O
  – In ROM risiede solo il caricatore di sistema (bootstrap loader)
Di seguito, gli stati di avanzamento di un processo:



Stati:

- DISATTIVATO
  - Il programma è in memoria secondaria. Un supervisore lo carica in memoria mediante una chiamata di sistema che crea una struttura di controllo di processo (Process Control Block, PCB)
- PRONTO
  - Il processo, pronto per l'esecuzione, rimane in attesa del suo turno
- ESECUZIONE
  - Il processore è stato attribuito al processo selezionato, la cui esecuzione avanza
- ATTESA
  - Il processo è sospeso in attesa di una risorsa attualmente non disponibile o di un evento non ancora verificatosi
- TERMINATO
  - Il processo ha concluso regolarmente le sue operazioni e si predispone ad abbandonare la sua macchina virtuale

Transizioni:

- Crea()
  - Assegna una macchina virtuale a un nuovo processo, aggiornando la lista dei processi pronti (ready list)
- Avvia()
  - Manda in esecuzione il primo processo della lista dei pronti
- Sospendi()
  - Il processo in esecuzione ha esaurito il suo quanto di tempo e torna in fondo alla lista dei pronti
- Attendi()
  - Il processo richiede l'uso di una risorsa o l'arrivo di un evento e viene sospeso se la risorsa è occupata o se l'evento non si è ancora verificato
- Notifica()
  - La risorsa richiesta dal processo bloccato è di nuovo libera o l'evento atteso si è verificato. Il processo ritorna nella lista dei pronti
- Termina()
  - Il processo in esecuzione termina il suo lavoro e rilascia la macchina virtuale

Modello di processo realizzato tramite struttura a tabella (Process Table), visto come array di strutture per la gestione dei processi. Ogni processo è rappresentato da un descrittore (Process Control Block) contenente:
– Identificatore del processo
– Contesto di esecuzione (stato interno) del processo
– Stato di avanzamento del processo
– Priorità (iniziale ed attuale)
– Diritti di accesso alle risorse e privilegi
– Puntatore al PCB del processo padre e degli eventuali processi figli
– Puntatore alla lista delle risorse assegnate alla macchina virtuale del processo

La tabella dei processi è un array di PCB, cioè contiene logicamente un PCB per tutti i processi correnti nel sistema. Listiamo completamente tutti i campi della tabella dei processi:

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Diversi metodi utili per determinare quando porre un processo in stato di esecuzione in sostituzione di un altro (switch)
– Scambio cooperativo (cooperative o non pre-emptive switching)
• Il processo in esecuzione decide quando passare il controllo al processo successivo
– Windows 3.1
– Scambio a prerilascio: il processo in esecuzione viene rimpiazzata da
• Un processo pronto a priorità maggiore (priority-based pre-emptive switching) → Sistemi detti "a tempo reale"
• All'esaurimento del suo quanto di tempo (time-sharing pre-emptive switching) → Unix, Windows NT

Il componente che avvia processi all'esecuzione (ma non il selettore, scheduler!) viene detto *dispatcher*. Deve essere molto efficiente perché gestisce ogni scambio e deve salvare il contesto del processo in uscita, installare quello del processo in entrata (context switch) e affidargli il controllo della CPU.
L'efficienza del dispatcher si misura in
– Percentuale di utilizzo della CPU
– Numero di processi avviati all'esecuzione per unità di tempo
– Durata di permanenza di un processo in stato di pronto

I processi in stato di pronto sono accodati in una struttura detta lista dei pronti (ready list). La più semplice gestione della lista è con tecnica a coda (First-Come-First-Served, FCFS)

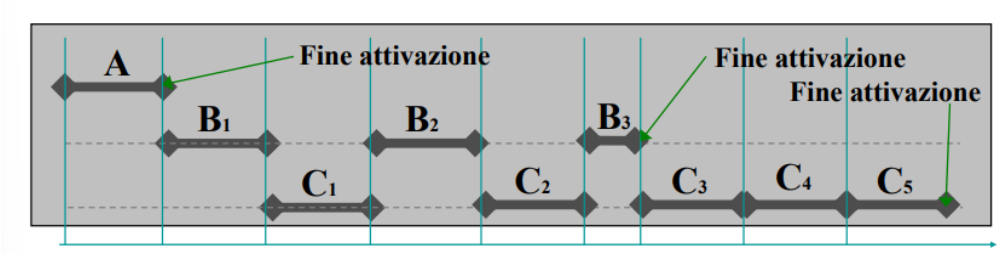
- Il primo processo ad entrare in coda sarà anche il primo avviato all'esecuzione
- Facile da realizzare e da gestire
- La garanzia di esecuzione di altri processi (fairness) dipende dalla politica di scambio
- Lo scambio cooperativo non offre garanzie

Le attività di un processo tipicamente comprendono sequenze di azioni eseguibili dalla CPU intervallate da sequenze di azioni di I/O. I processi si possono dunque classificare in
– CPU-bound
  • Comprendenti attività sulla CPU e di durata molto lunga
– I/O- bound
  • Comprendenti attività di breve durata sulla CPU, intervallate da attività di I/O molto lunghe
La tecnica FCFS penalizza i processi della classe I/O- bound.

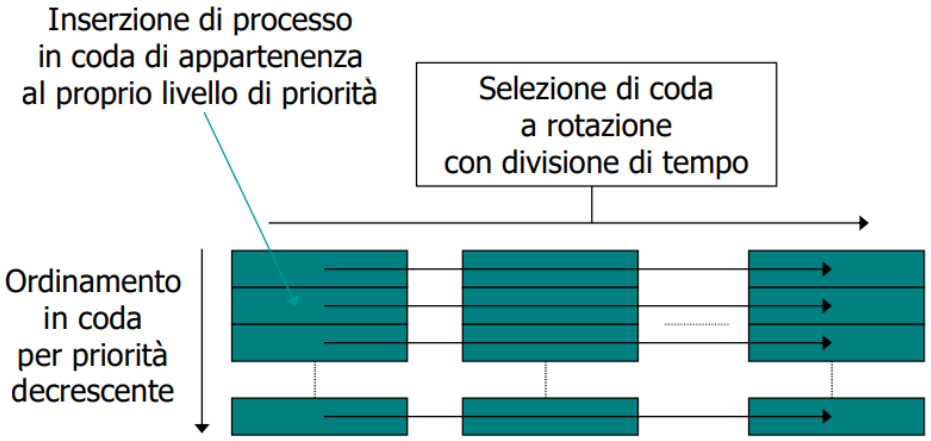
Il termine CPU-bound descrive uno scenario in cui l'esecuzione di un'attività o di un programma dipende fortemente dalla CPU. Prima di andare avanti, definiamo cos'è una CPU. La CPU si riferisce solitamente all'unità di elaborazione centrale di un dispositivo informatico, come un computer desktop. È responsabile del controllo dell'esecuzione di attività e programmi in un sistema informatico. In quanto tale, un dispositivo informatico non può funzionare senza di essa.

In un ambiente CPU-bound, il più delle volte il processore è l'unico componente utilizzato per l'esecuzione. Ciò significa che gli altri componenti del sistema informatico sono raramente utilizzati durante l'esecuzione. A causa di questa dipendenza, tutto ciò che riguarda l'esecuzione dei programmi dipende dalla CPU. Di conseguenza, se vogliamo che un programma venga eseguito più velocemente, dobbiamo aumentare la velocità della CPU.
Inoltre, le operazioni legate alla CPU tendono ad avere pochi e lunghi burst della CPU. Il burst della CPU si riferisce alla quantità di tempo impiegata per eseguire un'attività, di solito con la CPU. Di conseguenza, è sempre consigliabile assegnare una priorità più bassa a questi task per evitare di sprecare risorse.

Imponendo la suddivisione di tempo (time-sharing) sulla politica FCFS si deriva una tecnica di rotazione detta *round-robin.* Ogni task pronto viene eseguito turno per turno solo in una coda ciclica per una fetta di tempo limitata. Vediamone l'applicazione su tre processi A, B e C con tempi di esecuzione 2, 5 e 10 ms e quanto di tempo 2 ms:

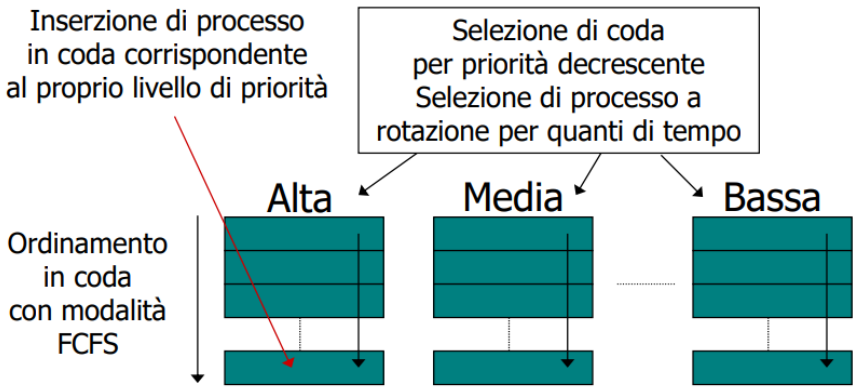


Vediamo un'idea di politica a rotazione con priorità, in cui si ha l'inserzione di processo in coda di appartenenza al proprio livello di priorità e un ordinamento in coda per priorità decrescente, con una selezione basata prima di tutto sulla priorità stessa e con una giusta divisione di tempo.



A ogni singolo processo possiamo attribuire una priorità individuale che denota il suo livello di privilegio nel sistema. Processi diversi possono poi essere categorizzati per attributi (p.es., CPU- bound, I/O- bound). Possiamo allora istituire una coda per ciascuna categoria di processo e ordinarla a priorità. Stabiliamo poi una politica di ordinamento tra code (p.es.: round-robin) ed otteniamo una politica di ordinamento a livelli, con rotazione tra code e con priorità entro ciascuna coda.

Possiamo anche facilmente (e utilmente) definire una politica duale alla precedente.
Istituiamo una coda per ogni livello di priorità attribuita ai processi e selezioniamo la coda a priorità più elevata. Applichiamo la politica a rotazione (round-robin) sul processo selezionato. Così otteniamo la politica a priorità con rotazione, con apposita selezione prioritaria tra code e a rotazione equa entro ciascuna coda.



Sincronizzazione

Processi indipendenti possono avanzare concorrentemente senza alcun vincolo di ordinamento reciproco • In realtà molti processi condividono risorse e informazioni funzionali. Per gestire la loro condivisione servono meccanismi di sincronizzazione di accesso.

Siano A e B due processi che condividono la variabile **X** inizializzata al valore **10**

- Il processo A deve incrementare **X** di **2** unità
- Il processo B deve decrementare **X** di **4** unità

A e B leggono **concorrentemente** il valore di **X**

- Ovvero A legge che **X** è **10** ma viene prerilasciato prima di poter eseguire la sua operazione, così anche B legge che **X** è **10**
- A questo punto entrambi eseguiranno le rispettive operazioni sul valore originale di **X**, in concorrenza, ma senza tener conto dell'operazione svolta dall'altro processo
- Il processo A scrive in X_A il proprio risultato (**12**)
- Il processo B scrive in X_B il proprio risultato (**6**)
- Ovviamente può accadere anche il viceversa (prima legge B e poi A, ecc.)

Il valore finale in **X** è l'ultimo tra X_A e X_B a essere scritto!

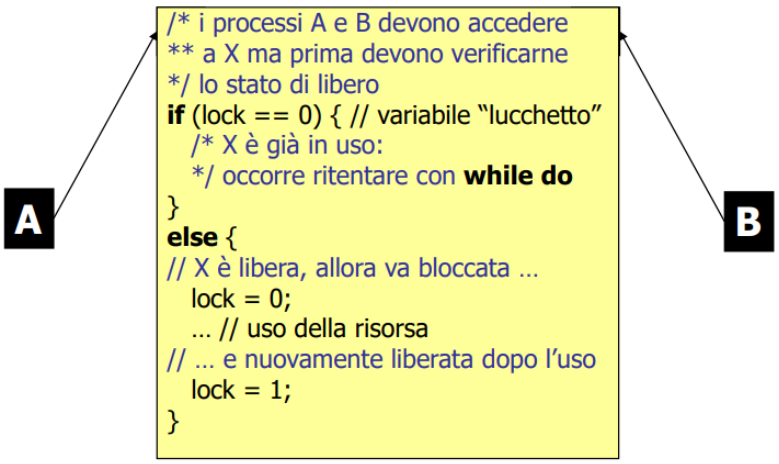
- **12** oppure **6**

Il valore atteso in **X** invece era **8**

- Ottenibile **solo** con sequenze (A;B) o (B;A) **indivise** di lettura e scrittura
- Ovvero, una volta che uno dei due processi ha iniziato a maneggiare (anche solo leggere) la variabile **X**, all'altro processo non deve essere consentito l'accesso alla variabile (non deve essere consentita l'esecuzione di istruzioni che maneggiano la variabile **X**)

Questo problema è detto race condition: il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguite le loro istruzioni. La sezione critica (o regione critica) è la porzione di codice che delimita l'accesso a una risorsa condivisa tra più flussi di esecuzione di un sistema concorrente ed è dove si determina la race condition.

La modalità di accesso indivisa a una variabile condivisa viene detta "in mutua esclusione". L'accesso consentito a un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa. Si utilizza una variabile logica "lucchetto" (lock) che indica se la variabile condivisa è al momento in uso a un altro processo che è anche detta "struttura mutex" (mutual exclusion).
Come viene implementata? Mostriamo prima una soluzione che non funziona.



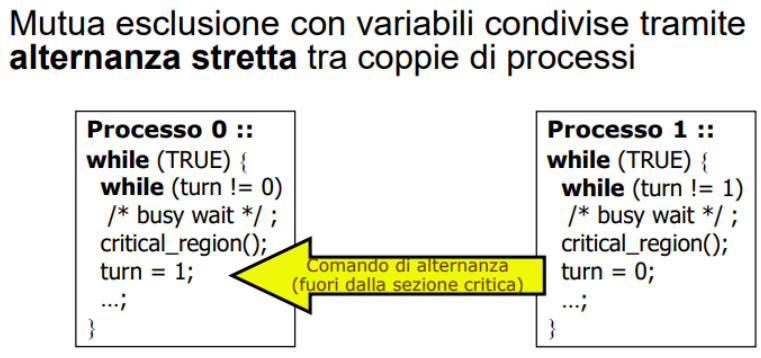
Ciascuno dei due processi può essere prerilasciato dopo aver letto la variabile lock ma prima di esser riuscito a modificarla.
Questa situazione è ancora una race condition e può generare pesanti inconsistenze. Inoltre, l'algoritmo mostrato richiede attesa attiva (busy wait) che causa spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto.

Quindi, la busy wait è l'attesa con verifica continua che una variabile cambi valore. Il processore rimane occupato con questa inutile verifica continua, invece che progredire con operazioni che possano far cambiare valore a lock.

Una soluzione al problema della sincronizzazione di processi è ammissibile se soddisfa le seguenti quattro condizioni:

1. Garantire accesso esclusivo
2. Garantire attesa finita
3. Non fare assunzioni sull'ambiente di esecuzione
4. Non subire condizionamenti dai processi esterni alla sezione critica

Possibili soluzioni:
1)



Tre gravi difetti → soluzione non ammissibile!
• Uso di attesa attiva (busy wait)
• Condizionamento esterno alla sezione critica
• Rischio di race condition sulla variabile di controllo

2)
Proposta da G. L. Peterson, migliore della precedente ma ingenua rispetto al funzionamento dei processori (multicore) di oggi e si applica solo a coppie di processi

```
IN(int i) :: {
int j = (1 - i); // l'altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == i)
 { // attesa attiva  };
OUT(int i) :: {
flag[i] = FALSE; }
```

```
Processo (int i) ::

while (TRUE) {
  IN(i);
  // sezione critica
  OUT(i);
  // altre attività
}
```

Immaginiamo ci siano due processi concorrenti che eseguono esattamente lo stesso codice, alternandosi sul processore

– Ognuno dei due processi vedrà sé stesso come il processo i e l'altro come il processo j

– Potendo essere i uguale solo a 0 o 1 ed essendo la controparte $j = 1 - i$, allora j è 0 quando i è 1 e viceversa

Ognuno dei due processi lascerà passare avanti l'altro se vede $flag[j]$ come 1 e turn come sé stesso (ovvero come i, che per un processo corrisponde a 0 e per l'altro corrisponde a 1)

– Significato semplificato: «anche l'altro processo ha richiesto l'accesso e io sono arrivato per secondo visto che turn ha il mio identificativo (ho quindi sovrascritto l'identificativo dell'altro)»

La condizione di uscita in *IN()* impone che il processo i resti in attesa attiva fin quando il processo j non abbia invocato *OUT()*

– Per *cui* $flag[j]$ = *FALSE*

– Si ha attesa attiva quando non vi sia coerenza tra la richiesta di turn e lo stato dell'altro processo

Su $flag[]$ non vi può essere scrittura simultanea che si ha invece su *turn*

–La condizione di uscita è però espressa in modo da evitare il rischio di race condition e non viene decisa solo dal valore assunto da *turn*

Esempio:

– Supponiamo vada in esecuzione prima il processo *0 (P0)*

• Esso vedrà i come 0 e j come 1

• $flag[0]$ sarà posta a 1

– Supponiamo che ora *P0* sia prerilasciato prima di modificare la variabile *turn* e vada in esecuzione il *processo 1 (P1)*

• Esso vedrà i come 1 e j come 0

• $flag[1]$ sarà posta a 1

• *turn* sarà posta a 1 (ovvero sé stesso)

```
IN(int i) :: {
int j = (1 - i); // l'altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == i)
 { // attesa attiva  };
OUT(int i) :: {
flag[i] = FALSE; }
```

```
Processo (int i) ::

while (TRUE) {
  IN(i);
  // sezione critica
  OUT(i);
  // altre attività
}
```

Supponiamo che ora anche P1 sia prerilasciato e torni in esecuzione P0 (continuando l'esecuzione precedente

• *turn* sarà posta a 0 (ovvero sé stesso)

• La condizione while è verificata e P0 si trova in attesa attiva (ripetendo while senza sosta fino al nuovo prerilascio)

A un certo punto P0 sarà prerilasciato a favore di P1

• P1 non soddisfa la condizione while e quindi prosegue nella sezione critica gestita in mutua esclusione

• Se P1 dovesse essere prerilasciato, P0 si troverebbe ancora in attesa attiva fino a rilasciare il processore; potrà entrare in sezione critica solo quando P1 concluderà la sua sezione critica, chiamando *OUT* e ponendo quindi $flag[1]$ = *FALSE*

Tecniche complementari e/o alternative:
– Disabilitazione delle interruzioni
- • Previene il prerilascio dovuto all'esaurimento del quanto di tempo e/o la promozione di processi a più elevata priorità
- • Può essere inaccettabile per sistemi soggetti a interruzioni frequenti
- • Sconsigliabile lasciare controllo interrupt a utenti (e se non li riattivano?) e inutile con due processori

– Supporto hardware diretto: *Test-and-Set-Lock*
- • Cambiare atomicamente valore alla variabile di lock se questa segnala "libero"
- • Evita situazioni di race condition ma comporta sempre attesa attiva

```
enter_region

!! regione critica: la zona di programma che delimita
!! l'accesso e l'uso di una variabile condivisa

leave_region
```

```
enter_region:
                    !! Copia il valore di LOCK in R1
 TSL R1, LOCK       !! e pone LOCK = 1 (bloccato)
                    !! inoltre, blocca memory bus

 CMP R1, 0          !! verifica se LOCK era 0 tramite R1
 JNE enter_region   !! attesa attiva se R1==1 (tornando a
                    !! enter_region)
 RET                !! altrimenti ritorna al chiamante
                    !! con possesso della regione critica

leave_region:
 MOV LOCK, 0        !! scrive 0 in LOCK (accesso libero)
 RET                !! ritorno al chiamante
```

52

Sia la soluzione di Peterson che quella TSL sono corrette ma hanno il difetto di eseguire busy wait .
Ulteriore possibile problema: *Inversion priority*.

- Consideriamo due processi: • H (ad alta priorità) e L (a bassa priorità)
- Supponiamo che H prerilasci il processore per eseguire I/O
- Supponiamo che H concluda le operazioni di I/O mentre L si trova nella sua sezione critica
- H rimarrà bloccato in busy waiting perché L non avrà più modo di concludere la sezione critica

Il problema del produttore-consumatore (conosciuto anche con il nome di problema del buffer limitato o bounded-buffer problem, presentato sotto) è un esempio classico di sincronizzazione tra processi. Il problema descrive due processi, uno produttore (in inglese producer) ed uno consumatore (consumer), che condividono un buffer comune, di dimensione fissata. Compito del produttore è generare dati e depositarli nel buffer in continuo. Contemporaneamente, il consumatore utilizzerà i dati prodotti, rimuovendoli di volta in volta dal buffer. Il problema è assicurare che il produttore non elabori nuovi dati se il buffer è pieno, e che il consumatore non cerchi dati se il buffer è vuoto.

La soluzione per il produttore è sospendere la propria esecuzione se il buffer è pieno; non appena il consumatore avrà prelevato un elemento dal buffer, esso "sveglierà" il produttore, che ricomincerà quindi a riempire il buffer. Allo stesso modo, il consumatore si sospenderà se il buffer è vuoto; non appena il produttore avrà scaricato dati nel buffer, risveglierà il consumatore. Questa soluzione può essere implementata tramite delle strategie di comunicazione tra processi, tipicamente con dei semafori. Una soluzione errata potrebbe dar luogo ad un *deadlock* (processi bloccati sempre), in cui entrambi i processi aspettano di essere risvegliati. Il problema può anche essere riscritto considerando più produttori e più consumatori distinti.

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

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

Soluzione sleep & wakeup; queste ultime non vengono memorizzate. Se non c'è una sleep in attesa, le wakeup vengono perse

Rischio race condition su variabile count
– Il buffer è vuoto e il consumer ha appena letto che count = 0
– Prima che il consumer istanzi la sleep, lo scheduler decide di fermare il consumer e di eseguire il producer
– Il producer produce un elemento e imposta count = 1
– Siccome count = 1 il producer emette wakeup (che nessuno ascolta)
– A un certo punto lo scheduler deciderà di eseguire di nuovo il consumer, il quale istanzia finalmente la sleep, ma siccome count è già stato riportato a 1 e la corrispondente wakeup è già stata emessa, il consumer non verrà più risvegliato

Soluzione mediante semaforo, dovuta ad E. W. Dijkstra (1965).
Richiede accesso indiviso (atomico) alla variabile di controllo detta semaforo
• Per questo la struttura semaforo si appoggia sopra una macchina virtuale meno potente che fornisce una modalità di accesso indiviso più primitiva
• Semaforo binario (contatore Booleano che vale 0 o 1)
• Semaforo contatore (consente tanti accessi simultanei quanto il valore iniziale del contatore)
La richiesta di accesso P, oppure down, decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante. L'avviso di rilascio V, oppure up incrementa di 1 il contatore e chiede al dispatcher di porre in stato di "pronto" il primo processo in coda sul semaforo.

L'uso di una risorsa condivisa R è racchiuso entro le chiamate di P e V sul semaforo associato a R.

```
Processo Pi ::
{ // avanzamento
  P(sem);
  /* uso di risorsa
     condivisa R */
  V(sem);
  // avanzamento
}
```

P(sem) viene invocata per richiedere accesso a una risorsa condivisa R

- Quale R tra tutte?

V(sem) viene invocata per rilasciare la risorsa

Il semaforo binario (mutex) è una struttura composta da un campo valore intero e da un campo coda che accoda tutti i PCB dei processi in attesa sul semaforo.
Nel semaforo binario, il valore del semaforo è compreso tra 0 e 1. È simile al blocco mutex, ma il mutex è un meccanismo di blocco, mentre il semaforo è un meccanismo di segnalazione.

```
void P(struct sem){
  if (sem.valore == 1)
     sem.valore = 0; // busy
  else {
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore = 1 ; // free
  if not_empty(sem.coda){
     ready(get(sem.coda));
     schedule();}
}
```

Nel semaforo binario, se un processo vuole accedere alla risorsa esegue un'operazione wait() sul semaforo e decrementa il valore del semaforo da 1 a 0. Quando rilascia la risorsa, esegue un'operazione signal() sul semaforo e incrementi il suo valore su 1. Se il valore del semaforo è 0 e un processo desidera accedere alla risorsa, esegue un'operazione wait() e si blocca fino a quando il processo corrente che utilizza le risorse rilascia la risorsa. L'accesso al campo deve essere atomico.

Il semaforo contatore ha la stessa struttura del mutex ma usa una logica diversa per il campo valore.

- (Valore > 0) denota disponibilità non esaurita
- (Valore < 0) denota richieste pendenti

Il valore iniziale denota la capacità massima della risorsa.

```
void P(struct sem){
  sem.valore -- ;
  if (sem.valore < 0){
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore ++ ;
  if (sem.valore <= 0){
     ready(get(sem.coda));
     schedule();}
}
```

L'uso di semafori a livello di programma è ostico e rischioso. Il posizionamento improprio delle P può causare situazioni di blocco infinito (deadlock) o anche esecuzioni erronee di difficile verifica (race condition).

È indesiderabile lasciare all'utente il pieno controllo di strutture così delicate.

Esempio 1

```
#define N …              /* posizioni del contenitore    */
typedef int semaforo;    /* P decrementa, V incrementa,
                            il valore 0 blocca la P      */
semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){                    void consumatore(){
   int prod;                             int prod;
   while(1){                             while(1){
      prod = produci();                     P(&non-vuoto);
      P(&non-pieno);                        P(&mutex);
      P(&mutex);                            prod = preleva();
      inserisci(prod);                      V(&mutex);
      V(&mutex);                            V(&non-pieno);
      V(&non-vuoto);                        consuma(prod);
   }                                     }
}                                     }
```

Il corretto ordinamento di P e V è critico!

Un diverso ordinamento delle P nel codice utente di Esempio 1 potrebbe causare situazioni di blocco infinito (deadlock)

Codice del produttore

```
P(&mutex);      // accesso esclusivo al contenitore
P(&non-pieno); // attesa spazi nel contenitore
```

In questo modo il consumatore non può più accedere al contenitore per prelevarne prodotti, facendo spazio per l'inserzione di nuovi → stallo = deadlock


```
#define N …              /* posizioni del contenitore    */
typedef int semaforo;    /* P decrementa, V incrementa,
                            il valore 0 blocca la P      */
semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){                    void consumatore(){
   int prod;                             int prod;
   while(1){                             while(1){
      prod = produci();                     P(&non-vuoto);
      P(&mutex);                            P(&mutex);
      P(&non-pieno);                        prod = preleva();
      inserisci(prod);                      V(&mutex);
      V(&mutex);                            V(&non-pieno);
      V(&non-vuoto);                        consuma(prod);
   }                                     }
}                                     }
```

Il corretto ordinamento di P e V è critico!

Linguaggi evoluti di alto livello offrono strutture esplicite di controllo delle regioni critiche, originariamente dette monitor, che definisce la regione critica. Il compilatore (non il programmatore!) inserisce il codice necessario al controllo degli accessi. Esso è un aggregato di sottoprogrammi, variabili e strutture dati. Solo i sottoprogrammi del monitor possono accederne le variabili interne e solo un processo alla volta può essere attivo entro il monitor. Proprietà garantita dai meccanismi del supporto a tempo di esecuzione del linguaggio di programmazione concorrente e il codice necessario è inserito dal compilatore direttamente nel programma eseguibile.

La garanzia di mutua esclusione da sola può non bastare per consentire sincronizzazione intelligente.
Due procedure operanti su variabili speciali (non contatori!) dette condition variables, consentono di modellare condizioni logiche specifiche del problema

- $Wait(< cond >)$ // forza l'attesa del chiamante
- $Signal(< cond >)$ // risveglia il processo in attesa

Il segnale di risveglio non ha memoria e va perso se nessuno lo attende,

1) La primitiva Wait permette di bloccare il chiamante qualora le condizioni logiche della risorsa non consentano l'esecuzione del servizio
– Contenitore pieno per il produttore
– Contenitore vuoto per il consumatore
2) La primitiva Signal notifica il verificarsi della condizione attesa al (primo) processo bloccato, risvegliandolo
– Il processo risvegliato compete con il chiamante della Signal per il possesso della CPU.
3) Wait e Signal sono invocate in mutua esclusione. Non si può verificare race condition ed è diverso dunque da sleep e wakeup visti in precedenza

Esempio 2

```
monitor PC
 condition non-vuoto, non-pieno;
 integer contenuto := 0;
 procedure inserisci(prod : integer);
 begin
  if contenuto = N then wait(non-pieno);
  <inserisci nel contenitore>;
  contenuto := contenuto + 1;
  if contenuto = 1 then signal(non-vuoto);
 end;
 function preleva : integer;
 begin
  if contenuto = 0 then wait(non-vuoto);
  preleva := <preleva dal contenitore>;
  contenuto :=  contenuto - 1;
  if contenuto = N-1 then signal(non-pieno);
 end;
end monitor;
```

```
procedure Produttore;
begin
 while true do begin
  prod := produci;
  PC.inserisci(prod);
 end;
end;
```

```
procedure Consumatore;
begin
 while true do begin
  prod := PC.preleva;
  consuma(prod);
 end;
end;
```

Monitor e semafori non sono utilizzabili per realizzare scambio di informazione tra elaboratori. Vediamo l'esempio del *message passing/scambio di messaggi*.

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

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

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

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

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

Il message passing è una tecnica per invocare un comportamento (cioè l'esecuzione di un programma) su un computer. A differenza della tecnica tradizionale di chiamare un programma per nome, il passaggio di messaggi utilizza un modello a oggetti per distinguere la funzione generale dalle implementazioni specifiche. Il programma che invoca invia un messaggio e si affida all'oggetto per selezionare ed eseguire il codice appropriato.

Per sincronizzare gruppi di processi, usiamo le barriere, che sono attività cooperative suddivise in fasi ordinate. La barriera blocca tutti i processi che la raggiungono fino all'arrivo dell'ultimo. Si applica indistintamente ad ambiente locale e distribuito e non comporta scambio di messaggi esplicito. L'avvenuta sincronizzazione dice implicitamente ai processi del gruppo che tutti hanno raggiunto un dato punto della loro esecuzione.

Metodo per valutare l'efficacia e l'eleganza di modelli e meccanismi per la sincronizzazione tra processi

- Filosofi a cena : accesso esclusivo a risorse limitate
- Lettori e scrittori : accessi concorrenti a basi di dati
- Barbiere che dorme : prevenzione di race condition

Scritto da Gabriel

- (Produttore e Consumatore)

Problemi pensati per rappresentare tipiche situazioni di rischio

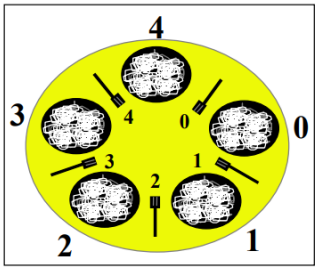
- Stallo con blocco (deadlock)
- Stallo senza blocco (starvation)
- Esecuzioni non predicibili (race condition)

Esaminiamoli singolarmente:

1) Filosofi a cena

- N filosofi sono seduti a un tavolo circolare
- Ciascuno ha davanti a se 1 piatto e 1 posata alla propria destra
- Ciascun filosofo necessita di 2 posate per mangiare
- L'attività di ciascun filosofo alterna pasti a momenti di riflessione

Soluzione A con stallo (*deadlock*)




```
void filosofo (int i){
  while (TRUE) {
    medita();
    P(f[i]);
    P(f[(i+1)%N]);
    mangia();
    V(f[(i+1)%N]);
    V(f[i]);
  };
}
```

L'accesso alla prima forchetta non garantisce l'accesso alla seconda!

Ogni forchetta modellata come un semaforo binario

Soluzione B con stallo (*starvation*)

```
void filosofo (int i){
while (TRUE) {
  OK = FALSE;
  medita();
  while (!OK) {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
     sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = TRUE;
   };
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
}
```

Errato: f contiene semafori che sono strutture dati

Un'attesa a durata costante difficilmente genera una situazione differente

Il problema ammette diverse soluzioni

1) Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a entrambe le forchette → Funzionamento garantito
2) In soluzione B, ciascun processo potrebbe attendere un tempo casuale invece che fisso → Funzionamento non garantito
3) Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività → Funzionamento garantito

Condizioni necessarie e sufficienti per uno stallo

– Accesso esclusivo a risorsa condivisa
– Accumulo di risorse • I processi possono accumulare nuove risorse senza doverne rilasciare altre
– Inibizione di prerilascio • Il possesso di una risorsa deve essere rilasciato volontariamente
– Condizione di attesa circolare • Un processo attende una risorsa in possesso del successivo processo in catena

Scritto da Gabriel

Prevenzione dello stallo

– Prevenzione
  - Impedire almeno una delle condizioni precedenti

– Riconoscimento e recupero
  - Ammettere che lo stallo si possa verificare
  - Essere in grado di riconoscerlo
  - Possedere una procedura di recupero (sblocco)

– Indifferenza
  - Considerare trascurabile la probabilità di stallo e non prendere alcuna precauzione contro di esso

– Attesa circolare
  - Difficile da rilevare e complessa da evitare o sciogliere

Prevenzione sulle richieste di accesso

1) A tempo d'esecuzione
- A ogni richiesta di accesso si verifica se questa possa portare allo stallo
  In caso affermativo non è però chiaro cosa convenga fare e la verifica a ogni richiesta è un onere molto pesante
- Assai oneroso → Occorre bloccare periodicamente l'avanzamento del sistema per analizzare lo stato di tutti i processi e verificare se quelli in attesa costituiscono una lista circolare chiusa
- Lo sblocco di uno stallo comporta la terminazione forzata di uno dei processi in attesa • Il rilascio delle risorse liberate sblocca la catena di dipendenza circolare

2) Prima dell'esecuzione
- All'avvio di ogni processo si verifica quali risorse essi dovranno utilizzare così da ordinarne l'attività in maniera conveniente

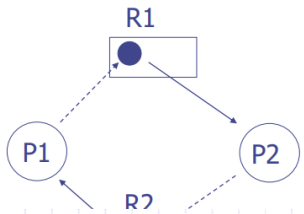
3) Staticamente
- Può essere un problema non risolvibile

Esercizi sincronizzazione

Grafo di Assegnazione delle Risorse

- Si consideri un sistema con 2 processi (P1, P2), e 2 tipologie di risorse (R1, R2) con disponibilità: 1 risorsa di tipo R1 e 1 risorsa di tipo R2.
- Si consideri la seguente cronologia di richieste:
  1) P2 richiede R1
  2) P1 richiede R2

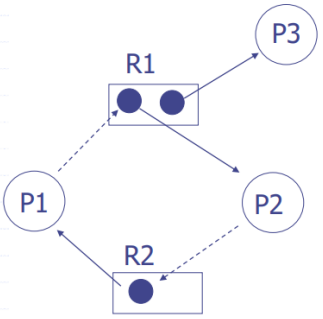
- Il grafo di allocazione delle risorse sarà il seguente:



Si forma un ciclo: dunque il sistema è in **stallo**

- Si consideri un sistema con 3 processi (P1, P2, P3), e 2 tipologie di risorse (R1, R2) con disponibilità: 2 risorse di tipo R1 e 1 risorsa di tipo R2.
- Si consideri la seguente cronologia di richieste:
  1) P2 richiede R1
  2) P1 richiede R2
  3) P2 richiede R2
  4) P3 richiede R1
  5) P1 richiede R1
- Verificare se alla fine il sistema si trovi in condizioni di stallo

- Il grafo di allocazione delle risorse sarà il seguente:



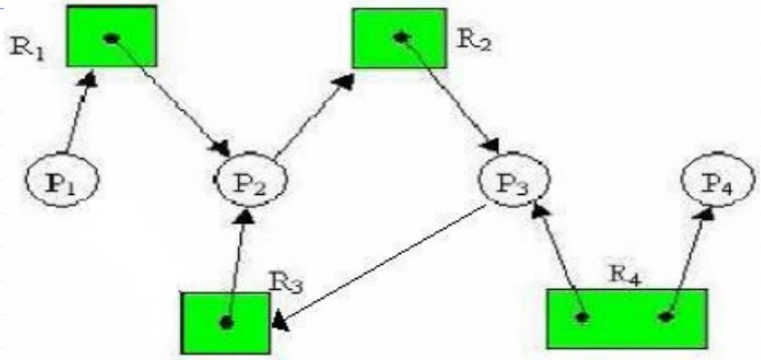
Si forma un ciclo ma il sistema NON è in **stallo**

Infatti un elemento del ciclo (R1) ha molteplicità 2 e una delle due istanze è fuori da cicli: P3 può terminare e rilasciare una istanza di R1 che viene così assegnata a P1

Verifica Deadlock

- Si consideri un sistema con 4 processi (P1, P2, P3, P4), e 4 tipologie di risorse (R1,R2,R3,R4) con disponibilità: 1 risorsa di tipo R1, 1 risorsa di tipo R2, 1 risorsa di tipo R3, 2 risorse di tipo R4.
- Si assuma che:
  - ogni volta che un processo richieda una risorsa libera, questa venga assegnata al processo richiedente;
  - ogni volta che un processo richieda una risorsa già occupata, il richiedente deve attendere che la risorsa si liberi prima di impossessarsene (coda FIFO di attesa)
- Si consideri la seguente successione cronologica di richieste e rilasci di risorse:
  1) P2 richiede R1,R2,R3 ⇨ Le risorse sono libere, quindi vengono assegnate, compresa R2
  2) P3 richiede R2,R4 ⇨ P3 ottiene R4, ma non R2 che è assegnata a P2
  3) P2 rilascia R2 ⇨ R2, rilasciata, viene assegnata subito a P3 che l'aveva richiesta
  4) P4 richiede R4 ⇨ Assegnata, poiché vi sono 2 istanze di R4
  5) P1 richiede R1 ⇨ R1 è già assegnata a P2 e la richiesta non può essere soddisfatta
  6) P2 richiede R2 ⇨ R2 è già assegnata a P3 e la richiesta non può essere soddisfatta
  7) P3 richiede R3 ⇨ R3 è già assegnata a P2 e la richiesta non può essere soddisfatta
- Verificare se alla fine il sistema si trovi in condizioni di stallo

Sol – Grafo Allocazione Risorse



- Alla fine delle operazioni descritte, il grafo di allocazione delle risorse appare come in figura. In tale grafo esiste un ciclo di richieste/assegnazioni che coinvolge P2, R2, P3, R3: pertanto, il sistema **è in stallo**.

Verifica Deadlock

- Dato il sistema descritto dalla seguente rappresentazione insiemistica di assegnazione delle risorse:

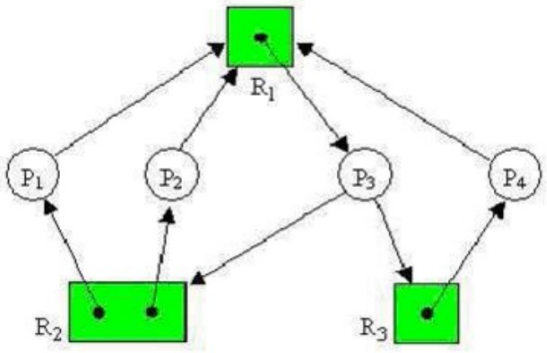
  P = P1, P2, P3, P4

  R = R1, R2, R2, R3

$$E = P_1 \to R_1, P_2 \to R_1, P_3 \to R_2, P_3 \to R_3, P_4 \to R_1$$
$$R_1 \to P_3, R_2 \to P_1, R_2 \to P_2, R_3 \to P_4$$

- si verifichi, anche tramite analisi del relativo grafo di allocazione delle risorse, se il sistema si trovi in condizione di stallo.
- Successivamente, si discuta per ogni processo se la sua rimozione forzata (con conseguente liberazione delle risorse eventualmente in suo possesso ed annullamento delle sue richieste di risorse) porti ad un cambiamento della situazione precedentemente rilevata.

Sol – Grafo Allocazione Risorse



Sol – Grafo Allocazione Risorse

- La figura riporta la versione grafica della rappresentazione insiemistica data, al cui interno si distinguono 3 percorsi chiusi:

$$\text{percorso 1: } R_2 \to P_1 \to R_1 \to P_3 \to R_2$$
$$\text{percorso 2: } R_2 \to P_2 \to R_1 \to P_3 \to R_2$$
$$\text{percorso 3: } R_3 \to P_4 \to R_1 \to P_3 \to R_3$$

- I percorsi 1 e 2 contengono risorse a molteplicità > 1, e quindi per ciascuno di essi occorre una specifica analisi di dettaglio. Il percorso 3 invece contiene solo risorse unarie, per cui possiamo affermare che i processi P3 e P4 sono sicuramente in stallo. Essendo il processo P3, che è in stallo, presente anche nei percorsi 1 e 2, possiamo concludere che anche i processi P1 e P2 si trovano in situazione di stallo. Pertanto, in definitiva, l'intero sistema è in **stato di stallo**.

- Analizziamo ora la situazione che seguirebbe all'eliminazione di singoli processi dal sistema:
  - eliminiamo P1: la situazione non cambierebbe in quanto l'anello di sicuro stallo (percorso 3) non verrebbe intaccato; inoltre anche il processo P2 rimarrebbe bloccato dato che la risorsa R1 da esso richiesta è impegnata nel percorso 3.
  - eliminiamo P2: la situazione è analoga alla precedente: gli altri processi rimarrebbero bloccati.
  - eliminiamo P3: la situazione cambierebbe, dato che R1, posseduta dal processo P3 eliminato, diverrebbe libera; i processi P1, P2 e P4 hanno tutti una richiesta pendente per tale risorsa, ed indipendentemente dall'ordine con il quale possano venir soddisfatte tali richieste, i processi riprenderanno sequenzialmente ad avanzare, conseguentemente liberando il sistema dal precedente stato di stallo.
  - eliminiamo P4: la situazione non cambierebbe in quanto la risorsa R3, rilasciata dal processo eliminato, verrebbe assegnata al processo P3; che però resterebbe ancora bloccato stante la perdurante indisponibilità della risorsa R2, con il conseguente blocco dei percorsi 1 e 2.

Verifica Stato Sicuro

- Un sistema si compone di 4 processi e 5 risorse condivise a diversa molteplicità.
- All'istante considerato, lo stato di allocazione delle risorse a processi e i loro bisogni massimi previsti sono riportati in Tabella.

Processo	Risorse $(R_1, R_2, R_3, R_4, R_5)$	
	Allocazione attuale	Richiesta massima
A	1 0 2 1 1	1 1 2 1 4
B	2 0 1 1 1	2 2 3 2 1
C	1 1 0 1 0	2 1 4 1 0
D	1 1 1 1 0	1 1 3 2 1

- Assumendo che il vettore di disponibità delle risorse allo stato corrente sia uguale a [0 0 2 1 2], si discuta se il sistema sia in uno **stato sicuro** (che eviti *deadlock* anche nel caso in cui tutti i processi richiedano simultaneamente il massimo delle risorse per loro richiedibili)

Verifica Deadlock – Sol

- Vettore disponibilità [0 0 2 1 2]

Processo	Risorse $(R_1, R_2, R_3, R_4, R_5)$	
	Allocazione attuale	Richiesta massima
A	1 0 2 1 1	1 1 2 1 4
B	2 0 1 1 1	2 2 3 2 1
C	1 1 0 1 0	2 1 4 1 0
D	1 1 1 1 0	1 1 3 2 1

- il processo D può essere eseguito fino alla fine
- Quando ha finito, il vettore delle risorse disponibili è [1 1 3 2 2]
- Sfortunatamente però ora il sistema potrebbe andare incontro a ***deadlock*** qualora i processi rimanenti chiedessero effettuassero la loro "Richiesta massima" di risorse
- Dunque la situazione iniziale non rappresentava uno **stato sicuro**

Algoritmo del banchiere

- Evita le situazioni di stallo
  - Come anche l'uso dei grafi di allocazione
- Detto così perché, simile a una banca (virtuosa), non versa mai tutte le risorse disponibili al fine di poter sempre soddisfare i propri clienti
- Richiede che i processi dichiarino il massimo quantitativo di risorse che useranno
- Ad ogni nuova richiesta verifica se l'assegnazione lascerebbe il sistema in uno stato sicuro
  - Se così è le risorse vengono assegnate
  - Viceversa il processo deve attendere

L'algoritmo del banchiere è un algoritmo per l'allocazione delle risorse e per evitare i deadlock che verifica la sicurezza simulando l'allocazione di una quantità massima predeterminata di tutte le risorse; quindi, effettua un controllo dello "stato s" per verificare la presenza di eventuali attività, prima di decidere se l'allocazione deve essere autorizzata a continuare.

L'algoritmo del banchiere si chiama così perché viene utilizzato nel sistema bancario per verificare se un prestito può essere concesso o meno a una persona. Supponiamo che in una banca ci sia un numero n di correntisti e che la somma totale dei loro soldi sia S. Se una persona chiede un prestito, la banca sottrae prima l'importo del prestito dal totale dei soldi che la banca ha e se l'importo rimanente è maggiore di S, allora solo il prestito viene concesso. Questo viene fatto perché se tutti i correntisti vengono a ritirare il loro denaro, la banca può farlo facilmente.
In altre parole, la banca non stanzierebbe mai il proprio denaro in modo tale da non poter più soddisfare le esigenze di tutti i suoi clienti. La banca cercherà di essere sempre al sicuro.

Realizzaz. algoritmo del banchiere

- N = num processi nel sistema; M = num risorse
- Servono alcune strutture dati:
  - **Disponibili:** matrice M; *Disponibili [j] = k* significa che sono disponibili k risorse del tipo *Rj*.
  - **Massimo:** matrice N x M; *Massimo [i,j] = k* significa che il processo *Pi* può richiedere un massimo di *k* istanze della risorsa di tipo *Rj*.
  - **Assegnate:** matrice N x M; *Assegnate [i,j] = k* significa che al processo *Pi* sono state attualmente assegnate *k* istanze della risorsa di tipo *Rj*.
  - **Necessità:** matrice N x M; *Necessità [i,j] = k* significa che il processo *Pi* per completare il suo compito può avere bisogno di altre *k* istanze della risorsa di tipo *Rj*.
    - Si noti che *Necessità [i,j] = Massimo [i,j] - Assegnate [i,j]*

- Algoritmo di verifica della sicurezza:
  1. Sia *Lavoro [j]= Disponibili [j]* per j = 0, 1, …, M-1 e *Fine [i] = falso* per *i = 0, 1, …, N-1*
  2. Si cerchi indice *i* tale che valgano entrambe:
     a) *Fine [i] == falso*
     b) *Necessità [i, j]<= Lavoro [j]* per j = 0, 1, …, M-1
     Se tale *i* non esiste, esegue passo 4.
  3. *Lavoro [j] = Lavoro [j] + Assegnate [i, j]*
     Fine [i] = vero
     torna al passo 2
  4. Se *Fine [i] == vero* per ogni *i*, allora il sistema è in stato sicuro
- O (M x N x N)

- Algoritmo di richiesta delle risorse
  - Se *Richieste [i] <= Necessità [i]*, allora esegue passo 2, altrimenti ERRORE per superato num max di richieste
  - Se *Richieste [i] <= Disponibili* allora esegue passo 3, altrimenti *Pi* deve attendere che si liberino delle risorse
  - Simula l'assegnazione delle risorse richieste al processo *Pi*, modificando lo stato di assegnazione delle risorse come segue:
    - *Disponibili = Disponibili – Richieste [i]*
    - *Assegnate [i] = Assegnate [i] + Richieste [i]*
    - *Necessità [i] = Necessità [i] - Richieste [i]*
  - Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al processo *Pi* si assegnano le risorse richieste; altrimenti *Pi* deve attendere.

Esempio algoritmo del banchiere

- Un sistema ha 4 processi (A, B, C, D) e 5 risorse (R1, R2, R3, R4, R5) da ripartire. L'attuale allocazione e i bisogni massimi sono i seguenti:

Processo	*Allocate*	*Massimo*
A	1 0 2 1 1	3 1 2 1 3
B	2 0 1 1 1	3 3 4 2 1
C	1 1 0 1 0	2 1 4 1 0
D	1 1 1 1 0	1 1 3 2 1

- Considerando il vettore delle risorse disponibili uguale a [0 1 3 1 2], e utilizzando l'Algoritmo del Banchiere, si discuta se il sistema sia in uno stato sicuro.

- Calcoliamo la matrice delle *Necessità*:

Processo	*Allocate*	*Massimo*	*Necessità*
A	1 0 2 1 1	3 1 2 1 3	2 1 0 0 2
B	2 0 1 1 1	3 3 4 2 1	1 3 3 1 0
C	1 1 0 1 0	2 1 4 1 0	1 0 4 0 0
D	1 1 1 1 0	1 1 3 2 1	0 0 2 1 1

- Inizialmente il vettore delle risorse disponibili è uguale a [0 1 3 1 2], quindi *Lavoro* è [0 1 3 1 2].
- Inoltre *Fine* è [0 0 0 0]

Processo	*Allocate*	*Massimo*	*Necessità*
A	1 0 2 1 1	3 1 2 1 3	2 1 0 0 2
B	2 0 1 1 1	3 3 4 2 1	1 3 3 1 0
C	1 1 0 1 0	2 1 4 1 0	1 0 4 0 0
D	1 1 1 1 0	1 1 3 2 1	0 0 2 1 1

- Il proc. D potrebbe essere eseguito fino alla fine poiché ciascun elemento di Necessità è minore o uguale al corrispondente elemento di *Lavoro* [0 1 3 1 2]
  Al termine *Lavoro* diventa [1 2 4 2 2], in quanto somma del vettore precedente e delle risorse allocate a D precedentemente [1 1 1 1 0].
  Inoltre *Fine* diventa [0 0 0 1].

Processo	*Allocate*	*Massimo*	*Necessità*
A	1 0 2 1 1	3 1 2 1 3	2 1 0 0 2
B	2 0 1 1 1	3 3 4 2 1	1 3 3 1 0
C	1 1 0 1 0	2 1 4 1 0	1 0 4 0 0
D	1 1 1 1 0	1 1 3 2 1	0 0 2 1 1

- Dopo, il proc. C potrebbe essere eseguito e al suo completamento, il vettore delle risorse disponibili (vettore *Lavoro* nell'algoritmo) diventa [2 3 4 3 2] mentre *Fine* diventa [0 0 1 1].
- Questo permette di eseguire e terminare il processo A ottenendo [3 3 6 4 3] come *Lavoro* e [1 0 1 1] come *Fine*.
- Così si può eseguire e terminare anche il processo B.
- **Il sistema è quindi in uno stato sicuro.**

- Semafori: variabili intere
  - contano il numero di richieste pendenti
  - il valore è 0 se non ci sono risorse disponibili a servire richieste (che quindi diventeranno pendenti) e > 0 altrimenti
- Due operazioni atomiche standard *Down* e *Up* (P e V)
  - down(S) … equivalente di P(S)
    - se S > 0 allora S= S - 1 ed il processo continua l'esecuzione
    - se S==0 ed il processo si blocca senza completare la primitiva
  - up(S) … equivalente di V(S)
    - se ci sono processi in attesa di completare la down su quel semaforo (e quindi necessariamente S == 0) uno di questi viene svegliato e S rimane a 0, altrimenti S viene incrementato;
    - in caso contrario (S > 0), allora S=S + 1

Semafori (2)

Soluzione del Produttore/ Consumatore con semafori

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

void producer(void)
{
    int item;

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


void consumer(void)
{
    int item;

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

Esercizi Sincronizzazione

Monitor (1)

- Oggetti (Strutture dati + procedure per accedervi)
- Mutua esclusione nell'esecuzione delle procedure
- Variabili di Condizione + wait() e signal()
- wait(X)
  - sospende sempre il processo che la invoca in attesa di una signal(X)
- signal(X)
  - sveglia uno dei processi in coda su X
  - se nessun processo è in attesa va persa
  - deve essere eseguita solo come ultima istruzione prima di uscire dal monitor (il processo svegliato ha l'uso esclusivo del monitor)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- Caratteristiche principali dei monitor Java
  - ME nell'esecuzione dei metodi synchronized
  - non ci sono variabili di condizione
  - wait(), notify() simili a sleep() , wakeup()
  - è possibile svegliare tutti i processi in attesa

- Schema di soluzione Produttore/Consumatore
  - ad ogni istante solo una procedura del monitor è in esecuzione
  - il buffer ha *N* posizioni

```
public class ProducerConsumer {
    static final int N = 100;                   // constant giving the buffer size
    static producer p = new producer( );        // instantiate a new producer thread
    static consumer c = new consumer( );// instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );  // instantiate a new monitor
    public static void main(String args[ ]) {
        p.start( );                             // start the producer thread
        c.start( );                             // start the consumer thread
    }
    static class producer extends Thread {
        public void run( ) {                    // run method contains the thread code
            int item;
            while (true) {                      // producer loop
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }     // actually produce
    }
    static class consumer extends Thread {
        public void run( ) {                    run method contains the thread code
            int item;
            while (true) {                      // consumer loop
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }     // actually consume
    }
```

Soluzione per Produttore/Consumatore in Java (parte 1)

```
    static class our_monitor {                  // this is a monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // counters and indices
        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
            buffer [hi] = val;                  // insert an item into the buffer
            hi = (hi + 1) % N;                  // slot to place next item in
            count = count + 1;                  // one more item in the buffer now
            if (count == 1) notify( );          // if consumer was sleeping, wake it up
        }
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo];                  // fetch an item from the buffer
            lo = (lo + 1) % N;                  // slot to fetch next item from
            count = count - 1;                  // one few items in the buffer
            if (count == N - 1) notify( );      // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Soluzione per Produttore/Consumatore in Java (parte 2)

Scambio Messaggi (1)

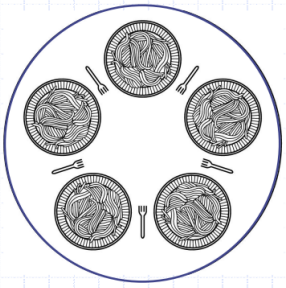
- Non richedono accesso a supporti di memorizzazione comune
- primitive base
  - send(destination,&msg)
  - receive(source, &msg)
- decine di varianti, nel nostro caso :
  - la receive blocca automaticamente se non ci sono messaggi
  - i messaggi spediti ma non ancora ricevuti sono bufferizzati dal SO
  - Tipo mailbox

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

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

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

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

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

Sol. Produttore/Consumatore con *N* messaggi

I filosofi a cena (1)

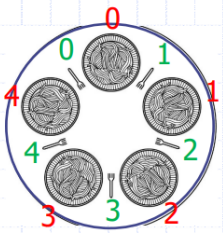
- I filosofi mangiano e pensano
- Per mangiare servono due forchette
- Ogni filosofo prende una forchetta per volta
- Come si può prevenire il *deadlock*



```
Filosofo(i) {
    while(1) {
        <pensa>
        if(i == X) {
            P(f [(i+1)%N]);
            P(f [i]);
        } else {
            P(f [i]);
            P(f [(i+1)%N]);
        }
        <mangia>
        V(f [i]);
        V(f [(i+1)%N]);
    }
}
```

Inizializzazione:
int semaforo **f[i]** = 1;

Per evitare deadlock inseriamo un filosofo "***mancino***": ad esempio, il filosofo ***X***



(4+1)%5 = 0

5 filosofi a cena coi monitor

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

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

      filosofo[(n+1)%5].notify();
   }
}
Filosofo(i){
   while (true){
      <pensa>
      Tavolo.raccogli(i);
      <mangia>
      Tavolo.deposita(i);
   }
}
```

Il problema dei lettori e scrittori (1)

- Un database molto esteso (db)
  - es. prenotazioni aeree ...
- Un insieme di processi che devono leggere o scrivere in db
- Più lettori possono accedere contemporaneamente a db
- Gli scrittori devono avere accesso esclusivo a db
- I lettori hanno precedenza sugli scrittori
  - se uno scrittore chiede di accedere mentre uno o più lettori stanno accedendo a db, lo scrittore deve attendere che i lettori abbiano finito

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc - 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read();        /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data();        /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base();      /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

Con il semaforo mutex gestisco la mutua esclusione sulle variabili condivise

Con il semaforo db gestisco l'alternanza lettori-scrittori sul database; solo il primo lettore di una sequenza deve passare attraverso questo semaforo

29

Il barbiere sonnolento (1)

- Un barbiere dorme sulla poltrona di lavoro quando non ci sono clienti
- All'arrivo del primo cliente, il barbiere si sveglia, lo fa sedere sulla poltrona e lo serve
- Se arrivano altri clienti mentre il barbiere è al lavoro, si siedono su una sedia in attesa
- C'è un numero massimo di sedie per l'attesa (es. 5) oltre il quale ulteriori clienti se ne vanno dal negozio senza attendere di essere serviti
- Finito di servire un cliente, questi esce dal negozio e un altro cliente in attesa viene servito

Il barbiere sonnolento (2): soluz.

Il semaforo **customer** viene utilizzato per addormentare il barbiere in assenza di clienti e risvegliarlo quando arrivano

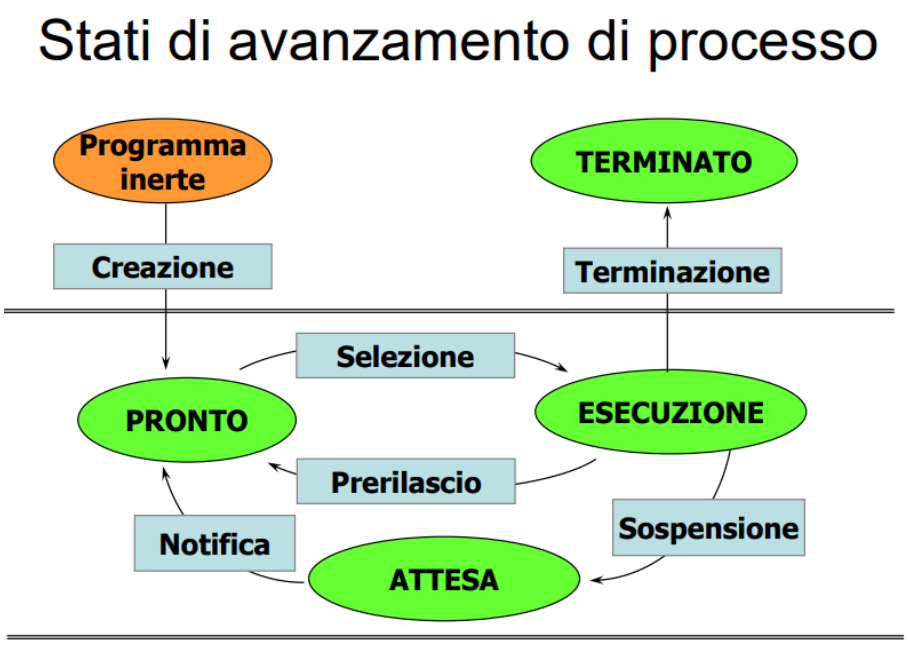
Il semaforo **barbers** viene utilizzato per occupare il barbiere e addormentare un cliente se il barbiere è già occupato a servire altri

Il semaforo **mutex** viene utilizzato per garantire la mutua esclusione sull'uso della variabile condivisa **waiting**

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

typedef int semaphore;              /* use your imagination */

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

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

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

Politiche di ordinamento

Ogni processo è associato a un descrittore chiamato Process Control Block che ne specifica le caratteristiche distintive e relaziona il processo alla sua macchina virtuale:
– Identificatore del processo
– Contesto di esecuzione del processo (tutte le informazioni necessarie a ripristinarne lo stato d'esecuzione dopo una sospensione o un prerilascio)
– Stato di avanzamento del processo (quindi, puntatore (d)alla lista del processi in quello stato)
– Priorità (iniziale, corrente)
– Diritti di accesso alle risorse e altri eventuali privilegi
– Discendenza familiare (puntatore al PCB del processo genitore e degli eventuali processi figli)
– Puntatore alla lista delle risorse assegnate al processo



Una decisione di scheduling è necessaria:
– alla creazione di un processo (processo genitore o figlio?)
– alla terminazione di un processo (chi lo sostituisce?) – quando il processo si blocca (in attesa di I/O, o di semaforo,…)

Importante ad esempio per evitare *inversion priority* se un processo importante attende che uno meno importante rilasci sezione critica. Lo scheduler può non avere info necesssarie ad ogni k-esima occorrenza dell'interrupt periodico (50 – 60 Hz)

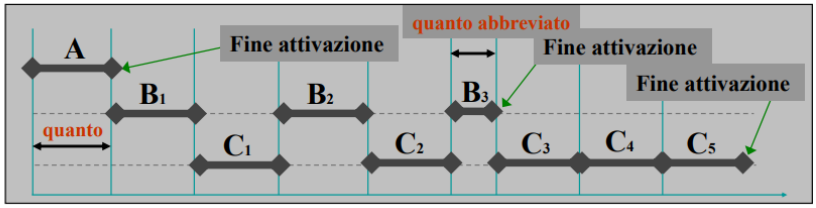
Diversi metodi per decidere come alternare i processi in esecuzione
– Scambio cooperativo (cooperative / non pre-emptive switch) → Il processo in esecuzione decide da solo quando cedere il controllo
– Scambio a prerilascio (inconsapevole) → Il processo in esecuzione viene rimpiazzato da un processo appena arrivato con maggiore importanza (priority-based pre-emptive → sistemi a tempo reale) o all'esaurimento del quanto di tempo (time-sharing pre-emptive → sistemi interattivi (Unix → Linux, Windows NT). Questo necessita di clock e sincronizzazione.
Il prerilascio si realizza tramite un meccanismo esterno all'esecuzione dei processi (un dispositivo (p.es., orologio) solleva una interruzione ed un gestore software la identifica e, se necessario, la notifica allo scheduler).

Lo scheduler è il componente del nucleo che decide l'ordinamento dei processi. È progettato prima dei processi che è chiamato a governare, adottando metriche diverse per i singoli sistemi (bath, no preemption, interattivi, sì preemption, ecc). Bisogna perciò rendere il suo operato parametrico rispetto a specifici attributi assegnati ai processi per non doverlo cambiare al variare delle applicazioni. Basta configurare opportunamente gli attributi dei processi.
Il dispatcher è il componente del nucleo che attua le scelte di ordinamento dei processi ed opera su mandato dello scheduler. Deve essere molto efficiente perché opera a ogni scambio di contesto (context switch). Salva il contesto del processo in uscita, installa quello del processo in entrata e gli affida il controllo della CPU.

L'applicazione influenza le politiche di ordinamento tramite il valore degli attributi considerati dai meccanismi del nucleo (per determinare l'ordinamento dei processi e per influenzare l'attribuzione delle risorse). L'efficienza delle politiche scelte si misura in termini di
– Percentuale di impiego utile della CPU (spesso con più processi che il kernel e il tempo di esecuzione di scheduler e dispatcher è sottratto ai processi)
– Numero di processi avviati all'esecuzione per unità di tempo (misura di produttività (throughput))
– Durata di permanenza di un processo in stato di pronto (tempo di attesa)
– Tempo di completamento (turn-around)
– Reattività rispetto alla richiesta di avvio di un processo (tempo di risposta)

La garanzia di esecuzione dei processi dipende criticamente dalla politica di scambio adottata (es. lo scambio cooperativo non offre alcuna garanzia e gli utenti in genere richiedono equità di opportunità, cosiddetta fairness). I processi in stato di pronto sono registrati in una struttura detta lista dei pronti (ready list). La più semplice gestione della lista è con tecnica a coda (First-Come-First-Served, FCFS) in cui il primo processo a entrare in coda sarà anche il primo a essere avviato all'esecuzione. È molto facile da realizzare e da gestire.

- Imponendo divisione di tempo (*time sharing*) sulla politica *FCFS* si ottiene una tecnica di rotazione detta ***round-robin***
- Vediamo l'applicazione di un quanto di tempo 2 su tre processi **A**, **B** e **C** con tempo di esecuzione 2, 5, 10 rispettivamente

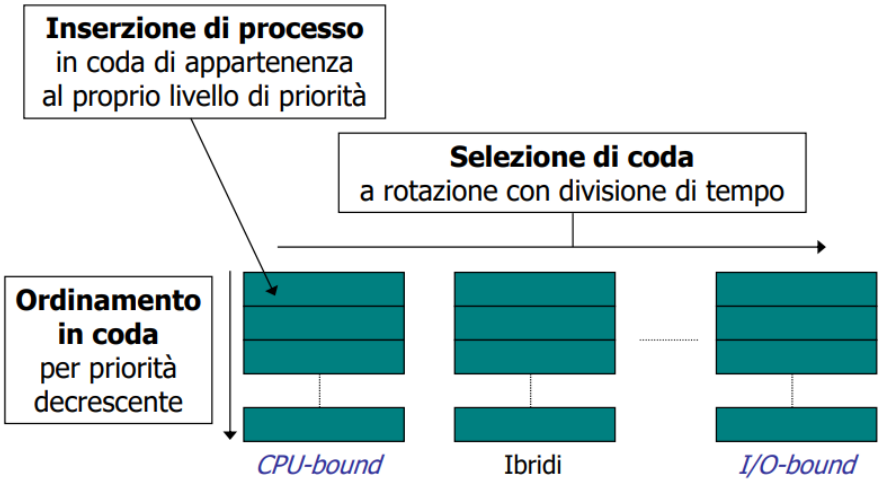


Esempio 1: politica di ordinamento a livelli - A rotazione con priorità

- Scelta di politica – 1
  - Assegnare un dato livello di privilegio a ogni singolo processo
- Meccanismo impiegato
  - Attributo rappresentato da una priorità statica o dinamica registrata nel PCB

- Scelta di politica – 2
  - Processi distinti sulla base di determinate caratteristiche
    - Note a priori, p.es.: *CPU-bound*, *I/O-bound*
    - Acquisite a tempo d'esecuzione, p.es.: tempo cumulato (di esecuzione o di attesa)
- Meccanismo impiegato
  - Rilevazione del valore di un dato campo del PCB

- Scelta di politica – 3
  - Coda ordinata a priorità per ciascuna classe di processi
  - Selezione di coda *round-robin*

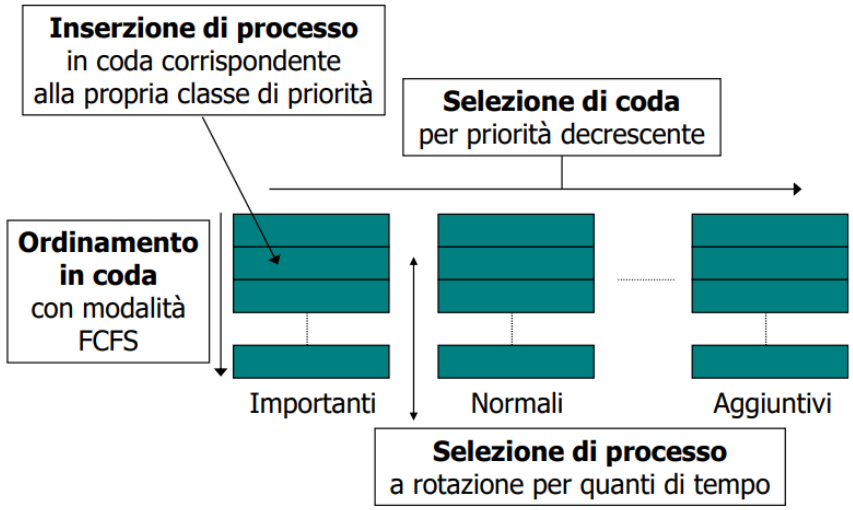


Esempio 2: politica di ordinamento a livelli - A priorità con rotazione

- Scelta di politica – 1
  - Assegnare un dato livello di privilegio a ogni singolo processo
- Meccanismo impiegato
  - Attributo rappresentato da una priorità statica o dinamica registrata nel PCB

- Scelta di politica – 2
  - Processi distinti sulla base di determinate caratteristiche
    - Statiche o dinamiche

- Scelta di politica – 3
  - Selezione di coda su base di priorità
  - Assegnazione di CPU con modalità *round-robin*



I meccanismi per realizzare scelte di ordinamento e gestione dei processi risiedono nel nucleo. Le politiche sono determinate fuori dal nucleo e sono decise nello spazio delle applicazioni, scegliendo quali valori assegnare ai parametri di configurazione dei processi considerati dai meccanismi di gestione.
Diverse classi di sistemi concorrenti richiedono politiche di ordinamento di processi specifiche , con tre classi generali

1) Sistemi “a lotti” (batch)
   • Ordinamento predeterminato; lavori di lunga durata e limitata urgenza; prerilascio non necessario
2) Sistemi interattivi
   • Grande varietà di attività; prerilascio essenziale
3) Sistemi in tempo reale
   • Lavori di durata ridotta ma con elevata urgenza; l’ordinamento deve riflettere l’importanza del processo; prerilascio possibile

Per tutti i sistemi, in generale, con le politiche di ordinamento si cerca di avere:
• Equità (fairness) nella distribuzione delle opportunità di esecuzione
• Coerenza (enforcement) nell’applicazione della politica a tutti i processi
• Bilanciamento nell’uso di tutte le risorse del sistema

In generale (si consideri che parliamo di *lavori* quando operiamo *senza prerilascio* e di *processi* quando operiamo *con prerilascio*):

- Per i sistemi a lotti:
  - Massimo prodotto per unità di tempo (throughput)
  - Massima rapidità di servizio per singolo lavoro (turn-around) – Media statistica
  - Massimo utilizzo delle risorse di elaborazione

  Politica di ordinamento FCFS (First Come First Served)
  - Senza prerilascio, senza priorità
  - Ordine di esecuzione = ordine di arrivo
  - Massima semplicità, basso utilizzo delle risorse

  Politica di ordinamento SJF (Shortest job first)
  - Senza prerilascio, richiede conoscenza dei tempi richiesti di esecuzione
  - Esegue prima il lavoro (job) più breve
  - Non è equo con i lavori non presenti all'inizio

  Politica di ordinamento SRTN (Shortest Remaining Time Next)
  - Variante di SJF con prerilascio
  - Esegue prima il processo più veloce a completare
  - Tiene conto di nuovi processi quando essi arrivano

- Per i sistemi interattivi:
  - Rapidità di risposta per singolo lavoro rispetto alla percezione dell'utente
  - Soddisfazione delle aspettative generali dell'utente

  Politica di ordinamento OQ : Ordinamento a quanti (Round Robin, RR)
  - Con prerilascio, senza priorità
  - Ogni processo esegue al più per un quanto alla volta
  - Lista circolare di processi

  Politica di ordinamento OQP : Ordinamento a quanti con priorità
  - Quanti diversi per livello di priorità (attribuendo priorità a processi e facendole eventualmente variare)

  Politica di ordinamento GP : Con garanzia per processo
  - Con prerilascio e con promessa di una data quantità di tempo di esecuzione (p.es. 1/n per n processi concorrenti). Le necessità di ciascun processo devono essere note (stimate) a priori
  - Esegue prima il lavoro maggiormente penalizzato rispetto alla promessa. Si ha una verifica periodica o a evento (soddisfacimento della promessa)

  Politica di ordinamento SG: Senza garanzia
  - Con prerilascio e priorità, opera sul principio della lotteria
    – Ogni processo riceve numeri da giocare
    – A priorità più alta corrispondono più numeri da giocare
    – A ogni scelta per assegnazione di risorsa, essa va al processo possessore del numero estratto
    – Le estrazioni avvengono periodicamente (= quanti) e/o a eventi (p.es. attesa di risorse non disponibili)
  - Comportamento impredicibile sul breve periodo, ma tende a stabilizzarsi statisticamente nel tempo

  Politica di ordinamento GU: Con garanzia per utente
  - Come GP ma con garanzia riferita a ciascun utente (possessore di più processi)

- Per i sistemi in tempo reale:
  - Rispetto delle scadenze temporali (deadline)
  - Predicibilità di comportamento (predictability)

I sistemi in tempo reale sono sistemi concorrenti nei quali il valore corretto deve essere prodotto entro un tempo fissato; oltre tale limite il valore prodotto ha utilità decrescente, nulla o addirittura negativa.

L'ordinamento (scheduling) di processi deve fornire garanzie di completamento adeguate ai processi. Deve essere analizzabile staticamente (predicibile) ed il caso peggiore è sempre quando tutti i processi sono pronti insieme per eseguire all'istante iniziale (critical instant).

– *Politica di ordinamento cyclic executive*

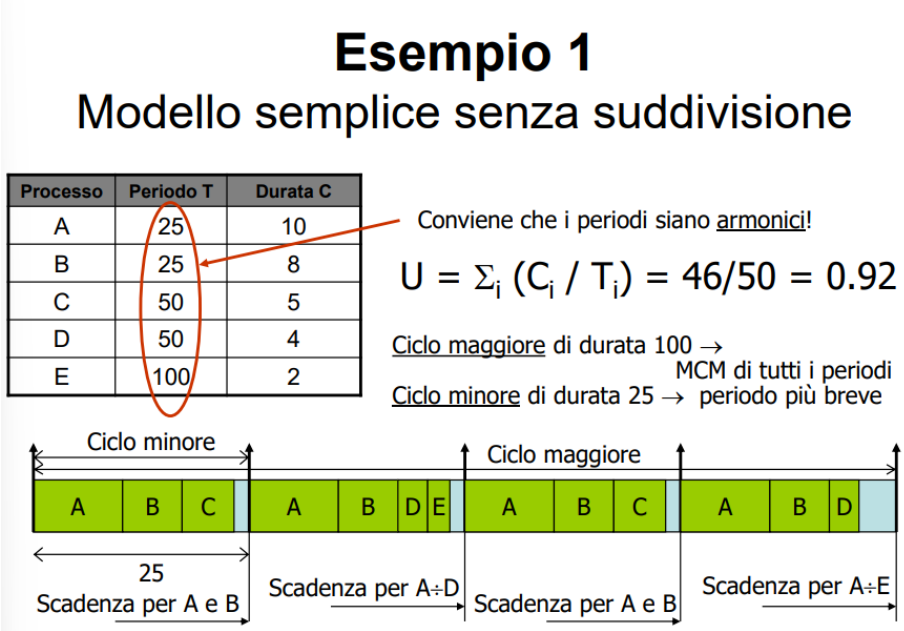
• L'applicazione consiste di un insieme fissato di processi periodici (ripetitivi) ed indipendenti con caratteristiche note

• Ciascun processo è suddiviso in una sequenza ordinata di procedure di durata massima nota

• L'ordinamento è costruito a tavolino come una sequenza di chiamate a procedure di processi fino al loro completamento

• Un ciclo detto maggiore (major cycle) racchiude l'invocazione di tutte le sequenze di tutti i processi

• Il ciclo maggiore è suddiviso in N cicli minori (minor cycle) di durata fissa che racchiude l'invocazione di specifiche sottosequenze



- *Politica di ordinamento a priorità fissa*

• Preferibilmente con prerilascio (a priorità)

– Processi periodici, indipendenti e noti

• Assegnazione di priorità secondo il periodo (rate monotonic)

– Per scadenza uguale a periodo (D = T), priorità maggiore per periodo più breve

– Test di ammissibilità sufficiente ma non necessario per n processi indipendenti

$$U = \sum_{i:=1}^{n} \left(\frac{C_i}{T_i} \right) \leq f(n) = n(2^{1/n} - 1)$$

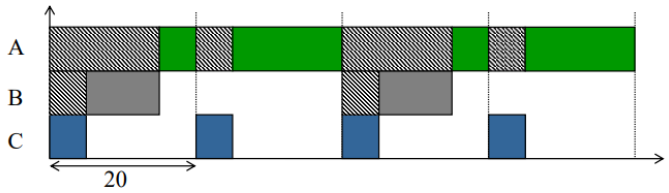
Dove U è il fattore di utilizzo, C_i è il tempo di calcolo del processo i, T_i è il tempo di rilascio

Esempio 2

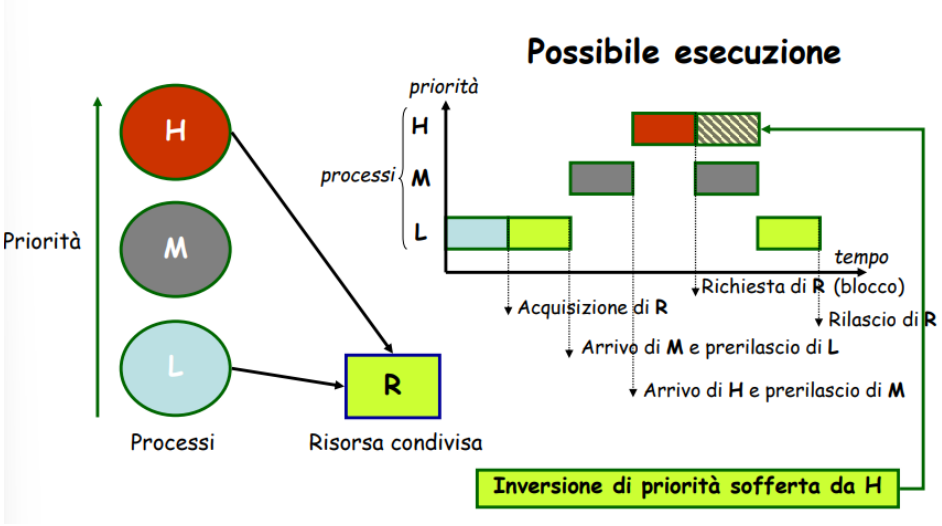
Caso semplice ordinamento a priorità

Processo	Periodo T	Durata C	Priorità
A	80	40	1 ← Bassa
B	40	10	2
C	20	5	3 ← Alta

Il test di ammissibilità fallisce $U = 1 > f(3) = 0{,}78$
ma il sistema è ammissibile!



- *Politica di ordinamento a priorità fissa con prerilascio e scadenza inferiore a periodo (D < T)*
  - Assegnazione di priorità secondo la scadenza
  - Rischio di inversione di priorità
    - Processi a priorità maggiore bloccati dall'esecuzione di processi a priorità minore
    - Effetto causato dall'accesso esclusivo a risorse condivise
    - Può condurre a blocco circolare (deadlock)



Esempio IP1
– Consideriamo tre processi L, M, H in ordine di priorità crescente
– Assumiamo che condividano la risorsa R (Mutex)
– Inversione di priorità
  - L si aggiudica R
  - H diviene attivo e vuole R
  - H deve attendere che L rilasci R (il tempo d'uso di R ha durata prevedibile)
  - M diviene attivo e blocca L (diverse priorità)
  - H deve attendere che M finisca e che L esca dalla sezione critica

Cosa accade se nel frattempo si attivano altri processi a priorità intermedia tra M e H?
Soluzione: Innalzamento delle priorità

Versione base (Basic Priority Inheritance)
– La BPI non impedisce il deadlock
1. L'innalzamento avviene solo quando un processo a priorità maggiore si blocca all'ingresso di una risorsa attualmente in possesso di un processo a priorità inferiore
2. Il processo che possiede la risorsa (e che ha avuto l'innalzamento di priorità) può così terminare senza altre interruzioni e l'arrivo di un altro processo di priorità ancora superiore causa prerilascio e riporta la situazione al punto 1

Versione avanzata (Immediate ceiling priority) → Evita il deadlock
• Ogni processo j ha una priorità statica di base PB_j
• Ogni risorsa condivisa i ha una priorità (ceiling) PC_i pari alla massima priorità dei processi che possono richiedere di usarla
• Ogni processo j ha una priorità dinamica $P_j = max\{PB_j, PC_i\}\ \forall$ risorsa condivisa i in suo possesso
• Un processo può acquisire una risorsa solo se la sua priorità dinamica corrente è maggiore del ceiling di tutte le risorse attualmente in possesso di altri processi
– Un processo a priorità maggiore può essere bloccato una sola volta durante l'intera sua esecuzione (solo per la durata della sezione critica del processo a priorità più bassa)

Esempio IP2
– Consideriamo tre processi L, M, H con priorità crescente
– Assumiamo che tutti condividano le risorse R1 e R2 (entrambe Mutex)
• Il priority ceiling di R1 e R2 è superiore alla priorità di H
– L acquisisce R1 e ne assume il ceiling poi si accinge a richiedere R2
– H diventa pronto a questo istante e vorrebbe prerilasciare L
• Ma non può perché la priorità di H non è superiore al ceiling di R1 e R2
– Quindi H resta pronto ma non riesce a prerilasciare L
– L acquisisce anche R2 e poi prosegue fino a rilasciare R1 e R2 ù
• La priorità di L ritorna al valore originale
– H ha ora priorità maggiore di L e di ogni altro eventuale M
• H può acquisire R2 proseguire e completare

La tecnica IPC impedisce il formarsi di catene di blocchi
– I processi H_i subiscono al più 1 blocco da parte di 1 processo L in possesso di risorsa R condivisa con {H}. Blocco = ritardo nel primo prerilascio

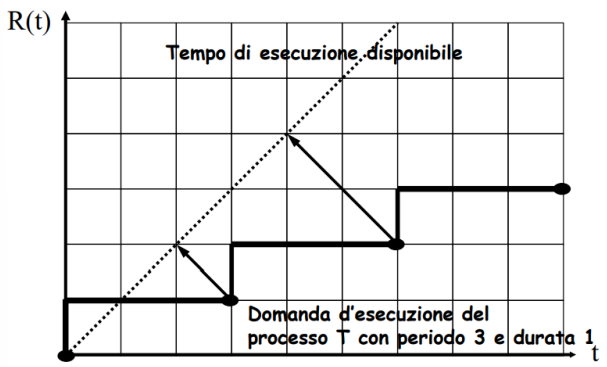
- **Per sistemi in tempo reale**
  - Calcolo del **tempo di risposta** R_i del processo i
    - **Tempo di blocco** del processo i
      - $B_i = \max_k\{C_k\}\ \forall$ risorsa k usata da processi a priorità più bassa di i
    - **Interferenza** subita dal processo i da parte di tutti i processi j a priorità maggiore
      - $I_i = \Sigma_j \lceil R_i/T_j \rceil C_j$
    - $R_i = C_i + B_i + I_i$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

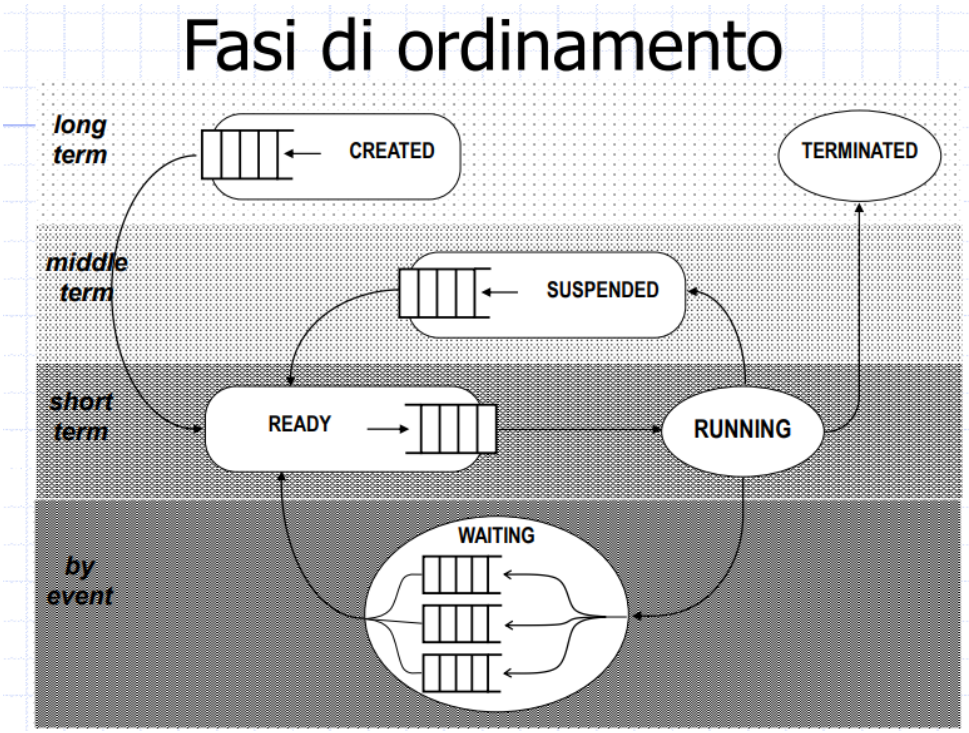
$$\omega_i^{k+1} := C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^k}{T_j} \right\rceil C_j$$

$$\omega_i^0 = C_i$$

Tempo di risposta **R**



Esercizi ordinamento



- Criteri quantitativi di valutazione prestazionale delle politiche di ordinamento
  - **Efficienza di utilizzo**
    - Tempo utile/tempo di gestione
  - ***Throughput***
    - Processi completati per unità di tempo
  - **Tempo di *turn-around***
    - Tempo di completamento
  - **Tempo di attesa**
  - **Tempo di risposta**

- Consiste nel selezionare un processo dalla ***ready list*** e attribuirgli la CPU
- L'operazione viene effettuata in modo coordinato dallo ***scheduler*** e dal ***dispatcher***
  - Moduli del nucleo del sistema operativo
    - Lo *scheduler* fissa la politica
    - Il *dispatcher* ne attua le scelte

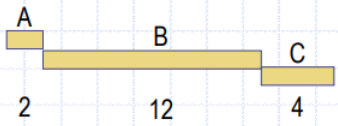
Alcune politiche di ordinamento
- ***First Come First Served*** [FCFS]
- ***Round Robin*** [RR]
- ***Shortest Job First*** [SJF]
  - Versione base senza prerilascio
  - Diventa ***Shortest Remaining Time Next*** [SRTN] se applicata con prerilascio
- Con attributo di priorità statica associata ai processi e con prerilascio [FPS]

- La CPU viene assegnata al processo che la richiede per primo
  - Selezione dei processi da una coda FIFO

Esempio
processo **A**: tempo di esecuzione = 2 [u.t.]
processo **B**: tempo di esecuzione = 12 [u.t.]
processo **C**: tempo di esecuzione = 4 [u.t.]

N.B. trascuriamo per semplicità i tempi di scambio di contesto

First Come First Served – 1.2



TEMPO DI ATTESA

T_{att} (A) = 0

T_{att} (B) = 2

T_{att} (C) = 2 + 12 = 14

T_{att} (medio) = (0 + 2 + 14) / 3 = 5,3 [u.t.]

TEMPO DI *TURN AROUND*

T_{ta} (A) = 2

T_{ta} (B) = 2 + 12 = 14

T_{ta} (C) = 2 + 12 + 4 = 18

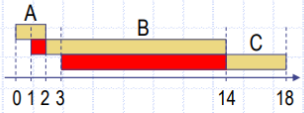
T_{ta} (medio) = (2 + 14 + 18) / 3 = 11,3 [u.t.]

… e il Tempo di Risposta ?

Tempo d'attesa: il tempo in cui un processo pronto per l'esecuzione rimane in attesa della CPU (waiting time).

Tempo di risposta: il tempo che trascorre tra la sottomissione del processo e l'ottenimento della prima risposta.

Esempio 2 processo **A**: tempo di arrivo = 0
tempo di esecuzione = 2 [u.t.]
processo **B**: tempo di arrivo = 1
tempo di esecuzione = 12 [u.t.]
processo **C**: tempo di arrivo = 3
tempo di esecuzione = 4 [u.t.]



TEMPO DI ATTESA

T_{att} (A) = 0

T_{att} (B) = 1

T_{att} (C) =11

T_{att} (medio) = (0 + 1 + 11) / 3 = 4 [u.t.]

TEMPO DI *TURN AROUND*

T_{ta} (A) = 2

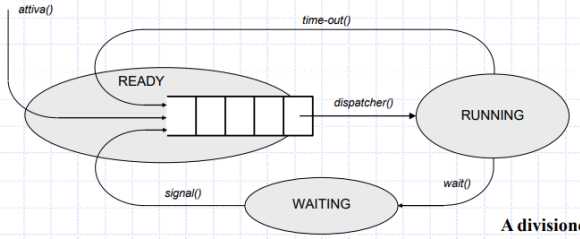
T_{ta} (B) = 1 + 12 = 13

T_{ta} (C) = 11 + 4 = 15

T_{ta} (medio) = (2 + 13 + 15) / 3 = 10 [u.t.]

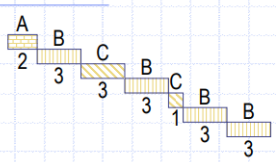
Round Robin – 1

- Opera come FCFS ma con prerilascio per esaurimento del quanto di tempo
  - La *ready list* viene trattata come una coda circolare



A divisione di tempo

Round Robin – 2



Tempo di arrivo di A = 0, di esecuzione = 2

Tempo di arrivo di B = 0, di esecuzione = 12

Tempo di arrivo di C = 0, di esecuzione = 4

Quanto di tempo = 3 [u.t.]

TEMPO DI ATTESA

T_{att} (A) = 0

T_{att} (B) = 2 + 3 + 1 = 6

T_{att} (C) = 2 + 3 + 3 = 8

T_{att} (medio) = (0 + 6 + 8) / 3 = 4,6 [u.t.]

TEMPO DI *TURN AROUND*

T_{ta} (A) = 2

T_{ta} (B) = 2 + 3 + 3 + 3 + 1 + 3 +3 = 18

T_{ta} (C) = 2 + 3 + 3 + 3 + 1 = 12

T_{ta} (medio) = (2 + 18 + 12) / 3 = 10,6 [u.t.]

Round Robin – 3

Calcolare i tempi di attesa e di *turn-around* medi con un valore di quanto prima di 1 e poi di 5 [u.t.]. Cambierà qualcosa?

Quanto di tempo = 1 [u.t.]

T_{att} (medio) = (2 + 6 + 6) / 3 = 4,6 [u.t.] T_{ta} (medio) = (4 + 18 + 10) / 3 = 10,6 [u.t.]

Quanto di tempo = 5 [u.t.]

T_{att} (medio) = (0 + 6 + 7) / 3 = 4,3 [u.t.] T_{ta} (medio) = (2 + 18 + 11) / 3 = 10,3 [u.t.]

Round Robin – 4

```
A  A
 B  B B B BBBBBBBB
  C  C C C
```

Quanto di tempo = 1 [u.t.]

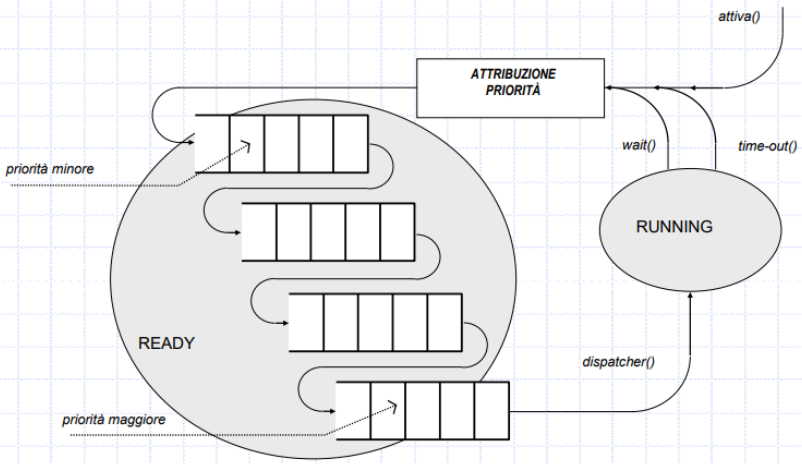
```
AaaA
bBbbBbBbBbBBBBBBBB
ccCccCcCcC
```

```
AA
  BBBBB      BBBBBBB
       CCCC
```

Quanto di tempo = 5 [u.t.]

```
AA
bbBBBBBbbbbBBBBBBB
ccccccCCCC
```

Round Robin con priorità multiple



Priorità: 5 elevata, 1 bassa

TEMPO DI ATTESA = 16,50 [u.t.]

TEMPO DI *TURN AROUND* = 22,33 [u.t.]

Processo	Arrivo	Esecuzione	Priorità
A	0	7	4
B	0	6	2
C	0	10	3
D	0	2	3
E	0	7	5
F	0	3	1

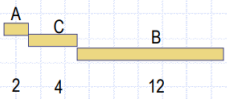
slot time = 4 [u.t.]

```
EEEE.EEE
aaaa.aaa.AAAA.AAA
cccc.ccc.cccc.ccc.CCCC.cc.CCCC.CC
dddd.ddd.dddd.ddd.dddd.DD
bbbb.bbb.bbbb.bbb.bbbb.bb.bbbb.bb.BBBB.BB
ffff.fff.ffff.fff.ffff.ff.ffff.ff.ffff.ff.FFF
```

Shortest Job First

- Meglio definita come
  - "***Shortest next-CPU-burst First***"
- La CPU viene assegnata al processo che ha il "*CPU-burst*" successivo più breve
- Può essere realizzata senza uso di prerilascio
- Oppure con prerilascio
  - SRTN

Shortest Job First senza prerilascio



TEMPO DI ATTESA

T_{att} (A) = 0

T_{att} (B) = 2 + 4 = 6

T_{att} (C) = 2

T_{att} (medio) = (0 + 6 + 2) / 3 = 2,6 [u.t.]

TEMPO DI *TURN AROUND*

T_{ta} (A) = 2

T_{ta} (B) = 2 + 4 + 12 = 18

T_{ta} (C) = 2 + 4 = 6

T_{ta} (medio) = (2 + 18 + 6) / 3 = 8,6 [u.t.]

Shortest Job First con prerilascio (SRTN)

TEMPO DI ATTESA = 7 [u.t.]

TEMPO DI *TURN AROUND* = 12,83 [u.t.]

Processo	Arrivo	Esecuzione
A	2	7
B	0	6
C	5	10
D	10	2
E	7	7
F	4	3

```
BBBBBB
----ff.FFF
--aaaa.aaa.A.aa.AAAAAA
------.---.-.DD
------.-ee.e.ee.eeeeee.EEEEEEE
-----c.ccc.c.cc.cccccc.ccccccc.CCCCCCCCCC
```

Esercizio con soluzioni – 1

Cinque processi *batch*, identificati dalle lettere A-E arrivano all'elaboratore allo stesso istante. I processi hanno un tempo di esecuzione stimato di 4, 6, 2, 3 e 6 unità di tempo rispettivamente, mentre le loro priorità (fissate esternamente) sono rispettivamente 2, 4, 5, 1 e 3 (con 5 valore maggiore). Per ognuno dei seguenti algoritmi di ordinamento determinare: **(i)** il ***tempo medio di risposta***, **(ii)** il **tempo medio di attesa e (iii)** il ***tempo medio di turn-around*** , trascurando i tempi dovuti allo scambio di contesto.

- *Round Robin* (con quanto di tempo = 2)
- Con priorità (senza prerilascio)
- FCFS
- SJF

Soluzione: Round Robin con quanto=2

Round Robin, quanto = 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Proc. A	A	A	a	a	a	a	a	a	a	a	A	A									
Proc. B	b	b	B	B	b	b	b	b	b	b	b	B	B	b	b	b	B	B			
Proc. C	c	c	c	c	C	C															
Proc. D	d	d	d	d	d	d	D	D	d	d	d	d	d	d	D						
Proc. E	e	e	e	e	e	e	e	e	E	E	e	e	e	e	e	E	E	e	e	E	E

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU	A	A	B	B	C	C	D	D	E	E	A	A	B	B	D	E	E	B	B	E	E
Coda	b	b	c	c	d	d	e	e	a	a	b	b	d	d	e	b	b	e	e		
	c	c	d	d	e	e	a	a	b	b	d	d	e	e	b						
	d	d	e	e	a	a	b	b	d	d	e	e									
	e	e	a	a	b	b															

processo	t. risposta	t. attesa	*turn-around*
A	0	8	12
B	2	13	19
C	4	4	6
D	6	12	15
E	8	15	21
medie	**20/5=4**	**52/5=10,4**	**73/5=14,6**

Soluzione: Con priorità

Con priorità

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Proc. A	a	a	a	a	a	a	a	a	a	a	a	a	a	A	A	A	A				
Proc. B	b	b	B	B	B	B	B	B													
Proc. C	C	C																			
Proc. D	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	D	D	D	
Proc. E	e	e	e	e	e	e	e	E	E	E	E	E	E								

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU	C	C	B	B	B	B	B	E	E	E	E	E	A	A	A	A	D	D	D		
Coda	b	b	e	e	e	e	e	a	a	a	a	a	d	d	d	d	d				
	e	e	a	a	a	a	a	d	d	d	d	d	d								
	a	a	d	d	d	d	d														
	d	d																			

processo	t. risposta	t. attesa	*turn-around*
A	14	14	18
B	2	2	8
C	0	0	2
D	18	18	21
E	8	8	14
medie	42/5=8,4	42/5=8,4	63/5=12,6

Soluzione: First Come First Served

FCFS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Proc. A	A	A	A	A																	
Proc. B	b	b	b	b	B	B	B	B	B	B											
Proc. C	c	c	c	c	c	c	c	c	c	c	C	C									
Proc. D	d	d	d	d	d	d	d	d	d	d	d	d	D	D	D						
Proc. E	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	E	E	E	E	E	E

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU	A	A	A	A	B	B	B	B	B	B	C	C	D	D	D	E	E	E	E	E	E
Coda	b	b	b	b	c	c	c	c	c	c	d	d	e	e	e						
	c	c	c	c	d	d	d	d	d	d	e	e									
	d	d	d	d	e	e	e	e	e	e											
	e	e	e	e																	

processo	t. risposta	t. attesa	*turn-around*
A	0	0	4
B	4	4	10
C	10	10	12
D	12	12	15
E	15	15	21
medie	**41/5=8,2**	**41/5=8,2**	**62/5=12,4**

Soluzione: Shortest Job First

SJF

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Proc. A	a	a	a	a	a	A	A	A	A												
Proc. B	b	b	b	b	b	b	b	b	B	B	B	B	B	B							
Proc. C	C	C																			
Proc. D	d	d	D	D	D																
Proc. E	e	e	e	e	e	e	e	e	e	e	e	e	e	e	E	E	E	E	E	E	

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU	C	C	D	D	D	A	A	A	A	B	B	B	B	B	B	E	E	E	E	E	E
Coda	d	d	a	a	a	b	b	b	b	e	e	e	e	e							
	a	a	b	b	b	e	e	e	e												
	b	b	e	e	e																
	e	e																			

processo	t. risposta	t. attesa	*turn-around*
A	5	5	9
B	9	9	15
C	0	0	2
D	2	2	5
E	15	15	21
medie	**31/5=6,2**	**31/5=6,2**	**52=10,4**

Esercizio con soluzioni – 2

Cinque processi *batch*, identificati dalle lettere A-E arrivano all'elaboratore allo stesso istante. I processi hanno un tempo di esecuzione stimato di 8, 10, 2, 4 e 8 unità di tempo rispettivamente, mentre le loro priorità (fissate esternamente) sono rispettivamente 2, 4, 5, 1 e 3 (con 5 valore maggiore). Per ognuno dei seguenti algoritmi di ordinamento determinare: **(i)** il ***tempo medio di turn-around*** e **(ii)** il **tempo medio di attesa**, trascurando i tempi dovuti allo scambio di contesto.

- *Round Robin* (con quanto di tempo = 2)
- Con priorità (senza prerilascio)
- FCFS
- SJF

Soluzioni Esercizio 2

RR

t_{att} (medio) = 15,6 [u.t.]

t_{ta}(medio) = 22 [u.t.]

priorità

t_{att} (medio) = 12,4 [u.t.]

t_{ta}(medio) = 18,8 [u.t.]

FCFS

t_{att} (medio) = 14 [u.t.]

t_{ta}(medio) = 20,4 [u.t.]

SJF

t_{att} (medio) = 8,8 [u.t.]

t_{ta}(medio) = 15,2 [u.t.]

Esercizio con soluzioni – 3

Cinque processi *batch*, identificati dalle lettere A-E, arrivano all'elaboratore agli istanti di tempo 0, 2, 5, 8 e 11 rispettivamente. I processi hanno un tempo di esecuzione stimato di 9, 1, 7, 3 e 5 unità di tempo rispettivamente, mentre le loro priorità (mantenute staticamente) sono rispettivamente 3, 2, 4, 5 e 1 (con 5 valore maggiore). Per ognuna delle seguenti politiche di ordinamento determinare **(i)** il **tempo medio di risposta**, **(ii)** il **tempo medio di *turn-around*** e **(iii)** il **tempo medio di attesa**, trascurando i tempi dovuti allo scambio di contesto.

- FCFS
- *Round Robin* (quanto di tempo = 3)
- *Round Robin* (quanto di tempo = 3) con priorità ma senza prerilascio
  - Nel caso di arrivo di un processo in contemporanea a un'uscita per *time_out*(), si dia precedenza al processo prerilasciato per *time_out*()
- SJF senza prerilascio
- SJF con prerilascio (SRTN)

FCFS

Processo	Arrivo	Esecuzione	Priorità
A	0	9	3
B	2	1	2
C	5	7	4
D	8	3	5
E	11	5	1

TEMPO DI RISPOSTA = 6,0 [u.t.]

TEMPO DI ATTESA = 6,0 [u.t.]

TEMPO DI *TURN AROUND* = 11,0 [u.t.]

```
AAAAAAAAA
--bbbbbbbB
-----ccccCCCCCCC
--------ddddddddDDD
----------eeeeeeeeeEEEEE
```

Round Robin quanto di tempo = 3 [u.t.]

```
AAAaAAAaaaAAA
--bB
-----ccCCCccccccCCCcccC
--------dddddDDD
----------eeeeeeeeEEEeEE
```

TEMPO DI RISPOSTA = 3,2 [u.t.]

TEMPO DI ATTESA = 6,0 [u.t.]

TEMPO DI *TURN AROUND* = 11,0 [u.t.

```
CPU  AAABAAACCCAAADDDCCCEEECEE
coda --ba-ccaaadddcccceeeccce
     --------ddcccceee
     ----------ee
     ----------
```

Attenzione, un classico errore è la tentazione di eseguire D dopo C, per alternanza, visto che non è mai stato eseguito prima, mentre A lo è stato. Ma scrivendo bene la coda è evidente che sarà prima, di nuovo, il turno di A.

Round Robin con priorità ma senza prerilascio

```
AAAAAAaaaaaaaaaaAAA
--bbbbbbbbbbbbbbbbbB
-----cCCCcccCCCC
--------dDDD
------------eeeeeeeeeEEEEE
```

TEMPO DI RISPOSTA = 5,6 [u.t.]

TEMPO DI ATTESA = 8,2 [u.t.]

TEMPO DI *TURN AROUND* = 13,2 [u.t.]

CPU

```
AAAAAACCCDDDCCCCAAABEEEEE
```

coda

```
--bbbcaadcccaaaabbbe
-----bbbaaaabbbbeee
--------bbbbeeee
------------e-------------
```

Attenzione, anche se è arrivato C che ha priorità maggiore di A, quest'ultimo può finire il suo quanto di tempo poiché il sistema non prevede prerilascio.

Round Robin con priorità e con prerilascio

Prerilascio

```
AAAAAaaaaaaaaaaAAAA
--bbbbbbbbbbbbbbbbbB
-----CCCcccCCCC
--------DDD
------------eeeeeeeeeEEEEE
```

TEMPO DI RISPOSTA = 5,2 [u.t.]

TEMPO DI ATTESA = 7,8 [u.t.]

TEMPO DI *TURN AROUND* = 12,8 [u.t.]

SJF senza prerilascio

B ha tempo di esecuzione inferiore ma non può prerilasciare A

TEMPO DI RISPOSTA = 4,8 [u.t.]

TEMPO DI ATTESA = 4,8 [u.t.]

TEMPO DI *TURN AROUND* = 9,8 [u.t.]

```
AAAAAAAAA
--bbbbbbbB
-----ccccccccccccCCCCCCC
---------ddDDD
------------eeEEEEE
```

SJF con prerilascio

TEMPO DI RISPOSTA = 3,4 [u.t.]

TEMPO DI ATTESA = 3,6 [u.t.]

TEMPO DI *TURN AROUND* = 8,6 [u.t.]

B prerilascia A

```
AAaAAAAAAA
--B
-----ccccccccccccCCCCCCC
---------ddDDD
------------eeEEEEE
```

A aveva tempo di esecuzione di 9 ma ora gli restano solo 5 unità da eseguire, quindi meno di C. Pertanto C non prerilascia A

Esercizio

Si supponga che tre clienti arrivino a una stazione di servizio per fare il pieno di benzina, e che ognuno impieghi il seguente tempo, noto a priori

auto	arrivo	servizio (in minuti)
A	8:00	8
B	8:06	5
C	8:10	2

Nell'ipotesi che alle 8:00 l'unica pompa di benzina della stazione sia libera, calcolare il tempo medio di attesa e il tempo medio di turn-around applicando politiche di ordinamento FCFS, SJF senza prerilascio e SJF con prerilascio (ovvero SRTN).

Risolto con FCFS

```
AAAAAAA
------bbBBBBB
----------cccCC
```

CPU

```
AAAAAAAABBBBBCC
```

coda

```
------bb--ccc
---------------
```

TEMPO DI RISPOSTA = (0+2+3)/3 = 5/3

TEMPO DI ATTESA = (0+2+3)/3 = 5/3

TEMPO DI *TURN AROUND* = (8+7+5)/3 = 20/3

Risolto con SJF

Senza prerilascio

```
AAAAAAAA
------bbBBBBB
----------cccCC
```

CPU: `AAAAAAAABBBBBCC`

coda:
```
------bb--ccc
--------------
```

TEMPO DI RISPOSTA = (0+2+3)/3 = 5/3

TEMPO DI ATTESA = (0+2+3)/3 = 5/3

TEMPO DI *TURN AROUND* = (8+7+5)/3 = 20/3

Con prerilascio - SRTN

```
AAAAAAAA
------bbBBbbBBB
----------CC
```

CPU: `AAAAAAAABBCCBBB`

coda:
```
------bb--bb
--------------
```

TEMPO DI RISPOSTA = (0+2+0)/3 = 2/3

TEMPO DI ATTESA = (0+4+0)/3 = 4/3

TEMPO DI *TURN AROUND* = (8+9+2)/3 = 19/3

Esercizio per casa

Cinque processi *batch*, identificati dalle lettere A-E, arrivano all'elaboratore allo stesso istante. I processi hanno un tempo di esecuzione stimato di 10, 6, 2, 4 e 8 unità di tempo rispettivamente, mentre le loro priorità (determinate esternamente) sono rispettivamente 3, 5, 2, 1 e 4 (con 5 valore maggiore). Per ognuno delle seguenti politiche di ordinamento determinare: **(i) il tempo medio di *turn-around*** e **(ii)** il **tempo medio di attesa**, trascurando i tempi dovuti allo scambio di contesto.

- *Round Robin* (quanto di tempo = 2)
- Con priorità esterna senza prerilascio
- FCFS
- SJF senza prerilascio

Soluzione quasi completa - 1/4

***RR* con quanto di tempo di ampiezza 2** (RIPORTIAMO SOLO PARTE DELLA SOLUZIONE – NON TUTTA)

Proc. A	A	A	a	a	a	a	a	a	a	a	A	A	a	a	a	a	a	a	A	A	a	a	a	a	…	
Proc. B	b	b	B	B	b	b	b	b	b	b	b	b	B	B	b	b	b	b	b	b	B	B				
Proc. C	c	c	c	c	C	C																				
Proc. D	d	d	d	d	d	d	D	D	d	d	d	d	d	d	D	D										
Proc. E	e	e	e	e	e	e	e	e	E	E	e	e	e	e	e	e	E	E	e	e	e	e	E	E	…	

CPU	A	A	B	B	C	C	D	D	E	E	A	A	B	B	D	D	E	E	A	A	B	B	E	E	…	
	b	b	c	c	d	d	e	e	a	a	b	b	d	d	e	e	a	a	b	b	e	e	a	a	…	
Coda	c	c	d	d	e	e	a	a	b	b	d	d	e	e	a	a	b	b	e	e	a	a				
	d	d	e	e	a	a	b	b	d	d	e	e	a	a	b	b										
	e	e	a	a	b	b																				

processo	t. risposta	t. attesa	*turn-around*
A	…	…	…
B	2	16	22
C	4	4	6
D	6	12	16
E	…	…	…
medie	**…/5**	**…/5**	**…/5**

Soluzione quasi completa - 2/4

Con priorità esterna, senza prerilascio 3, 5, 2, 1, 4 (RIPORTIAMO SOLO PARTE DELLA SOLUZIONE – NON TUTT

Proc. A	a	a	a	a	a	a	a	a	a	a	a	a	a	A	A	A	A	A	A	A	A	A	A			
Proc. B	B	B	B	B	B	B																				
Proc. C	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	…		
Proc. D	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	…		
Proc. E	e	e	e	e	e	e	E	E	E	E	E	E	E	E												

CPU	B	B	B	B	B	B	E	E	E	E	E	E	E	A	A	A	A	A	A	A	A	A	A			
	e	e	e	e	e	e	a	a	a	a	a	a	a	c	c	c	c	c	c	c	c	c	c	…		
Coda	a	a	a	a	a	a	c	c	c	c	c	c	c	d	d	d	d	d	d	d	d	d	d	…		
	c	c	c	c	c	c	d	d	d	d	d	d	d													
	d	d	d	d	d	d																				

processo	t. risposta	t. attesa	*turn-around*
A	14	14	24
B	0	0	6
C	…	…	…
D	…	…	…
E	6	6	14
Medie	**…/5**	**…/5**	**…/5**

Soluzione quasi completa - 3/4

FCFS (RIPORTIAMO SOLO PARTE DELLA SOLUZIONE – NON TUTTA)

Proc. A	A	A	A	A	A	A	A	A	A	A																
Proc. B	b	b	b	b	b	b	b	b	b	b	B	B	B	B	B	B										
Proc. C	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	C	C								
Proc. D	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	D	D	D	D	D			
Proc. E	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	E	E	…

CPU	A	A	A	A	A	A	A	A	A	A	B	B	B	B	B	B	C	C	D	D	D	D	E	E	…	
	b	b	b	b	b	b	b	b	b	b	c	c	c	c	c	c	d	d	e	e	e	e	e			
Coda	c	c	c	c	c	c	c	c	c	c	d	d	d	d	d	d	e	e								
	d	d	d	d	d	d	d	d	d	d	e	e	e	e	e	e										
	e	e	e	e	e	e	e	e	e	e																

processo	t. risposta	t. attesa	*turn-around*
A	0	0	10
B	10	10	16
C	16	16	18
D	18	18	22
E	22	22	…
Medie	**66/5**	**66/5**	**…/5**

Soluzione quasi completa - 4/4

SJF (RIPORTIAMO SOLO PARTE DELLA SOLUZIONE – NON TUTTA)

Proc. A	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	A	A	A	A	…
Proc. B	b	b	b	b	b	b	B	B	B	B	B	B												
Proc. C	C	C																						
Proc. D	d	d	D	D	D	D																		
Proc. E	e	e	e	e	e	e	e	e	e	e	e	e	E	E	E	E	E	E	E	E				

CPU	C	C	D	D	D	D	B	B	B	B	B	B	E	E	E	E	E	E	E	E	A	A	A	A	…
Coda	d	d	b	b	b	b	e	e	e	e	e	e	a	a	a	a	a	a	a	a					
	b	b	e	e	e	e	a	a	a	a	a	a													
	e	e	a	a	a	a																			
	a	a																							

processo	t. risposta	t. attesa	*turn-around*
A	20	20	…
B	6	6	12
C	0	0	2
D	2	2	6
E	12	12	20
Medie	**40/5**	**40/5**	**…/5**

Attenzione: questa non è una politica a priorità classica.
Si tratta di una versione speciale (potremmo chiamarla *fair priority*, ovviamente con prerilascio)

Ogni **X** unità-tempo consecutive in esecuzione:
Priorità := - floor(**X**),
Priorità ≥ **0**,

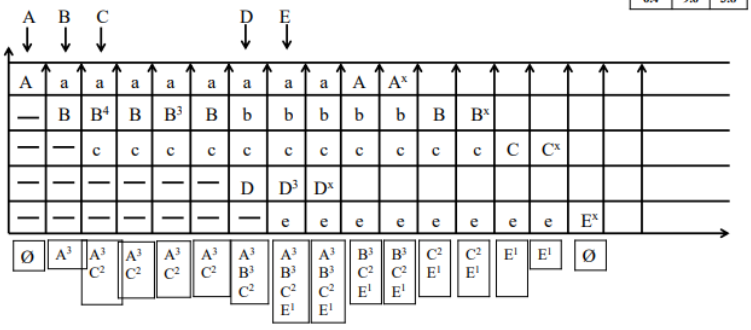
Una volta abbassata la priorità, essa non viene più rialzata.
La priorità non può scendere sotto lo 0

"*Nel caso di arrivo di un processo in contemporanea a un'uscita per time_out(), si dia precedenza al processo prerilasciato per time_out()*"
(Applicata per eventi uscita/arrivo appartenenti alla stessa classe priorità)

Pid	T_arr	T_exec	Priority
A	0	3	3
B	1	7	5
C	2	2	2
D	6	3	4
E	7	1	1

Fair Priority con X = 2
floor(X) = 1

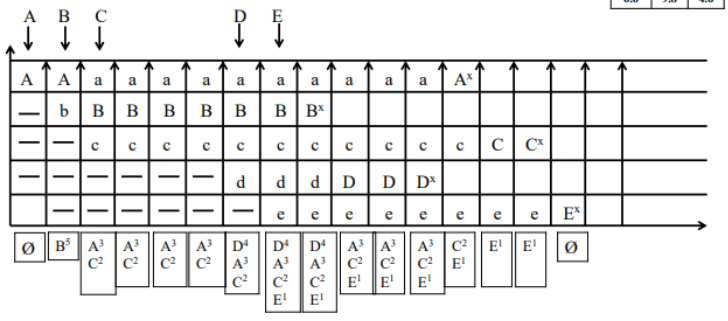
T_{att}	T_{ta}	T_r
8	11	0
5	12	0
11	13	11
0	3	0
8	9	8
6.4	**9.6**	**3.8**



Pid	T_arr	T_exec	Priority
A	0	3	3
B	1	7	5
C	2	2	2
D	6	3	4
E	7	1	1

RR con priorità senza prerilascio,
Quanto = **2** [u.t]

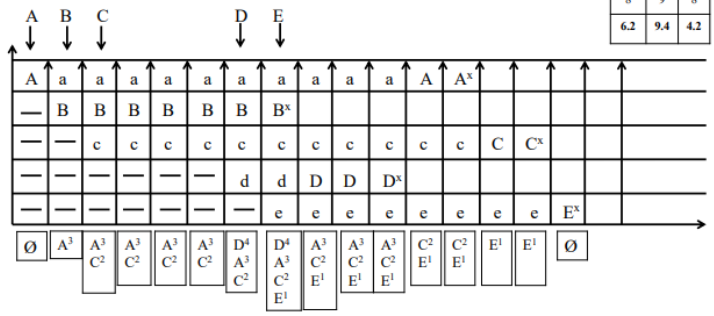
T_{att}	T_{ta}	T_r
10	13	0
1	8	1
11	13	11
3	6	3
8	9	8
6.6	**9.8**	**4.6**



Pid	T_arr	T_exec	Priority
A	0	3	3
B	1	7	5
C	2	2	2
D	6	3	4
E	7	1	1

RR con priorità con prerilascio,
Quanto = **2** [u.t]

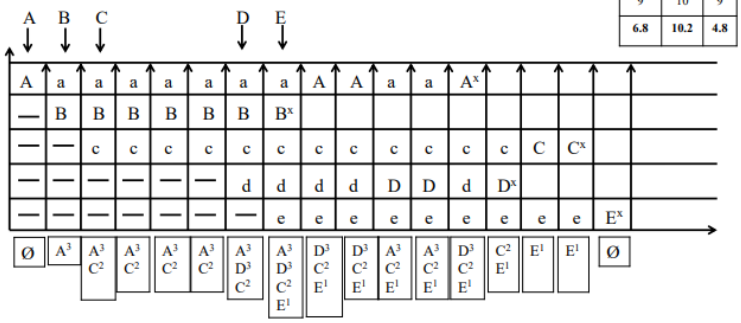
T_{att}	T_{ta}	T_r
10	13	0
0	7	0
11	13	11
2	5	2
8	9	8
6.2	**9.4**	**4.2**



Pid	T_arr	T_exec	Priority
A	0	4	3
B	1	7	5
C	2	2	2
D	6	3	3
E	7	1	1

RR con priorità con prerilascio,
Quanto = **2** [u.t]

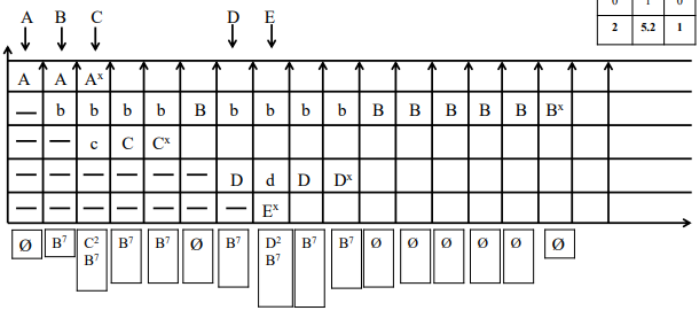
T_{att}	T_{ta}	T_r
9	13	0
0	7	0
11	13	11
5	8	4
9	10	9
6.8	**10.2**	**4.8**



Pid	T_arr	T_exec	Priority
A	0	3	3
B	1	7	5
C	2	2	2
D	6	3	4
E	7	1	1

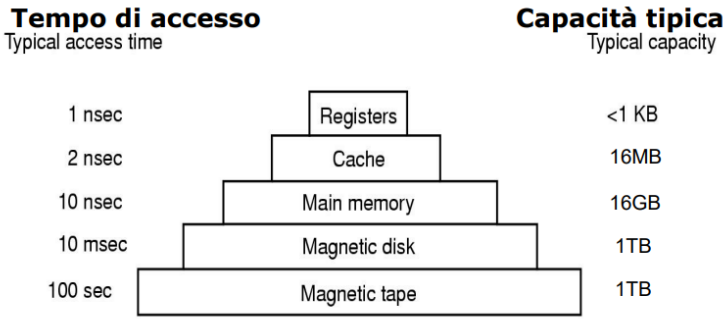
SJF con prerilascio (ovvero **SRTN**)

T_{att}	T_{ta}	T_r
0	3	0
8	15	4
1	3	1
1	4	0
0	1	0
2	**5.2**	**1**

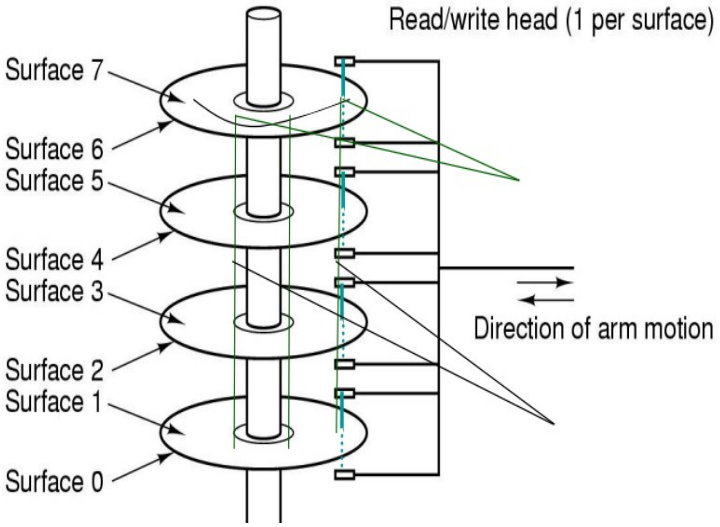


Ricapitolazione concerti di base

La gerarchia di memoria è così organizzata:



I registri sono interni alla CPU; dimensione:
– 32 bit su processori a 32-bit
– 64 bit su processori a 64-bit
• La cache è controllata dall'hardware ed è suddivisa in blocchi chiamati line con ampiezza tipica 64 B
– L1 dentro la CPU, L2 adiacente alla CPU
• L2 condivisa (Intel) o propria di ciascun core (AMD)?
– Hit (2 cicli di clock), miss (memoria)
– Write through, copy back
• I dischi magnetici hanno capienza 100 volte superiore e costo/bit 100 volte inferiore rispetto alla RAM
– Ma per tempo di accesso 1000 volte peggiore



HDD è la parte più lenta del PC
• SSD formati da insieme di chip e non da disco e testina
• Simile a memorie flash come le chiavette USB (ma più veloci)
• Non ha parti fisiche in movimento
• Più solidi di un HDD (no movimento…)
• Più veloci di un HHD (fino anche a 200 volte)
• Non c'è frammentazione (no movimento…)
• Silenziosi (no movimento…)
• Consumo elettricità/batteria inferiore (2W vs 6W come picco)
• Più leggeri di un HDD
• Tipicamente meno capienti di un HDD
• Più costosi
• Continue scritture e riscritture usurano le celle

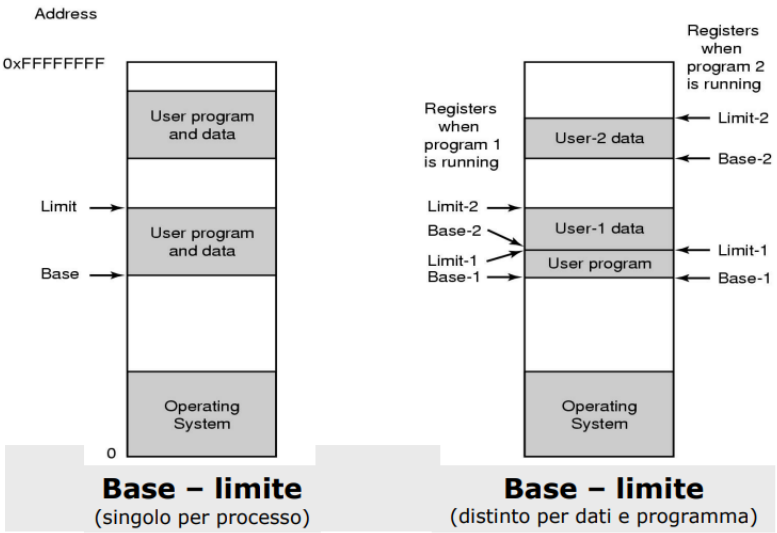
SSD veloce e poco capiente per software base
- Sistema Operativo e altre applicazioni di uso molto frequente

• HDD molto capiente ed economico per applicazioni di uso infrequente e file
- es. foto, musica, film, ma non solo

• In questo modo si ottiene una macchina molto veloce all'avvio e nella maggior parte dei casi mentre si mantiene comunque un'ampia capacità di immagazzinamento dati

CMOS è una memoria volatile ma alimentata da una piccola batteria. il CMOS può essere inteso anche come una tecnologia utilizzata per la produzione di circuiti integrati. Pur trovando ampio utilizzo nella produzione di microprocessori, microcontroller, memoria RAM statica (SRAM, acronimo di static RAM) e altre tipologie di processori logici digitali, la sigla CMOS ha finito con l'essere legata a doppio filo con un particolare componente delle motherboard.
– Memorizza ora e da quale disco fare boot
– Memorizza impostazioni BIOS diverse da default

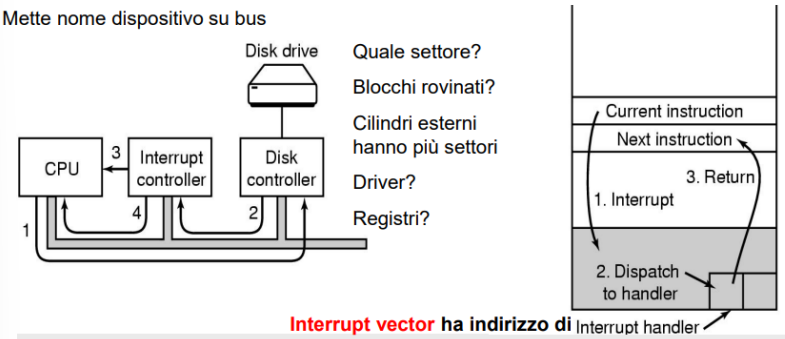
Interazione memoria-programmi:
– Come proteggere i programmi tra loro e il kernel dai programmi
– Come gestire la rilocazione
– Quando si compila un programma non si sa in che area della memoria verrà caricato
- Soluzione hardware, due registri (base e limite)
- Verifica e somma base+indirizzo costa qualche ciclo di CPU



Nella sua forma più rudimentale la ripartizione della RAM
tra processi distinti utilizza 2 registri speciali
– Base e limite, i cui valori formano parte importante del contesto del processo
– L'allocazione del processo in RAM richiede rilocazione della sua memoria virtuale
• In generale la gestione dello spazio di memoria virtuale dei processi utilizza un dispositivo di MMU (Memory Management Unit) logicamente interposto tra CPU e memoria
– Sotto la responsabilità del Sistema Operativo
• Attenzione: a ogni context switch la cache è piena di dati del processo precedente

Trattamento delle interruzioni – 1

1. Driver dice a controller cosa fare
2. Segnale "finito" su certe linee bus
3. Imposta pin in CPU
4. Mette nome dispositivo su bus



Attivazione di un dispositivo di I/O Gestione delle interruzioni

L'uso delle interruzioni per l'interazione con i dispositivi evita il ricorso al polling
• L'interazione tipica avviene in 4 passi successivi come illustrato in figura
1. Il gestore del dispositivo programma il controllore di dispositivo scrivendo nei suoi registri di interfaccia
2. Il controllore agisce sul dispositivo e poi informa il controllore delle interruzioni
3. Il controllore delle interruzioni asserisce un valore (pin) di notifica verso la CPU
4. Quando la CPU si dispone a ricevere la notifica il controllore delle interruzioni comunica anche l'identità del dispositivo
• Così che il trattamento dell'interruzione sia attribuito al gestore appropriato

All'arrivo di una interruzione
– I registri PC (Program Counter) e PSW (Program Status Word) sono posti sullo stack del processo corrente
– La CPU passa al "modo operativo protetto"
– Il parametro principale che denota l'interruzione serve come indice nel *vettore delle interruzioni*
• Così si individua il gestore designato a servire l'interruzione
– La parte *immediata* del gestore esegue nel contesto del processo interrotto
• La parte del servizio meno urgente può essere invece *differita* e demandata a un processo dedicato
Prima ogni scheda I/O aveva un livello di interrupt fisso e un indirizzo fisso per i registri.
– Se si acquistavano due dispositivi con lo stesso valore di interrupt?
• Con Plug & Play, il sistema assegna centralmente i livelli di interrupt e gli indirizzi di I/O e poi li rivela alle schede

BIOS (Basic Input Output System)
– Contiene sw a basso livello per la gestione di I/O
– Viene caricato all'avvio del computer
– Verifica quanta RAM e quali dispositivi base (tastiera,ecc) sono presenti
– Fa scan dei bus ISA e PCI per rilevare dispositivi ad essi connessi
• I dispositivi vecchi (prima di plug & play, detti legacy) sono rilevati e registrati
• Vengono registrati anche i dispositivi plug & play
• Se ci sono nuovi dispositivi dall'ultimo avvio, questi vengono configurati (assegnati livelli di interrupt e indirizzi I/O)
– Determina il dispositivo di boot dalla lista in memoria CMOS

Il primo settore del dispositivo di boot viene letto in memoria ed eseguito
– Contiene un programma che esamina la tabella di partizione e determina quale partizione sia attiva
– Da tale tabella, viene caricato un secondo boot loader
• Legge il sistema operativo dalla partizione attiva e lo esegue
– Il sistema operativo interroga il BIOS per ottenere informazioni sulla configurazione del sistema
• Per ogni dispositivo controlla l'esistenza del driver
– Se non c'è chiede di inserire CD o Floppy
– Se ci sono li carica nel kernel
– Poi, esegue varie inizializzazioni ed esegue programma iniziale (login, GUI)

La maggior parte dei servizi del Sistema Operativo sono eseguiti in risposta a invocazioni esplicite di processi (chiamata di sistema). Le chiamate di sistema sono nascoste in procedure di libreria predefinite

– L'applicazione non effettua direttamente chiamate di sistema

– La procedura di libreria svolge il lavoro di preparazione necessario ad assicurare la corretta invocazione della chiamata di sistema

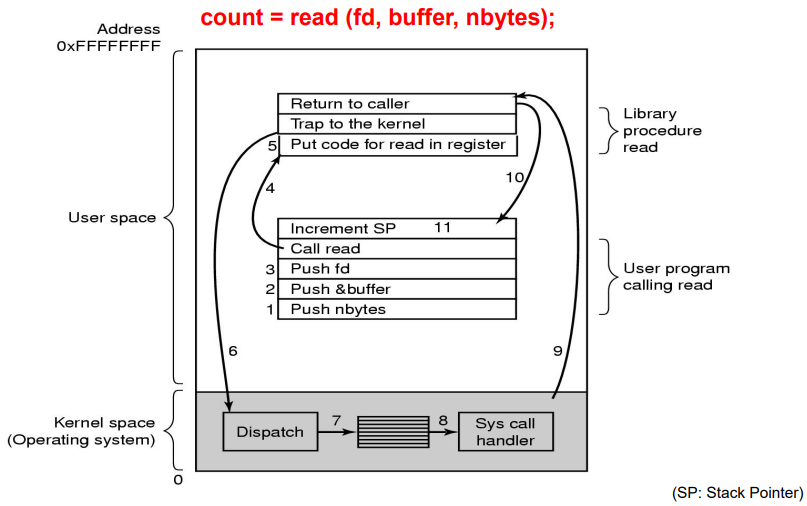
La prima istruzione di una chiamata di sistema (trap) deve attivare il modo operativo privilegiato

– Inizia esecuzione ad un indirizzo prefissato del kernel

– Il parametro della chiamata designa l'azione da svolgere e la convenzione per trovare gli altri eventuali parametri

– Il meccanismo complessivo è simile a quello già visto per il trattamento delle interruzioni

- Le interruzioni sono asincrone
- Le chiamate di sistema invece sono sincrone



(SP: Stack Pointer)

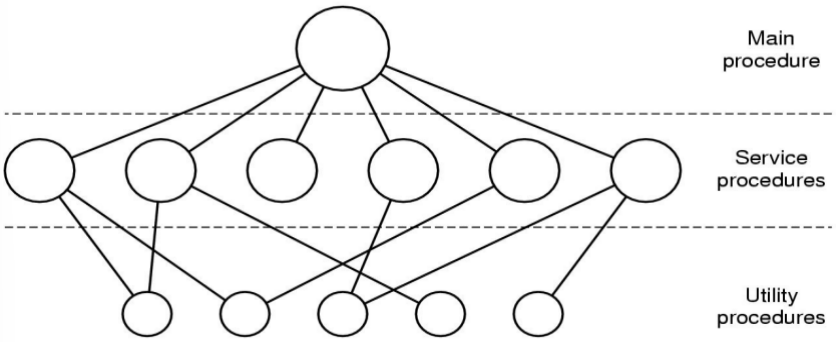
1. Il programma applicativo effettua una chiamata di sistema
- (2.-3.) Prima pone sullo stack i parametri secondo una antica convenzione C/UNIX
4. Poi invoca la procedura di libreria corrispondente alla chiamata
5. Questa pone l'ID della chiamata in un luogo noto al S/O
6. Poi esegue l'istruzione trap per passare all'esecuzione in modo operativo privilegiato
7. Il S/O individua la chiamata da eseguire
8. La esegue
9. Poi ritorna al chiamante oppure a un nuovo processo
10. Ritorna come farebbe da return di procedura
11. Cancella dati nello stack facendo avanzare il puntatore

System Calls (1)

- Una shell base tramite fork (UNIX):

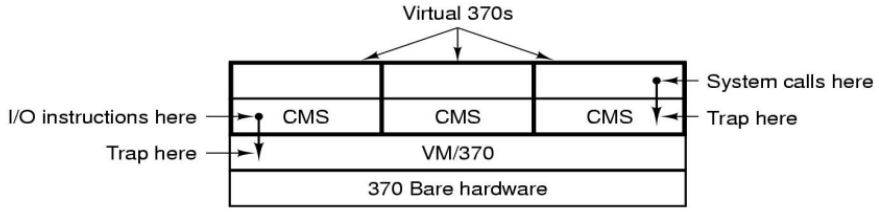
```
while (TRUE) {                          /* repeat forever
*/
   type_prompt( );                      /* display prompt */
   read_command (command, parameters)   /* input from terminal */

if (fork() != 0) {                      /* fork off child process */
   /* Parent code */
   waitpid( -1, &status, 0);            /* wait for child to exit */
} else {
   /* Child code */
   execve (command, parameters, 0);     /* execute command */
 }
}
```

Struttura monolitica



Un'architettura monolitica non ha struttura
– Il S/O è una collezione "piatta" di procedure
• Ognuna delle quali può chiamarne qualunque altra
• Nessuna forma di information hiding
– Il S/O è un singolo .o che collega tutte le procedure che lo compongono
• L'unica struttura riconoscibile in essa è data dalla
convenzione di attivazione delle chiamate di sistema
– Parametri messi in un posto preciso (stack) e poi esegue trap

Organizzazione di base:
1. Programma principale che invoca le procedure di servizio richieste;
2. Insieme di procedure che eseguono le system call;
3. Insieme di procedure di utilità che sono di ausilio per le procedure di servizio
Generalizzazione è la struttura a strati (layers), creata da Dijkstra.

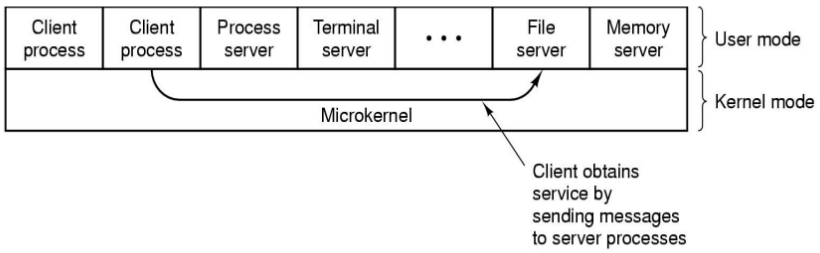


Sistema a macchina virtuale

Un primo esempio di questo sistema si ebbe negli anni' 70 con VM/370 sviluppato da IBM Scientific Center, Cambridge, Massachusetts per fornire time sharing su sistemi batch dell'azienda
– Basato sull'intuizione che un S/O a divisione di tempo in realtà realizza 2 fondamentali funzioni
1. Multiprogrammazione
2. Virtualizzazione dell'elaboratore fisico
– Separandole e ponendo 2. alla base si possono offrire copie identiche di "macchine virtuali" (copie dell'hardware) a S/O distinti che realizzano la multiprogrammazione secondo un criterio loro proprio

Da qui si ebbe l'idea per un CMS (Conversational Monitor System), quindi un S/O interattivo a divisione di tempo mono-utente. Esegue sopra una macchina virtuale realizzata da VM/370. Da qui l'idea della virtualizzazione di elaboratori logici o fisici ha avuto notevole seguito

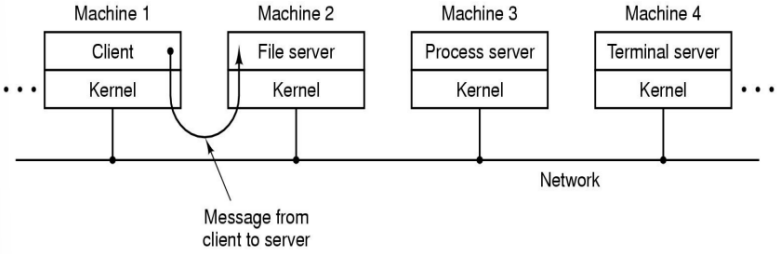
Struttura di tipo cliente-servente



L'architettura client-server, alternativamente chiamata modello client-server, è un'applicazione di rete che suddivide le attività e i carichi di lavoro tra client e server che risiedono sullo stesso sistema o sono collegati da una rete di computer. Essa è tipicamente caratterizzata da postazioni di lavoro, PC o altri dispositivi di più utenti, collegati a un server centrale tramite una connessione Internet o un'altra rete. Il client invia una richiesta di dati e il server accetta e soddisfa la richiesta, inviando i pacchetti di dati all'utente che ne ha bisogno. Questo modello è chiamato anche rete client-server o modello di network computing.
Per riassumere brevemente:

- In primo luogo, il cliente invia la sua richiesta tramite un dispositivo abilitato alla rete.
- Il server di rete accetta ed elabora la richiesta dell'utente.
- Infine, il server invia la risposta al cliente.

Struttura distribuita



Un sistema operativo distribuito (DOS) è un tipo essenziale di sistema operativo. I sistemi distribuiti utilizzano molti processori centrali per servire più applicazioni e utenti in tempo reale. Di conseguenza, i lavori di elaborazione dei dati sono distribuiti tra i processori.
Il sistema collega più computer attraverso un unico canale di comunicazione. Inoltre, ognuno di questi sistemi dispone di un proprio processore e di una propria memoria. Inoltre, queste CPU comunicano tramite bus ad alta velocità o linee telefoniche. I singoli sistemi che comunicano attraverso un unico canale sono considerati come un'unica entità. Sono anche noti come sistemi ad accoppiamento libero.

L'architettura di S/O a modello cliente-servente è anche detta a micro-kernel
• L'idea portante è di limitare al solo essenziale le responsabilità del nucleo delegando le altre a processi di sistema nello spazio di utente
– I processi di sistema sono visti come serventi
– I processi utenti sono visti come clienti
• Il ruolo del nucleo di S/O è di gestire i processi e supportare le loro comunicazioni
• Se un servizio va in crash difficilmente lo farà tutto il sistema
• Idea "pulita" ma prestazioni generalmente scadenti

Il modello di architettura microkernel (a volte indicato come modello di architettura plug-in) è un modello naturale per l'implementazione di applicazioni basate su prodotti. Un'applicazione basata sul prodotto è quella che viene confezionata e resa disponibile per il download in versioni come un tipico prodotto di terze parti. Tuttavia, molte aziende sviluppano e rilasciano le loro applicazioni aziendali interne come prodotti software, con tanto di versioni, note di rilascio e funzionalità collegabili. Anche queste applicazioni si adattano naturalmente a questo pattern. Il pattern dell'architettura microkernel consente di aggiungere ulteriori funzionalità dell'applicazione come plug-in all'applicazione principale, garantendo l'estensibilità e la separazione e l'isolamento delle funzionalità.

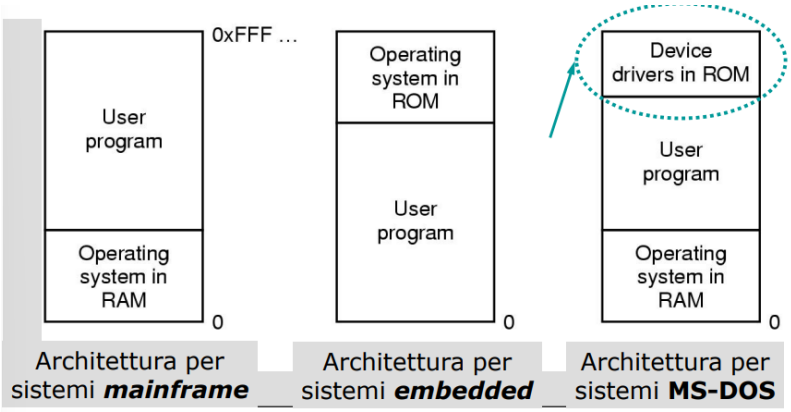
Features	Microkernel	Monolithic Kernel
Definizione	È un tipo di kernel che implementa un sistema operativo fornendo una gestione dello spazio degli indirizzi a basso livello, IPC e gestione dei thread.	È un tipo di kernel in cui l'intero sistema operativo viene eseguito alla velocità del kernel.
Dimensione	È di dimensioni ridotte.	È più grande del microkernel.
Velocità	L'esecuzione dei processi è più lenta.	L'esecuzione dei processi è più veloce.
Basic	Implementa i servizi del kernel e dell'utente in spazi di indirizzi diversi.	Implementa i servizi utente e kernel nello stesso spazio di indirizzi.
Sicurezza	È più sicuro del kernel monolitico.	È meno sicuro del microkernel.
Stabilità	Il fallimento di un singolo processo non influisce sugli altri processi.	In un kernel monolitico, se un servizio si guasta, si guasta l'intero sistema.
Estensibilità	È facile da estendere.	È difficile da estendere.
Codice	È necessario più codice per scrivere un microkernel.	È necessario meno codice per scrivere un kernel monolitico.
Elaborazione	Comunicazione I microkernel utilizzano le code di messaggistica per ottenere IPC.	I kernel monolitici utilizzano segnali e socket per ottenere l'IPC.
Maintainability	La manutenzione richiede tempo e risorse supplementari.	È facilmente manutenibile

Debug	È facile da debuggare.	È difficile da debuggare.
Esempio	Symbian, L4Linux, K42, Mac OS X, PikeOS, HURD, ecc.	Linux, BSD, Solaris, OS-9, DOS, OpenVMS, ecc.

Gestione della memoria

Nell'ottica degli utenti applicativi la memoria deve essere
– Capiente
– Veloce
– Permanente (non volatile)
Solo l'intera gerarchia di memoria nel suo insieme possiede tutte queste caratteristiche. Il gestore della memoria è la componente di S/O incaricata di soddisfare le esigenze di memoria dei processi.
Esistono due classi fondamentali di sistemi di gestione della memoria
1. Per processi allocati in modo fisso
2. Orientate a processi soggetti a migrazione da memoria principale a disco durante l'esecuzione
La memoria disponibile è in generale inferiore a quella necessaria per tutti i processi attivi simultaneamente.

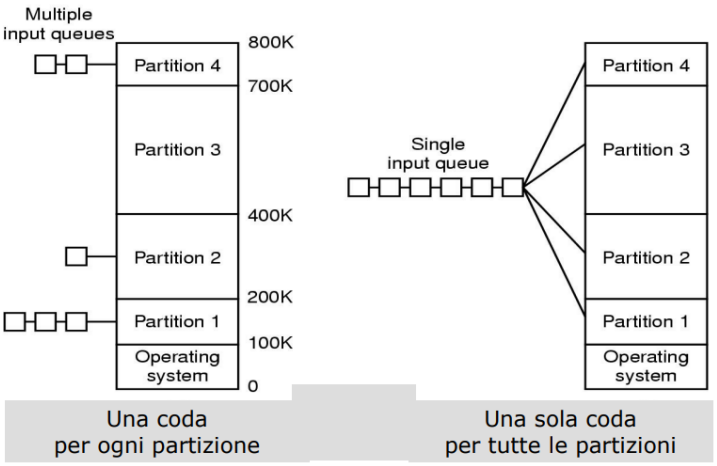
Prendiamo i *sistemi monoprogrammati*:
• Esegue un solo processo alla volta
• La memoria disponibile è ripartita solo tra quel processo e il S/O
• L'unica scelta progettuale rilevante in questo caso è decidere dove allocare la memoria (dati e programmi) del S/O
• La parte di S/O ospitata in RAM è però solo quella che contiene l'ultimo comando invocato dall'utente

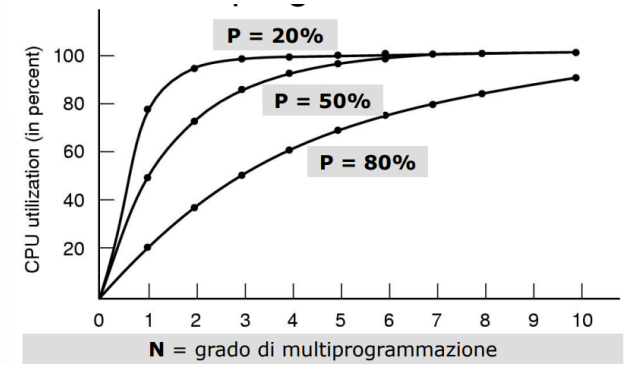


La *frammentazione* indica la suddivisione di dati in più parti. Ne abbiamo di due tipi:
• Frammentazione interna
– Con la memoria divisa in blocchi di dimensione uguale, alcuni blocchi rimangono riempiti solo in parte
• Frammentazione esterna
– Con la memoria divisa in blocchi di dimensione variabile, rimangono delle aree di memoria sparse e libere non copribili da blocchi
• Conseguenza: spreco di memoria

La forma più rudimentale di gestione della memoria per questi sistemi crea una partizione per ogni processo staticamente all'avvio del sistema. Le partizioni possono avere dimensione diversa.
Il problema diventa assegnare dinamicamente processi a partizioni, cercando di minimizzare la frammentazione interna.
A ogni nuovo processo (o lavoro) viene assegnata la partizione di dimensione più appropriata
– Una coda di processi per partizione
– Scarsa efficacia nell'uso della memoria disponibile

In generale, si ha un'assegnazione opportunistica, con una sola coda per tutte le partizioni.
• Quando si libera una partizione, viene assegnata al processo a essa più adatto e più avanti nella coda
• Oppure assegnata al "miglior" processo scandendo l'intera coda
– I processi più "piccoli" sono discriminati quando invece meriterebbero di essere privilegiati in quanto più interattivi





Valutazione dei vantaggi della multiprogrammazione
Valutazione probabilistica di quanti processi debbano eseguire in parallelo per massimizzare l'utilizzazione della CPU sotto l'ipotesi che :
• Ogni processo impegni il P% del suo tempo in attività di I/O
• N processi simultaneamente in memoria
– L'utilizzo stimato della CPU allora è $1 – P^N$

Esempio Progettazione Memoria

Si consideri un computer con 32 MB di memoria e 80% di attesa I/O media per ogni processo
– 16 MB riservati per il sistema operativo
– 4 MB riservati per ciascun processo
– In totale si hanno quindi 4 processi simultaneamente in memoria
– Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^4 =$ **60%**

Aggiungendo altri 16 MB
– Si possono avere 8 programmi simultaneamente in memoria
– Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^8 =$ **83%**

Aggiungendo altri 16 MB
– Si possono avere 12 programmi simultaneamente in memoria
– Con P = 0,8 si ha una utilizzazione della CPU di $1 - 0{,}8^{12} =$ **93%**

Rilocazione
– Interpretazione degli indirizzi emessi da un processo in relazione alla sua collocazione corrente in memoria. Occorre distinguere tra riferimenti assoluti permissibili al programma e riferimenti relativi da rilocare

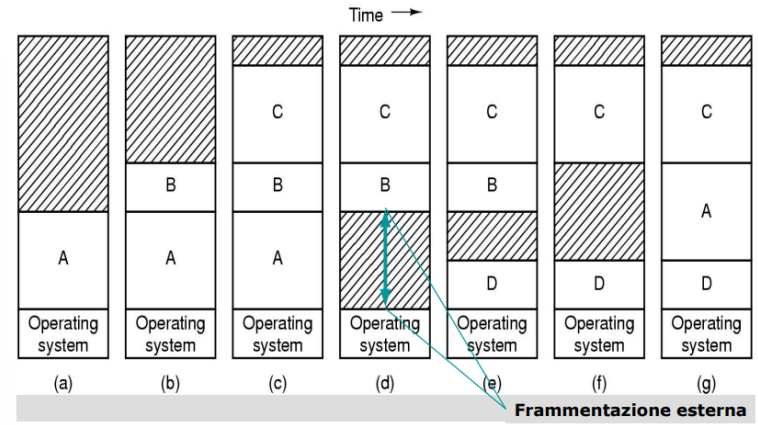
Protezione
– Assicurazione che ogni processo operi soltanto nello spazio di memoria a esso permissibile. Soluzione storica adottata da IBM
– Memoria divisa in blocchi (2 KB) con codice di protezione per blocco (4 bit)
– La PSW di ogni processo indica il suo codice di protezione
– Il S/O blocca ogni tentativo di accedere a blocchi con codice di protezione diverso da quello della PSW corrente

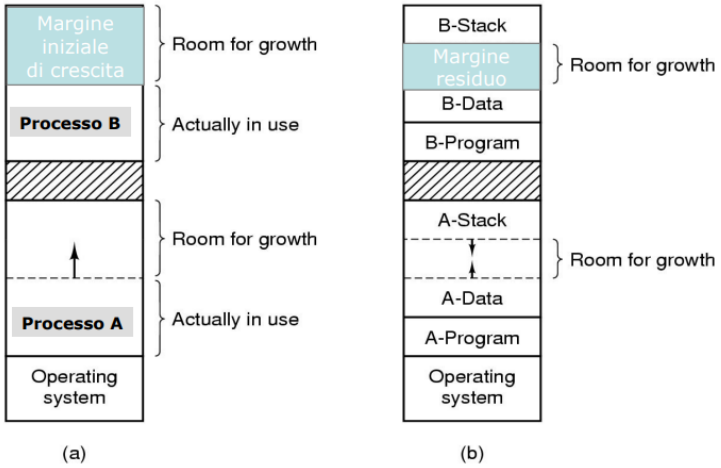
Soluzione combinata (rilocazione + protezione)
– Un processo può accedere memoria solo tra la base e il limite della partizione a esso assegnata

– Valore base aggiunto al valore di ogni indirizzo riferito (operazione costosa)
– Il risultato confrontato con il valore limite (operazione veloce

La tecnica più rudimentale per alternare processi in memoria principale senza garantire allocazione fissa è lo *swapping*. Trasferisce processi interi e assegna partizioni diverse nel tempo ed il processo rimosso viene salvato su memoria secondaria; ovviamente solo le parti modificate.

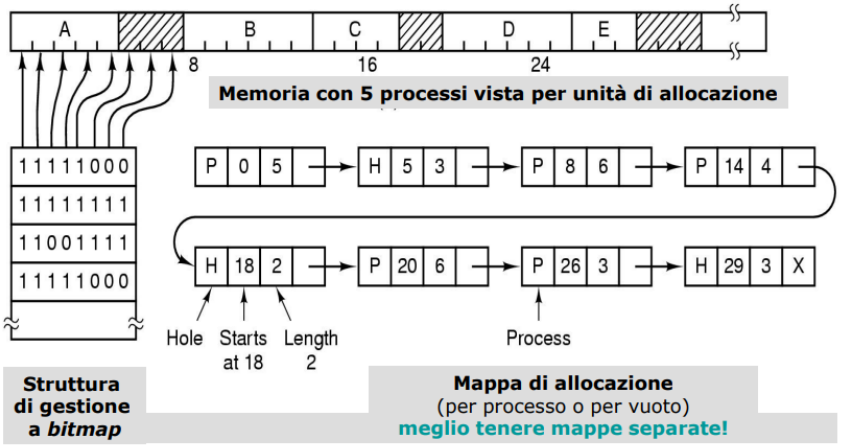
Processi diversi richiedono partizioni di ampiezze diverse assegnate ad hoc
– Rischio di frammentazione esterna
– Occorre ricompattare periodicamente la memoria principale, pagando un costo temporale importante
(spostando 4 B in 40 ns., servono 5.37 s. per una RAM ampia 512 MB).
Le dimensioni di memoria di un processo possono variare nel tempo. Difficile ampliare dinamicamente l'ampiezza della partizione assegnata. Meglio assegnare con margine.





Con memoria principale allocata dinamicamente è essenziale tenere traccia del suo stato d'uso. Due strategie principali
1) Mappe di bit
• Memoria vista come insieme di unità di allocazione (1 bit per unità)
– Unità piccole → struttura di gestione grande
» Esempio: Unità da 32 bit e RAM ampia 512 MB → struttura ampia 128 M bit = 16 MB → 3.1 % (= 1/32)

La strategia alternativa usa *liste collegate*.
Nella sua versione più semplice la memoria è vista a segmenti (segmento = processo | spazio libero tra processi)
Ogni elemento di lista rappresenta un segmento
– Ne specifica punto di inizio, ampiezza e successore
– Liste ordinate per indirizzo di base

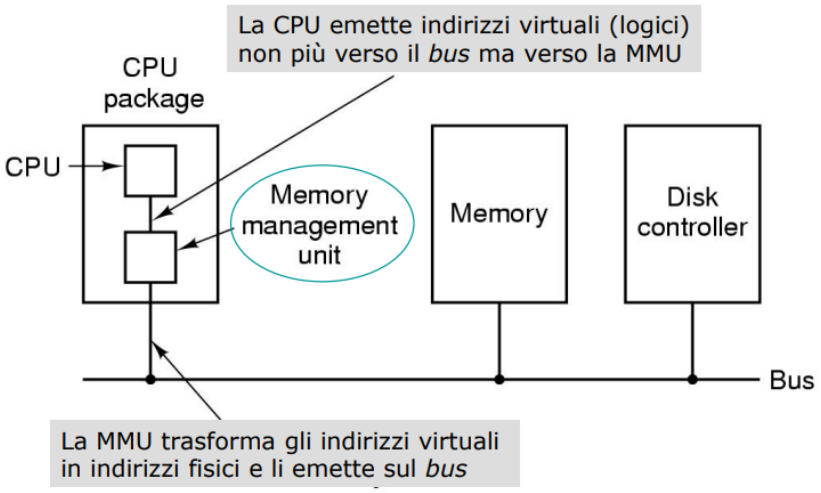


Varie strategie di allocazione
• First fit : il primo segmento libero ampio abbastanza
• Next fit : come First fit ma cercando sempre avanti
• Best fit : il segmento libero più adatto
• Worst fit : sempre il segmento libero più ampio
• Quick fit : liste diverse di ricerca per ampiezze "tipiche"

Scritto da Gabriel

Una singola partizione o anche l'intera RAM sono presto divenute insufficienti per ospitare un intero processo. La prima soluzione fu di suddividere il processo in parti chiamate *overlay*.
– Veniva caricata in RAM una parte alla volta
– Non appena "consumata" le veniva sovrapposta la parte successiva
– Suddivisione a cura del programmatore

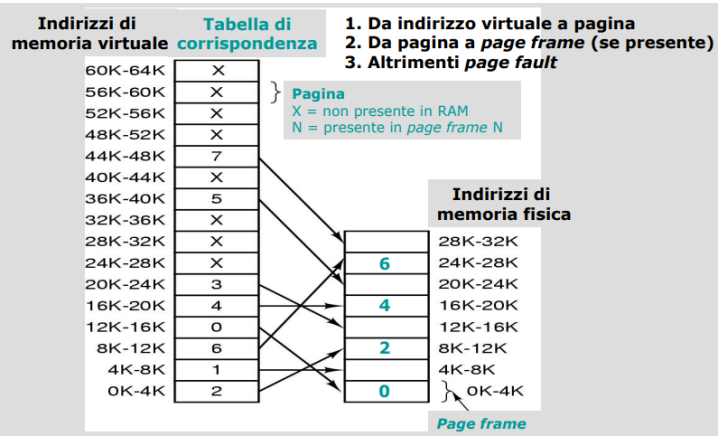
L'idea di memoria virtuale nasce nel '61. Il principio cardine è che un singolo processo può liberamente avere ampiezza maggiore della RAM disponibile. Basta caricarne in RAM solo la parte strettamente necessaria lasciando il resto su disco
- Senza intervento del programmatore
- Ogni processo ha un suo proprio spazio di memoria virtuale
Gli indirizzi generati dal processo non denotano più direttamente una locazione in RAM
– Ma vengono interpretati da un'unità detta MMU che li mappa verso indirizzi fisici reali, prima di essere emessi sul bus
– Il tipo di interpretazione a carico della MMU dipende dalla tecnica usata per la gestione della memoria virtuale



Per realizzare la memoria virtuale ci sono due tecniche principali:

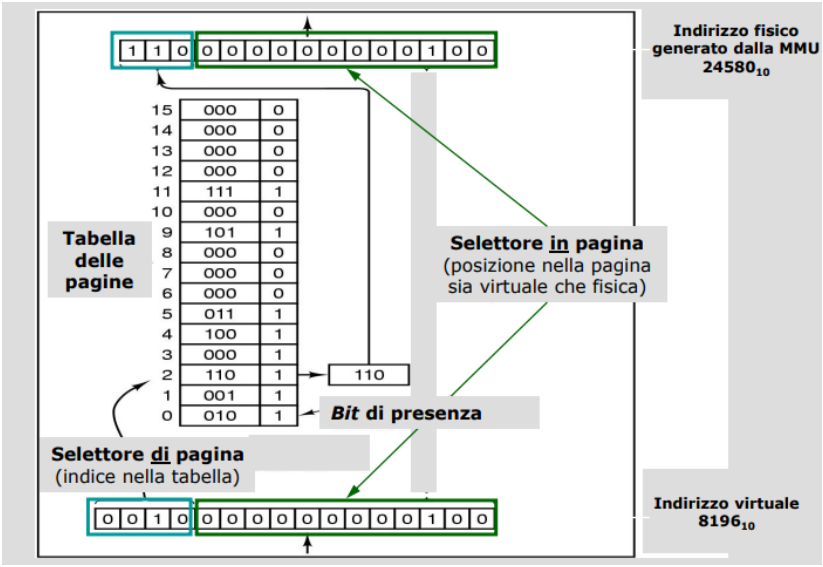
1) Paginazione

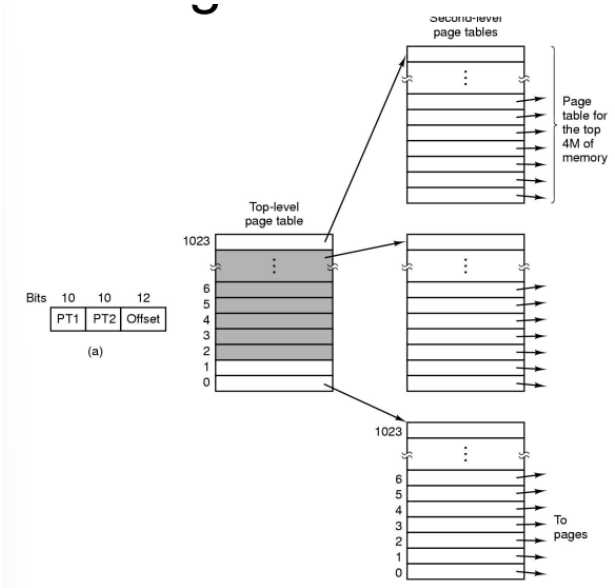
La memoria virtuale è suddivisa in unità a dimensione fissa dette pagine. La RAM è suddivisa in unità "cornici" ampie come le pagine (page frame). Il trasferimento da e verso disco avvengono sempre in pagine. Di ogni pagina occorre sapere se sia presente in RAM oppure no.
– Bit di presenza
– Se una pagina è assente quando riferita si genera un evento page fault gestito dal S/O tramite trap

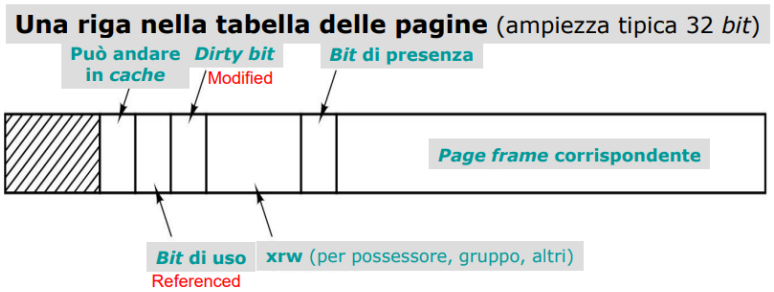


La traduzione da indirizzo virtuale a fisico avviene tramite una tabella delle pagine
– Indicizzata per numero di pagina
• Indirizzo $_{fisico}$ = ϕ (indirizzo $_{virtuale}$)
• La tabella può essere molto grande
– Indirizzi virtuali da 32 bit e pagine da 4 KB
→ memoria virtuale da 4 GB = 1 M pagine!
• Ciascun processo ha la sua (grande) tabella delle pagine
– Poiché ha il suo spazio di indirizzamento virtuale

La traduzione deve essere molto *veloce*
– Ogni istruzione potrebbe fare riferimento più volte alla tabella delle pagine
• Dunque, se un'istruzione impiega ad es. 4 ns, allora il riferimento alla page table deve avvenire in circa 1 ns, viceversa sarà un bottleneck del sistema
– Ogni indirizzo emesso dal processo (istruzione o operando) deve essere tradotto
• Semplicemente (e concettualmente) potrebbe utilizzare un vettore di registri (uno per ogni pagina virtuale) caricato a ogni cambio di contesto (vedi figura nella slide seguente)
– Lineare e non si rischia di dover accedere a memoria per scoprire il riferimento, ma costoso cambiare tutti i registri ad ogni context switch
• Oppure come una struttura sempre residente in RAM
– Un singolo registro punta all'inizio della page table
– Difficile che sia usato come soluzione in modo puro







L'indirizzo di disco ove la pagina si trova quando non è in RAM non è nella tabella
– La tabella delle pagine serve alla MMU (hardware)
– Il caricamento della pagina da disco viene effettuato dal S/O (software) all'occorrenza di un page fault
– L'informazione dell'uno non serve all'altro

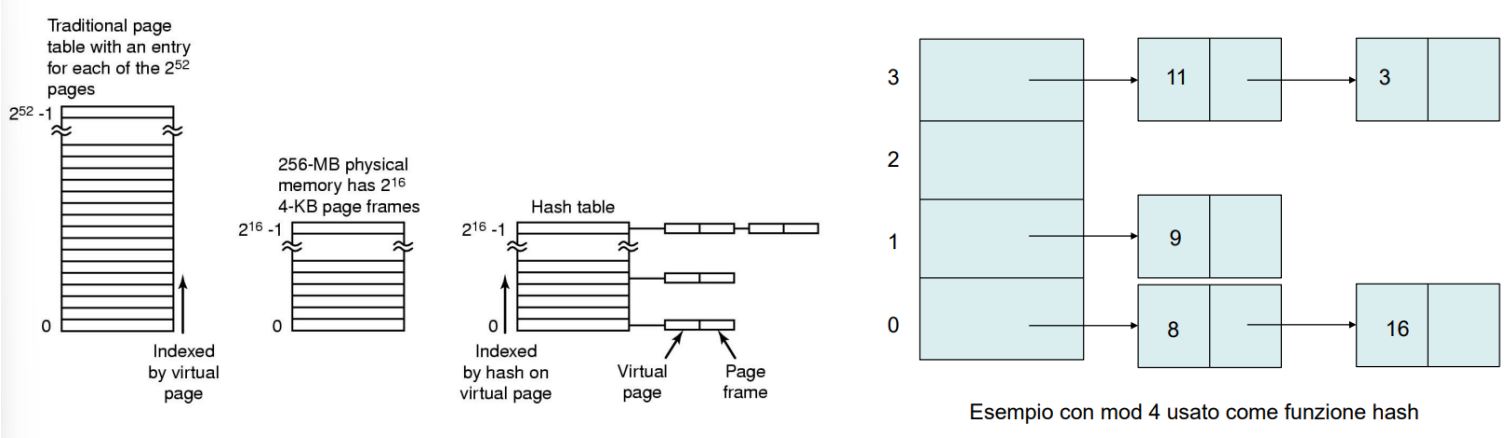
La tabella delle pagine è così grande che non può risiedere su registri
– Dunque, deve stare in RAM
– Riferirla per ogni indirizzo emesso (istruzioni e operandi) ha un impatto devastante sulle prestazioni
• Serve una struttura supplementare (HW) più agile che ne sia come una cache
– Piccola memoria associativa che consente scansione parallela (translation lookaside buffer, TLB)
• Solitamente interna alla MMU
• Ricerca su tutte le righe simultaneamente
– Basata sull'osservazione che un processo in genere usa più frequentemente poche pagine
• Località spaziale e temporale dei riferimenti

Ogni indirizzo emesso verso la MMU viene prima trattato con la TLB
– Se la sua pagina è presente e l'accesso richiesto è permesso la traduzione avviene tramite TLB
• Senza accedere alla tabelle delle pagine
– Se non presente si ha l'equivalente di una cache miss e le informazioni richieste vengono caricate in TLB dalla tabella delle pagine
• Rimpiazzando una cella in TLB e riflettendone il valore nella tabella delle pagine, ma solo se cambiato.

Scritto da Gabriel

Oggi le TLB sono prevalentemente realizzate in software invece che in hardware nelle MMU
– Le prestazioni sono accettabili
– La MMU ne guadagna in semplicità e riduzione di spazio che viene dedicato ad altri usi ritenuti più vantaggiosi (cache)
Con le architetture a 64 bit però le tabelle delle pagine assumono dimensioni proibitive
– 64 bit → memoria virtuale da 16 EB (Exabyte) • (1 E = 1 G = 1 G)
– Pagine da 4 KB → 4 P pagine • (1 P = 1 M = 1 G)
– 32 bit per pagina in tabella → ampiezza 16 PB (Petabyte)

Serve un'altra soluzione, che impiega una *tabella invertita*.
– Non più una riga per pagina ma per page frame in RAM, con considerevole risparmio di spazio
– La traduzione da virtuale a fisico diventa però molto più complessa
• Poiché la pagina potrebbe risiedere in qualunque page frame bisognerebbe scandire l'intera tabella per trovarla
– Per ogni indirizzo emesso dal processo!
– Grande dispendio di tempo
– Ricerca velocizzata dall'uso di TLB
– E anche realizzando la tabella invertita come una tabella hash indicizzata da f_{Hash} (indirizzo $_{virtuale}$)
I dati relativi alle pagine i cui indirizzi virtuali indicizzino una stessa riga di tabella vengono collegati in lista



Esempio con mod 4 usato come funzione hash

Quando si produce un page fault il S/O deve rimpiazzare una pagina
– Salvando su disco la pagina rimossa, ma solo se modificata nell'uso
• Inopportuno rimpiazzare pagine in uso frequente
– Altrimenti si paga prezzo doppio dovendole riportare troppo presto in RAM
• Problema del tutto analogo a quello della cache
– Anche di quelle emulate a software per la gestione di informazioni logiche

Rimpiazzo ottimale (optimal replacement)
– Rimpiazza la pagina in memoria che non sarà usata per maggior tempo
• La scelta perfetta non è realizzabile
– Perché il S/O non ha modo di sapere quali pagine il processo accederà in futuro
• Un po' come scegliere il processo più breve
• Le scelte realizzabili sono sempre e solo approssimazioni sotto-ottimali
– Sulla base di osservazioni empiriche sull'uso recente delle pagine attualmente in RAM

NRU (Not Recently Used)
– Per ogni page frame vengono aggiornati
• Bit M (modified), inizializzato a 0 dal S/0
• Bit R (referenced), posto a 0 periodicamente dal S/O per stimare la frequenza d'uso

– Le pagine nei page frame sono classificate in
• Classe 0: non riferita, non modificata
• Classe 1: non riferita, modificata
• Classe 2: riferita, non modificata
• Classe 3: riferita, modificata
– NRU sceglie una pagina a caso nella classe non vuota a indice più basso

FIFO
– Rimuove la pagina di ingresso più antico in RAM
• Basta una lista ordinata di page frame
– Ogni inserimento viene marcato in coda e la rimozione avviene dalla testa

Second chance
– Corregge FIFO rimpiazzando solo le pagine con bit R = 0
• Altrimenti il page frame viene considerato appena caricato, posto in fondo alla coda e R viene posto a 0
• Degenera in FIFO quando tutti i page frame siano stati recentemente riferiti

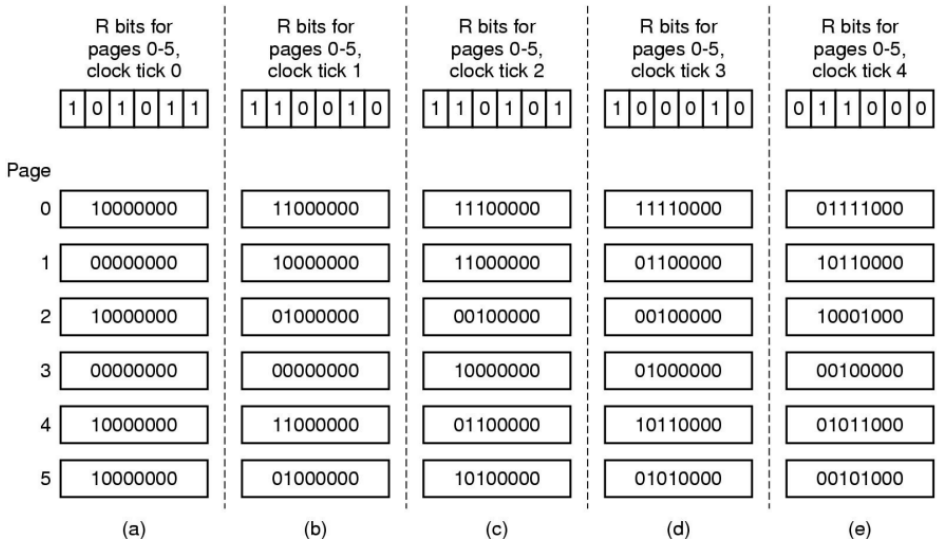
Orologio
– Come SC ma i page frame sono mantenuti in una lista circolare
• L'indice di ricerca si muove come una lancetta

LRU (Least Recently Used)
– Approssima l'algoritmo ottimale
– Richiede lista aggiornata ad ogni riferimento a memoria
– Necessita di hardware dedicato

NFU (Not Frequently Used)
– Realizzabile a software
– Per ogni page frame aggiorna periodicamente un "contatore" C che cresce di più se R = 1
– Richiede l'implementazione di una struttura dati aggiuntiva.
– L'assistenza hardware è elevata.
– In LRU il rilevamento degli errori è difficile rispetto ad altri algoritmi.
– Ha un'accettabilità limitata.
– LRU è molto costoso da gestire.

Aging (not frequently used modificato)
– Realizzabile a software
– Per ogni page frame aggiorna periodicamente un "contatore" C che cresce di più se R = 1
• Non incrementa C con R ma gli inserisce R a sinistra
– Approssima LRU con differenze importanti
• Valuta solo periodicamente (a grana grossa)
• Usando N bit per C perde memoria dopo N aggiornamenti

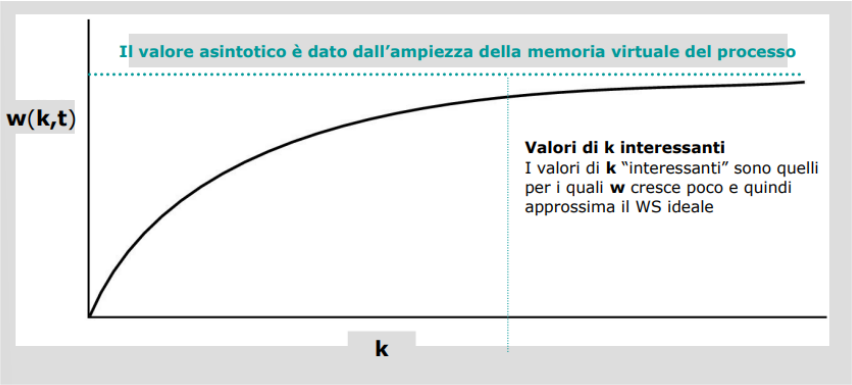
Invece di incrementare semplicemente i contatori delle pagine referenziate, dando la stessa importanza ai riferimenti di pagina indipendentemente dal tempo, il contatore di riferimento di una pagina



Scritto da Gabriel

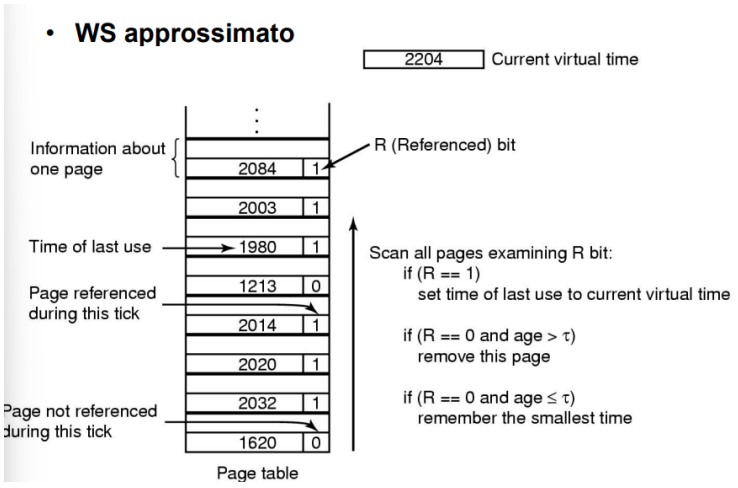
viene prima spostato a destra (diviso per 2), prima di aggiungere il bit referenziato a sinistra di quel numero binario. Ad esempio, se una pagina ha fatto riferimento ai bit 1,0,0,1,1,0 negli ultimi 6 tick di orologio, il suo contatore di riferimento avrà il seguente aspetto: 10000000, 01000000, 00100000, 10010000, 11001000, 01100100. I riferimenti alle pagine più vicine al momento attuale hanno un impatto maggiore rispetto ai riferimenti alle pagine di molto tempo fa. Ciò garantisce che le pagine a cui si fa riferimento più di recente, anche se meno frequentemente, abbiano una priorità maggiore rispetto alle pagine a cui si fa riferimento più frequentemente in passato. In questo modo, quando una pagina deve essere sostituita, viene scelta quella con il contatore più basso.

Studi accurati mostrano come i processi emettano la maggior parte dei loro riferimento entro un ristretto spazio locale (località dei riferimenti)
• Working set (WS) è l'insieme di pagine che un processo ha in uso a un dato istante
– Se la memoria non basta ad accogliere il WS si crea il fenomeno di thrashing
– Se il WS viene caricato prima dell'esecuzione si ha prepaging (evitando page fault)
– $w(k,t)$ è l'insieme di pagine che soddisfano i k riferimenti emessi al tempo t
– Funzione monotonica crescente

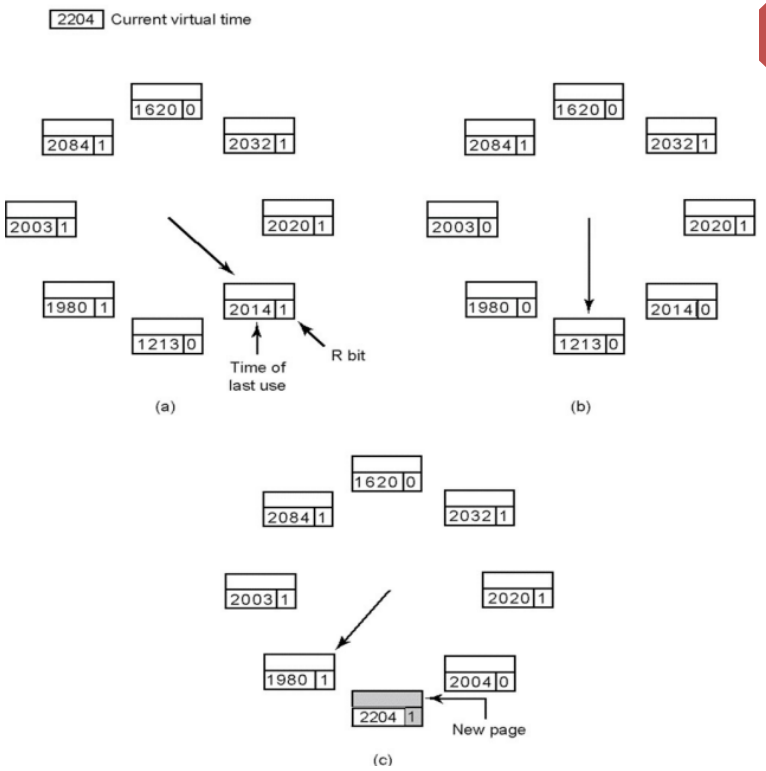


Se si conoscesse il WS dei processi le pagine da rimpiazzare sarebbero quelle che non vi fossero comprese
• Conoscere precisamente il WS dei processi a tempo d'esecuzione è però troppo costoso
– Quanto deve valere k?
– Più facile fissare t come $(t, t + \Delta t)$
• Considerando t come valore dell'effettivo tempo di esecuzione di quel processo (tempo virtuale corrente e non del tempo trascorso
• WS è fatto dalle pagine riferite dal processo nell'ultimo Δt

WS approssimato
– Simile all'Aging
• Ogni page frame in RAM ha un attributo temporale usato insieme all'attributo riferito (R = 1)
– Tale attributo prende il valore t del tempo virtuale corrente all'arrivo di un page fault
– R e M sono posti a 1 dall'hardware
– R è posto a 0 (se non in uso) da un controllo periodico e al page fault
• Al page fault sono rimpiazzabili le pagine con R = 0 e attributo antecedente all'intervallo $(t - \Delta t, t)$
– Se non ci sono, si prende la più vecchia con R = 0
– Se all'istante t tutti i page frame avessero R = 1 verrebbe rimpiazzata una pagina scelta a caso, con M = 0
• Nel caso peggiore bisogna scandire l'intera RAM!

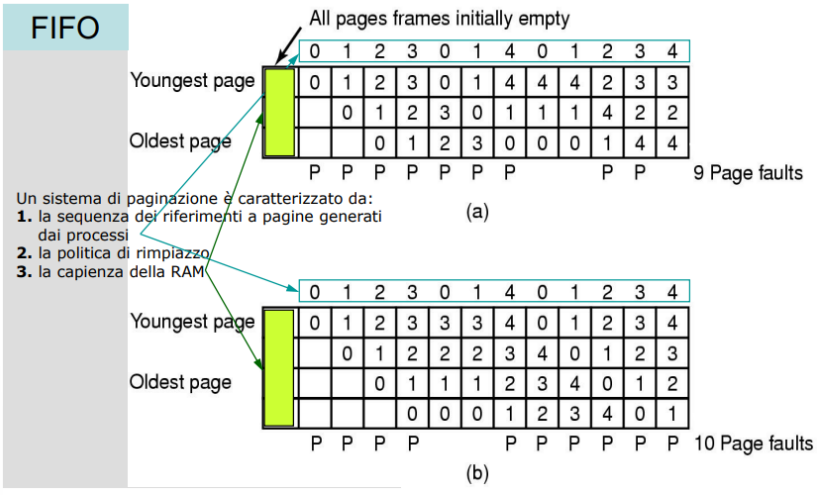


WS approssimato con orologio
– Page frame organizzati in lista circolare
• Come per l'orologio semplice
• Ma con le informazioni del WS approssimato
– Una "lancetta" indica il page frame corrente
• Al page fault se R = 1 la lancetta avanza e R = 0
• Se R = 0 si valuta l'attributo temporale
– Se fuori da $w(k,t)$ e con M = 0 allora rimpiazzo
– Altrimenti il page frame va in una coda di trasferimento su disco e la lancetta avanza alla ricerca di un page frame rimpiazzabile direttamente. Quando N pagine in coda si trasferisce su disco
• Se nessun page frame è rimpiazzabile allora si sceglie una pagina con M = 0 altrimenti quella cui punta la lancetta

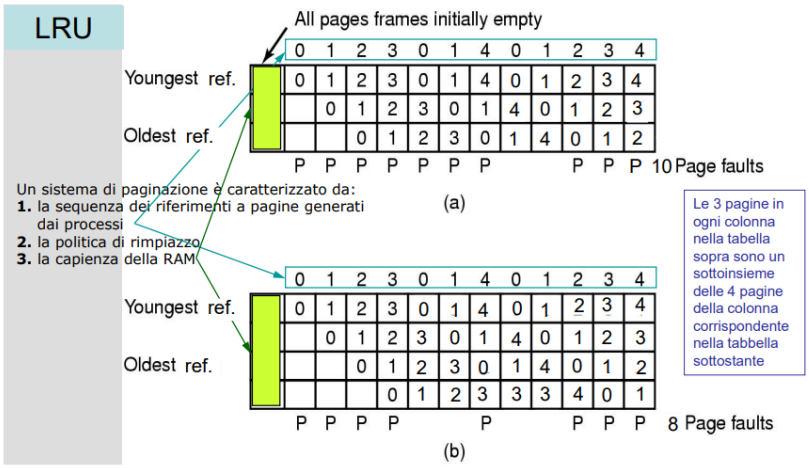


Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Nel 1969 Lazlo Belady mostrò che la frequenza di page fault non sempre decresce al crescere dall'ampiezza della RAM
– Un semplice contro-esempio usando FIFO come strategia di rimpiazzo
  • Sequenza di riferimenti: 0 1 2 3 0 1 4 0 1 2 3 4
  • RAM con 3 page frame : 9 page fault
  • RAM con 4 page frame : 10 page fault
  • LRU è immune dall'anomalia di Belady
– Ma la sua forma "pura" è irrealizzabile

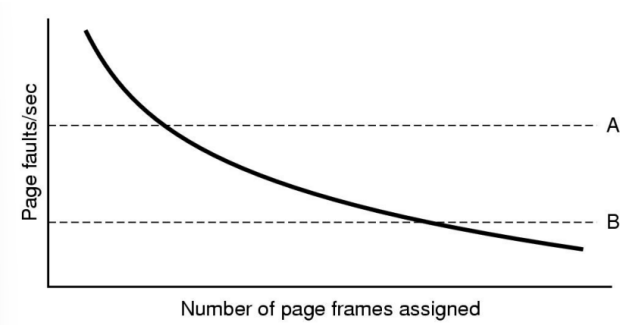


Una classe di algoritmi particolarmente interessante è quella che soddisfa la proprietà:
– $M(m,r) \subseteq M(m+1,r)$
dove m rappresenta il numero di page frame, mentre r sono i riferimenti
– "assumendo gli stessi riferimenti, le pagine caricate con m page frame sono un sottoinsieme di quelle caricate con $m+1$ page frame"
• Detti *stack algorithms*
– Sono immuni dall'anomalia di Belady (LRU, Optimal Replacement)



Nel rimpiazzare una pagina occorre scegliere consapevolmente tra:
Politiche locali
• Rimpiazzo nel WS del processo che ha causato il page fault
• In tal caso ogni processo conserva una quota fissa di RAM
Politiche globali
• La scelta avviene tra page frame senza distinzione di processo
• L'allocazione di RAM a disposizione di ogni processo varia dinamicamente nel tempo

Le politiche globali sono più efficaci
– Specialmente se l'ampiezza del WS può variare durante l'esecuzione
• Però bisogna decidere quanti page frame assegnare a ogni singolo processo
• Le politiche locali hanno prestazioni inferiori
– Se il WS di un processo cresce l'allocazione fissa causa rimpiazzi indesiderati
• Thrashing
– Anche con RAM disponibile non usata da altri processi
– Se il WS si riduce si ha invece spreco di memoria
• Non tutte le politiche si adattano all'uso in entrambe le varianti



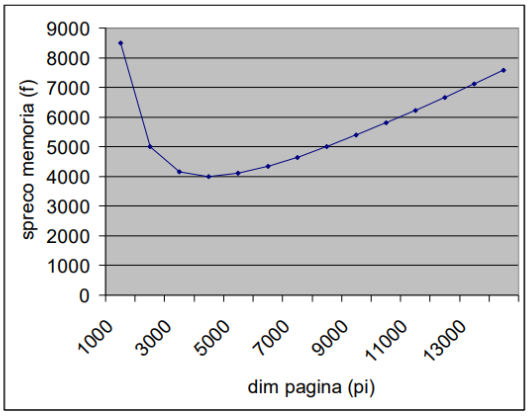
Controllo del carico
– Anche con le migliori politiche può accadere che a volte il sistema subisca thrashing
• Se i WS di tutti i processi eccedono la capacità di memoria
– PFF (Page Fault Frequency) indica che alcuni hanno bisogno di più memoria ma nessuno ha bisogno di meno memoria
• SWAP!
– Rimuoviamo in successione alcuni processi finché il thrashing si ferma

Quale dimensione di pagina?
– Pagine ampie
• Maggiore rischio di frammentazione interna
– In media ogni processo lascia inutilizzata metà del suo ultimo page frame
– Pagine piccole
• Maggiore ampiezza della tabella delle pagine

Per $\sigma = 1$ MB e $\varepsilon = 8$ B…

Il valore ottimo può essere definito matematicamente
– σ B dimensione media di un processo
– π B dimensione media di una pagina
– ε B per riga in tabella delle pagine
– Spreco per processo come $f(\pi) = (\sigma / \pi) \times \varepsilon + \pi / 2$
  • Parte di tabella delle pagine + frammentazione interna
  • Derivata prima è $- \sigma \varepsilon / \pi^2 + 1/2$
  • Ponendo uguale a zero si ha che il minimo di $f(\pi)$ si ha per $\pi = \sqrt{(2 \sigma \varepsilon)}$

$$f(\pi) = (\sigma / \pi) \times \varepsilon + \pi / 2$$



Per $\sigma = 1$ MB e $\varepsilon = 8$ B si ha $\pi = 4$ KB
• Per RAM di ampiezza crescente può convenire un valore di π maggiore
– Ma di certo non linearmente
• In generale la memoria virtuale non è distinta per dati e istruzioni
– Nella prima metà del ’70 vi sono stati elaboratori importanti (PDP-11) che fornivano invece spazi di indirizzamento distinti
• Programmed Data Processor (2 KB cache, 2 MB RAM)

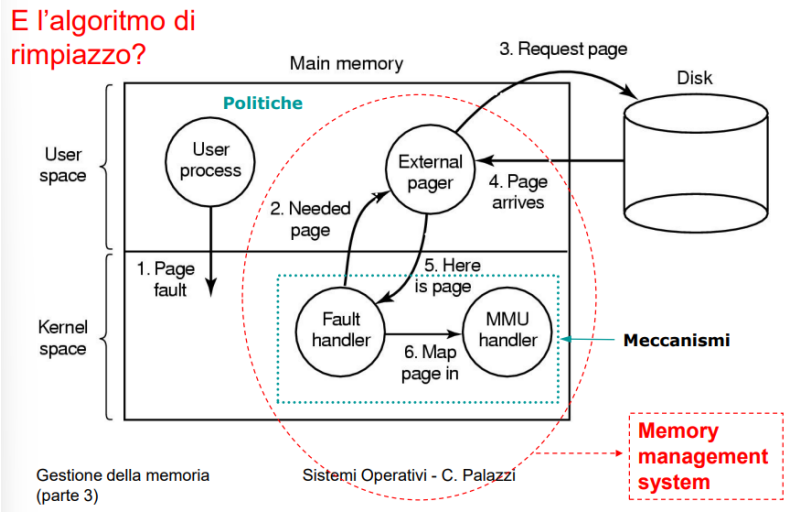
Il S/O compie azioni chiave
– A ogni creazione di processo
  • Per determinare l’ampiezza della sua allocazione
  • Per creare la tabella delle pagine corrispondente
– A ogni cambio di contesto
• Per caricare la MMU e “pulire” la TLB
– A ogni page fault
  • Per analizzare il problema e operare il rimpiazzo
– A ogni terminazione di processo
  • Per rilasciarne i page frame
  • Per rimuoverne la tabella delle pagine

Per trattare un page fault bisogna capire quale riferimento è fallito
  – Per poter completare correttamente l’istruzione interrotta
• Il Program Counter dice a quale indirizzo il problema si è verificato
  – Ma non sa distinguere tra istruzione e operando
• Capirlo è compito del S/O
  – Orrendamente complicato dai molti effetti laterali causati dagli “acceleratori” hardware
  – Il S/O deve annullare lo stato erroneo e ripetere daccapo l’istruzione fallita

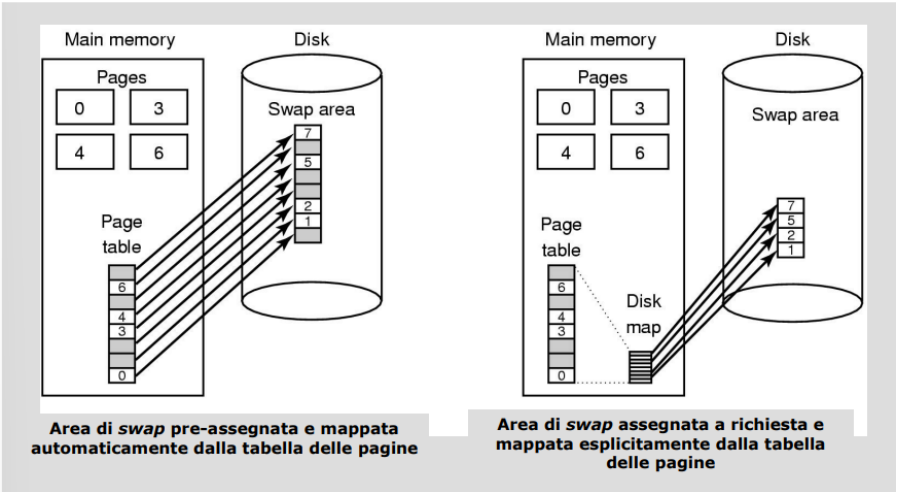
Realizzazione della paginazione
• Page fault: l’hw fa trap al kernel e salva il PC sullo stack
• Un programma assembler salva i dati nei registri e poi chiama il sistema operativo
• Il S.O. scopre il page fault e cerca di capire di quale pagina si tratti (visionando i registri o recuperando il PC e simulando l’istruzione)
• Ottenuto l’indirizzo virtuale causa del page fault, il S.O. verifica che si tratti di indirizzo valido (altrimenti *kill* del processo) e cerca un page frame vuoto o con pagina rimpiazzabile

Scritto da Gabriel

• Se la pagina da rimpiazzare è dirty (M=1), si imposta il suo spostamento su disco (il processo corrente viene sospeso nel frattempo) e il frame viene bloccato
• Quando il page frame è libero, vi copia la pagina richiesta (il processo viene di nuovo sospeso nel frattempo)
• All'arrivo dell'interrupt del disco, la page table è aggiornata e il frame è indicato come normale
• Il PC viene reimpostato per puntare all'istruzione causa del page fault
• Il processo causa del page fault è pronto per esecuzione e il S.O. ritorna al programma assembler che lo aveva chiamato
• Il programma assembler ricarica i registri e altre info; poi torna in user space per continuare l'esecuzione



Un'area del disco può essere riservata per ospitare le pagine temporaneamente rimpiazzate
– Area di swap
• Ogni processo ne riceve in dote una frazione
– Che rilascia alla sua terminazione
– I puntatori (base, ampiezza) a questa zona devono essere mantenuti nella tabella delle pagine del processo
• Ogni indirizzo virtuale mappa nell'area di swap direttamente rispetto alla sua base
• Idealmente
– L'intera immagine del processo potrebbe andare subito nell'area di swap alla creazione del processo
– Altrimenti potrebbe andare tutta in RAM e spostarsi nell'area di swap quando necessario
• Però sappiamo che i processi non hanno dimensione costante
– Allora è meglio che l'area di swap sia frazionata per codice e dati
• Se l'area di swap non fosse riservata allora occorrerebbe ricordare in RAM l'indirizzo su disco di ogni pagina rimpiazzata
– Informazione associata alla tabella delle pagine



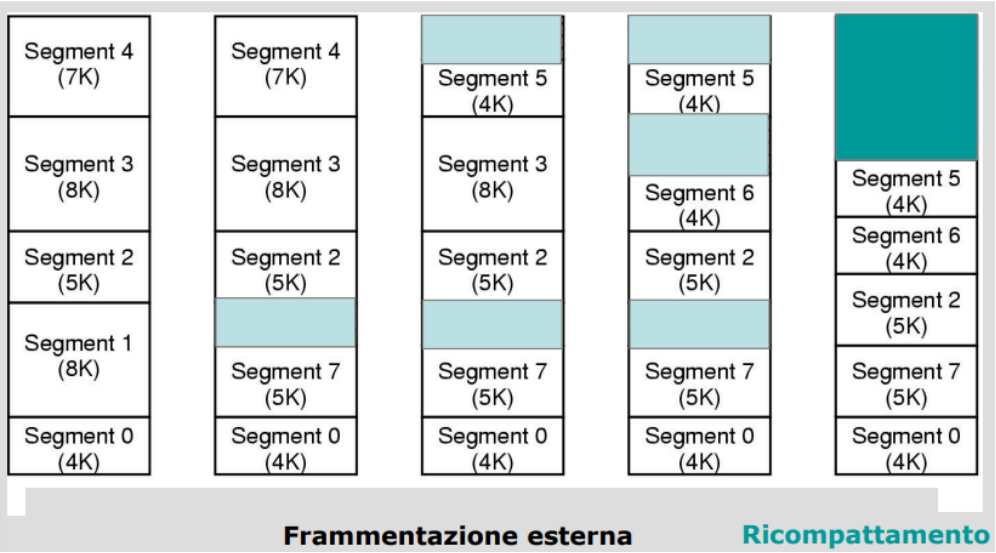
Scritto da Gabriel

LINUX
– Partizione dedicata allo SWAP, con file system apposito
    • Consumo una delle possibili partizioni del disco
– Dimensione impostabile dall'utente in fase di installazione
    • Almeno la stessa dimensione della RAM se si vuole gestire l'ibernazione (copia di tutto il contenuto della RAM nell'area di SWAP e ricaricamento in fase di riattivazione)
Windows (2000, XP, …)
– Uso di file di swap
– hiberfil.sys (usato per copiare la RAM in caso di ibernazione del sistema)
– pagefile.sys (usato quando la memoria RAM non è sufficiente)
– Se il file viene frammentato, le prestazioni calano

Per separare le politiche dai meccanismi
– Conviene svolgere nel nucleo del S/O solo le azioni più delicate
    • Gestione della MMU
– Specifica dell'architettura hardware
    • Trattamento immediato del page fault
– Largamente indipendente dall'hardware
– Demandando il resto della gestione a un processo esterno al nucleo
    • Scelta delle pagine e loro trasferimento
– Trattamento differito del page fault

• Spazi di indirizzamento completamente indipendenti gli uni dagli altri
    – Per dimensione e posizione in RAM
• Entrambe possono variare dinamicamente
• Entità logica nota al programmatore e destinata a contenere informazioni coese
    – Codice di una procedura
    – Dati di inizializzazione di un processo
    – Stack di processo
• Si presta a schemi di protezione specifica
– Perché il tipo del suo contenuto può essere stabilito a priori
    • Ciò che non si può fare con la paginazione
• Causa frammentazione esterna

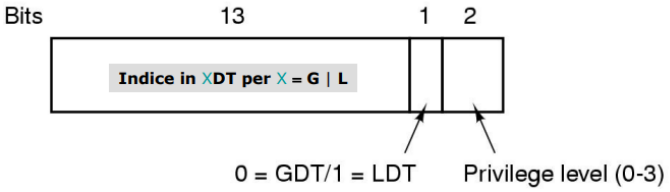
	Paginazione	**Segmentazione**
Il programmatore ne deve essere consapevole	No	Sì
Consente N spazi di indirizzamento lineari	N = 1	N ≥ 1
La sua ampiezza può eccedere la capacità della RAM	Sì	Sì
Consente di separare e distinguere tra codice e dati	No	Sì
Consente di gestire contenuti a dimensione variabile nel tempo	No	Sì
Consente di condividere parti di programmi tra processi	No	Sì
A quale obiettivo risponde	Consentire spazi di indirizzamento più grandi della RAM	Consentire la separazione logica tra aree dei processi e la loro protezione specifica



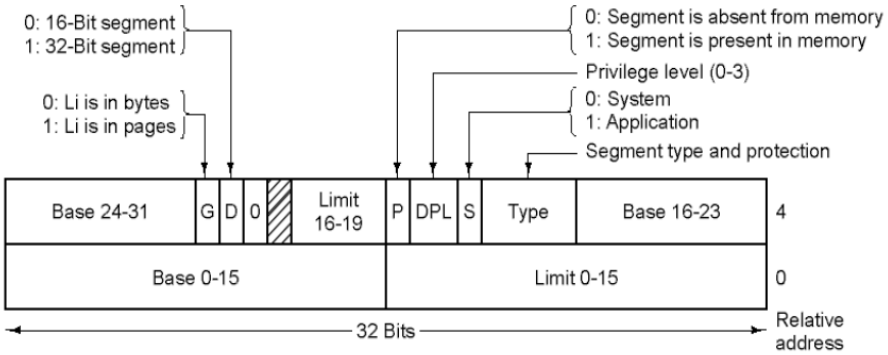
Vista la grande ampiezza potenziale i segmenti sono spesso paginati

- Nel caso del Pentium di Intel
  - Fino a 16 K segmenti indipendenti di ampiezza massima 4 GB (32 bit)
- Una LDT per processo
  - Local Descriptor Table, che descrive i segmenti del processo
- Una singola GDT per l'intero sistema
  - Global Descriptor Table, che descrive i segmenti del S/O

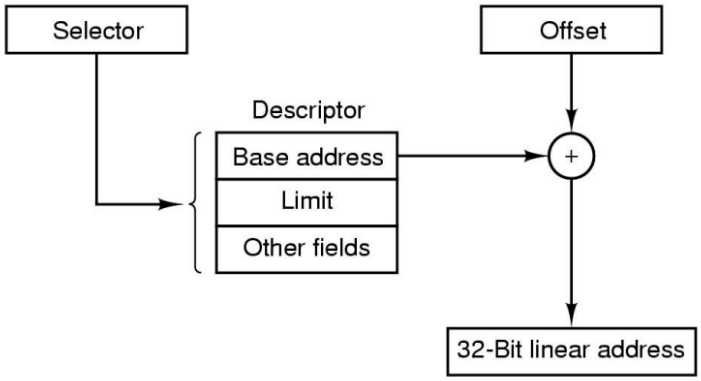
Per accedere a un segmento, un programma Pentium prima carica selettore di quel segmento in uno dei sei registri di segmento



- 6 registri di segmento
  - Di cui 1 denota il segmento corrente
- LDT e GDT contengono $2^{13} = 8$ K descrittori di segmento
  - I descrittori di segmento sono espressi su 8 B
    - La **base** del segmento in RAM è espressa su 32 *bit*
    - Il **limite** su 20 *bit* per verificare la legalità dell'*offset* fornito dal processo
      - Consente ampiezza massima a 1 MB (per **granularità** a B)
      - Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)

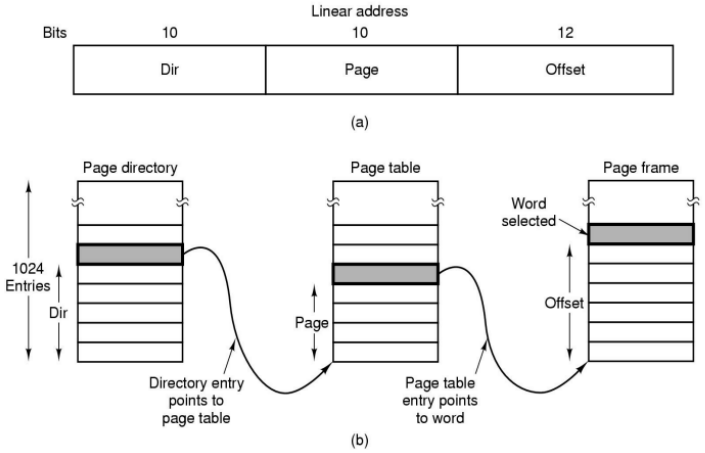


Descrittore di segmento di Pentium relativo al codice (lievi differenze con quello relativo ai dati)



Conversion di una coppia (selettore, offset) in un indirizzo lineare

L'indirizzo **lineare** ottenuto da (base di segmento + *offset*) può essere interpretato come
- Indirizzo **fisico** se il segmento considerato non è paginato
- Indirizzo **logico** altrimenti
  - Nel qual caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa
    - 10 *bit* : indice in catalogo di tabelle delle pagine
      - » 2^{10} righe da 32 *bit* ciascuna (base di tabella denotata)
    - 10 *bit* : indice in tabella delle pagine selezionata
      - » 2^{10} righe da 32 bit ciascuna (base di *page frame*)
    - 12 *bit* : posizione nella pagina selezionata
      - » *Offset* in pagina da 4 KB



L'indirizzo **lineare** mappato sullo spazio virtuale

Esercizi gestione memoria

Esercizio 1 - soluzione

Dato un sistema di *swapping* e una memoria con zone disponibili di ampiezza: 8, 4, 14, 18, 17, 9, 12, 15 KB, in questo ordine, indicare quale area venga prescelta dalla politica ***Next Fit*** a fronte della richiesta di caricamento di un segmento di ampiezza 3 KB dopo aver caricato un segmento ampio 12 KB:

A: 4 KB
B: 18 KB
C: 14 KB
D: 9 KB

E con ***First fit? Best fit? Worst fit?***

Esercizio 2 - soluzione

Sia data memoria dotata di 4 *page frame*, inizialmente libere, e 8 pagine di memoria virtuale. Utilizzando la politica FIFO per il rimpiazzo delle pagine, indicare quanti *page fault* si verifichino a fronte della stringa di riferimenti: 0 1 7 2 3 2 7 1 0 3:

A: 4
B: 3
C: 6
D: 2

Riferimenti	0	1	7	2	3	2	7	1	0	3
RAM	0	1	7	2	3	3	3	3	0	0
		0	1	7	2	2	2	2	3	3
			0	1	7	7	7	7	2	2
				0	1	1	1	1	7	7
	P	P	P	P	P			P		

6 PF

Esercizio 3

Si consideri la seguente serie di riferimenti a pagine di memoria:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Si considerino le seguenti politiche di rimpiazzo:
- FIFO
- LRU
- Optimal

Quanti *page fault* avvengono considerando un numero di *page frame* della RAM di 1, 2, 3, 4, 5, 6, 7 ?

Esercizio 3 - FIFO

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
Meno vecchio	1	2	3	4	4	1	5	6	2	1	1	3	7	6	6	2	1	1	3	6
RAM		1	2	3	3	4	1	5	6	2	2	1	3	7	7	6	2	2	1	3
più vecchio			1	2	2	3	4	1	5	6	6	2	1	3	3	7	6	6	2	1
	P	P	P	P		P	P	P	P	P		P	P	P		P	P		P	P

16 PF

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
RAM	1	2	3	4	4	4	5	6	2	1	1	3	7	6	6	2	1	1	3	3
		1	2	3	3	3	4	5	6	2	2	1	3	7	7	6	2	2	1	1
			1	2	2	2	3	4	5	6	6	2	1	3	3	7	6	6	2	2
				1	1	1	2	3	4	5	5	6	2	1	1	3	7	7	6	6
	P	P	P	P		P	P	P	P	P		P	P	P		P	P		P	

14 PF

Esercizio 3 - LRU

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
Rif. + recentem.	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
RAM		1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3
Rif. - recentem.			1	2	3	4	2	1	5	6	6	1	1	3	7	6	3	3	1	2
	P	P	P	P		P	P	P	P	P		P	P	P		P	P			P

15 PF

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
RAM	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
		1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3
			1	2	3	4	2	1	5	6	6	1	2	3	7	6	3	3	1	2
				1	1	3	4	2	1	5	5	6	1	2	2	7	6	6	6	1
	P	P	P	P		P	P			P	P	P			P					

10 PF

Esercizio 3 - OPTIMAL

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
RAM	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	6
		2	2	2	2	2	2	2	2	2	2	2	7	7	7	2	2	2	2	2
			3	4	4	4	5	6	6	6	6	6	6	6	6	6	1	1	1	1
	P	P	P	P			P	P				P	P			P	P			P

Ultimo 6 riferito può essere inserito in qualsiasi posizione

11 PF

Riferimenti	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
RAM	1	1	1	1	1	1	1	1	1	1	1	1	7	7	7	7	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
				4	4	4	5	6	6	6	6	6	6	6	6	6	6	6	6	6
	P	P	P	P			P	P				P				P				P

8 PF

Esercizio 3 - Soluzioni

# page frame	FIFO	LRU	*Optimal*
1	20	20	20
2	18	18	15
3	16	15	11
4	14	10	8
5	10	8	7
6	10	7	7
7	7	7	7

Esercizio 4

Sia dato un sistema di gestione della memoria principale basato su segmentazione con indirizzi logici e fisici espressi su 16 *bit*. Si consideri la tabella dei segmenti riportata di seguito, ove il prefisso 0x denota l'uso di notazione Esadecimale:

Segmento	Base	Limite
$0x0$	$0x0219$	$0x600$
$0x1$	$0x2300$	$0x014$
$0x2$	$0x0090$	$0x100$
$0x3$	$0x1327$	$0x580$
$0x4$	$0x1952$	$0x096$
...	...	...
$0xF$	...	...

Si mostri graficamente la mappa di memoria corrispondente, indicando anche il suo grado percentuale di occupazione.

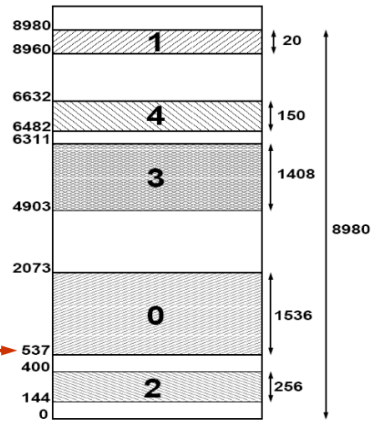
Esercizio 4 - soluzione

Con un po' di calcoli:

0x219 =
2×256 + 1×16 + 9×1 =
537

Eccetera ...

L'area di memoria occupata va dall'indirizzo 0 all'indirizzo 8980. L'ampiezza complessiva dei segmenti in tale zona misura 3370 B con grado di occupazione: **37, 53%**.



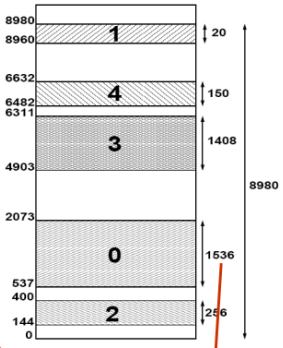
Esercizio 4.1

Si fornisca l'indirizzo fisico corrispondente ai seguenti indirizzi virtuali espressi in notazione decimale:

a. $< 0, 1984 >$
b. $< 1, 18 >$
c. $< 2, 250 >$
d. $< 3, 1400 >$
e. $< 4, 112 >$

Esercizio 4.1 - soluzione

Ricordando che ...



Allora:

indirizzo virtuale	indirizzo fisico
$< 0, 1984 >$	Errore: indirizzo illegale.
$< 1, 18 >$	$8960 + 0018 = 8978$
$< 2, 250 >$	$0144 + 0250 = 0394$
$< 3, 1400 >$	$4903 + 1400 = 6303$
$< 4, 112 >$	$6482 + 0112 = 6594$

1984 > 1536

Esercizio 4.2

Indicare il modo nel quale i segmenti di indice 0–4 possano essere mappati su disco, assumendo blocchi di ampiezza 1 KB, calcolando anche la quantità percentuale di memoria *sprecata* di conseguenza.

Esercizio 4.2 - soluzione

Blocchi di ampiezza 1 KB = 1024 B, quindi:

Segmento	Ampiezza	# Blocchi occupati
0	1536 B	2
1	20 B	1
2	256 B	1
3	1408 B	2
4	150 B	1
Totale	3370 B	7

Totale B × Blocchi = 1024 B × 7 = 7168 B
Totale spreco = (7168 – 3370) B = 3798 B
Spreco di memoria: 3798 B / 7168 B = **53%**

Esercizio 5

Si consideri la matrice (o *array* bidimensionale):

```
int A[][] = new int[100][100];
```

Si assuma che la posizione **`A[0][0]`** di tale matrice sia posta alla locazione **200** di una memoria paginata con **3** pagine di dimensione **200** B.
Si assuma che un valore **`int`** occupi **1** B.
Si assuma che le pagine siano inizialmente tutte vuote e che il processo (il cui codice occupa esattamente **200** B) abbia la seguente esecuzione:

```
for (int i = 0; i < 100; i++){
   for (int j = 0; j < 100; j++){
      A[i][j] = 0;}}
```

Quanti *page fault* saranno generati usando LRU?

La matrice `A[i][j]` è scritta linearmente in memoria:

`A[0][0],A[0][1],A[0][2],…,A[0][99],A[1][0], A[1][1],…,A[99][99]`

Il processo azzera le celle della matrice proprio nel loro ordine di memorizzazione.

Quindi inizialmente verranno caricati nelle pagine
Pagina 0: Il programma (che occupa esattamente 200 B)
Pagina 1: Celle da A[0][0] a A[1][99] incluse
Pagina 2: Celle da A[2][0] a A[3][99] incluse

A ogni iterazione del programma la memoria viene acceduta per leggere l'istruzione successiva
→ la pagina relativa al processo verrà continuamente acceduta

I primi 200 azzeramenti faranno accesso a Pagina 1, i successivi 200 a Pagina 2; l'ulteriore azzeramento (cella `A[4][0]`) causerà un *page fault* ; la pagina usata meno recentemente è Pagina 1 che verrà sostituita con le celle da `A[4][0]` a `A[5][99]` incluse.
Arrivati all'azzeramento di `A[6][0]` sarà Pagina 2 ad essere stata usata meno di recente
E così via, causando in tutto 1 *page fault* iniziale per caricare il processo in Pagina 0 e 50 *page fault* di Pagina 1 e 2 (25 ciascuno) per caricare le porzioni di matrice.

TOTALE = 51 *page fault*

Esercizio 6 (1/2)

Si consideri un sistema dotato di memoria virtuale, con memoria fisica divisa in 8 *page frame* condivisa da 4 processi contemporaneamente attivi: A, B, C e D. Si supponga che all'istante 100 lo stato della memoria sia il seguente (con il campo riferita avente SI = 1 e NO = 0):

Processo	pag. logica	pag. fisica	istante caricam.	R
A	0	7	50	1
A	1	6	37	0
B	5	5	30	1
B	3	4	97	0
C	2	0	15	1
C	5	2	70	0
C	9	1	92	0
D	8	3	27	0

Esercizio 6 (1/2)

Si supponga che il sistema utilizzi un algoritmo di sostituzione ***second chance* (globale),** e che la lista delle pagine accedute all'istante 100 sia:

C2(15,1) - D8(27,0) - B5(30,1) - A1(37,0) - A0(50,1) - C5(70,0) - C9(92,0) - B3(97,0)

dove C2(15, 1) sia il primo elemento della lista; in tale elemento, C2 indica che si tratta della pagina logica 2 del processo C, mentre (15, 1) indica che tale pagina è stata caricata in memoria all'istante 15 e che il suo bit riferita è uguale a 1.

Si considerino i due seguenti casi, eseguiti in sequenza:
Caso a) all'istante 101, C riferisce la pagina logica 5
Caso b) all'istante 105, A riferisce la pagina logica 9

Assumendo che non vi siano altre operazioni che modifichino i bit riferita, a parte quelle sopra elencate, si scriva la lista delle pagine accedute aggiornata dopo aver eseguito, in successione, i punti a) e b).

Esercizio 6 – Soluzione

Caso a) La pagina C5 è già in memoria (nella pagina fisica 2) e quindi viene marcata come riferita

La lista risultante è dunque:
C2(15,1) - D8(27,0) - B5(30,1) - A1(37,0) - A0(50,1) - C5(70,1) - C9(92,0) - B3(97,0)

Caso b) La pagina logica A9 non si trova in memoria, l'algoritmo *second chance* esamina C2: siccome è stata riferita, viene inserita in coda alla lista ed il campo riferita viene posto uguale a 0; dopodichè, si esamina D8, che non risulta riferita e viene dunque eliminata. La lista risultante è dunque:

B5(30,1) - A1(37,0) - A0(50,1) - C5(70,1) - C9(92,0) - B3(97,0) - C2(15,0) - A9(105,1)

E se la politica di rimpiazzo fosse stata **locale?**

Allora è come se ci fossero liste separate e in particolare quella di C risulta
C2(15,1) - C5(70,0) - C9(92,0)

Quindi per il **Caso a)**, siccome C5 è già in memoria (nella pagina fisica 2), essa viene marcata come riferita e la lista risultante è dunque: C2(15,1) - C5(70,1) - C9(92,0)

Invece, per A, la lista corrispondente sarebbe: A1(37,0) - A0(50,1)

Quindi per il **Caso b)**, siccome la pagina logica A9 non si trova in memoria, l'algoritmo *second chance* esamina A1 che non risulta riferita e viene dunque rimpiazzata.
La lista risultante è dunque: A0(50,1) - A9(105,1)

Per completezza, se la lista (per ipotesi) fosse stata A1(37,1) - A0(50,0)
Allora A1 sarebbe stato portato in fondo con bit riferito azzerato e poi A0 sarebbe stato rimpiazzato, inserendo comunque A9 in fondo, con risultato: A1(37,0) – A9(105,1)

File system

La maggior parte dell'informazione applicativa(i dati) ha durata, ambito e dimensione più ampi della vita delle applicazioni che la usano. Queste sono le 3 esigenze principali
• Persistenza dei dati
• Possibilità di condivisione dei dati tra applicazioni distinte
• Nessun limite di dimensione fissato a priori
• Il file system è il servizio di S/O progettato per soddisfare questi bisogni
Il termine *file* designa un insieme di dati correlati, residenti in memoria secondaria e trattati unitariamente
Il termine *file system (FS)* designa la parte di S/O che si occupa di file in termini di
– Organizzazione
– Gestione
– Realizzazione
– Accesso

La progettazione di FS affronta 2 problemi chiave
– Cosa occorre offrire all'utente applicativo e secondo quali forme concrete
  • Modalità di accesso a file
  • Struttura logica e fisica di file
  • Operazioni ammissibili su file

– Come ciò possa essere realizzato in modo pratico ed economico
    • Garantendo la massima indipendenza dall'architettura fisica di supporto

Il file è un concetto logico realizzato tramite meccanismi di astrazione
– Per salvare informazione su memoria secondaria e potendola ritrovare in seguito senza conoscerne né la struttura logica e fisica né il funzionamento
– All'utente non interessa come ciò avviene
– Interessa invece poter designare le proprie unità di informazione mediante nomi logici unici e distinti
– L'utente vede e tratta solo nomi di file
– Le caratteristiche distintive di un file sono
    • Attributi (tra cui il nome)
    • Struttura interna
    • Operazioni ammesse

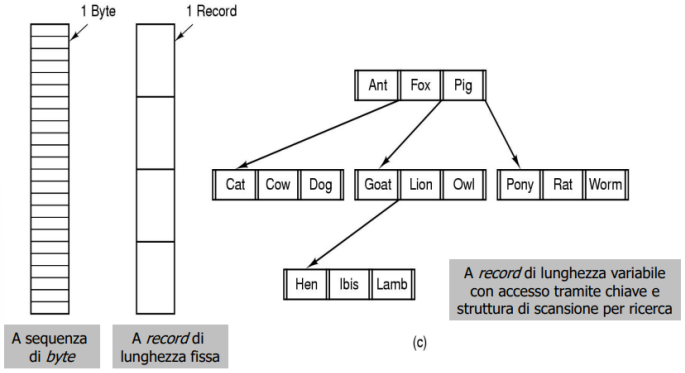
Vediamo nel dettaglio *gli attributi* di un file:

Nome
– Stringa composta da 8 – 255 caratteri, inclusi numeri e caratteri speciali
– Con ≥ 1 estensioni che possono designare il "tipo" di file come visto dall'utente
• MS-DOS (base di Windows 95 e Windows 98)
    – Nomi da 1 – 8 caratteri, con ⍰ 1 estensione da 1 – 3 caratteri, designante, senza distinzione tra maiuscolo e minuscolo (case insensitive)
• UNIX (base di GNU/Linux)
    – Nomi fino a 14 (ora 255) caratteri, case sensitive, con estensioni, solo informative, senza limite di numero e di ampiezza
    – In generale, l'utente può configurare presso il S/O l'associazione tra l'ultima estensione del file ed il tipo applicativo corrispondente
• Dimensione corrente
• Data di creazione
    – Può non essere mostrata
• Data di ultima modifica
    – Indica la "freschezza" del contenuto
• Creatore e possessore
    – Possono essere distinti
• P.es.: il compilatore crea file di proprietà dell'utente
• Permessi di accesso
    – Lettura, scrittura, esecuzione

			Flag
Protezione	Permesso di accesso al *file*		
Password	Chiave di accesso al *file*		
Creatore	Identità del processo che ha creato il *file*		
Proprietario	Identità del processo utilizzatore del *file*		
Uso	0 – lettura/scrittura	1 – sola lettura (*read-only*)	
Visibilità	0 – normale	1 – *file* non visibile (*hidden*)	
Livello	0 – normale	1 – *file* di sistema	
Archiviazione	0 – salvato (*backed up*)	1 – non salvato	
Tipo di contenuto	0 – ASCII	1 – binario	
Tipo di accesso	0 – sequenziale	1 – casuale (*random*)	
Permanenza	0 – normale	1 – da eliminare dopo l'uso (*temporary*)	
Accesso esclusivo	0 – libero	≠ 0 – bloccato (*locked*)	

La struttura dei dati all'interno di un file può essere considerata da 3 punti di vista distinti
– Livello utente
    • Il programma applicativo associa autonomamente significato al contenuto grezzo del file

– Livello di struttura logica
  • A monte dell'interpretazione dell'utente il S/O organizza i dati grezzi in strutture logiche per facilitarne il trattamento
– Livello di struttura fisica
  • Il S/O mappa le strutture logiche sulle strutture fisiche della memoria secondaria disponibile (p.es.: settori o blocchi su disco)



• Le possibili strutture logiche di un file sono:
– A sequenza di byte
– A record di lunghezza e struttura interna fissa
– A record di lunghezza e struttura interna variabile

Sequenza di byte (byte stream)
– La strutturazione logica più rudimentale e flessibile
  • La scelta di UNIX (→ GNU/Linux) e MS Windows
– Il programma applicativo sa come dare significato al contenuto informativo del file
  • Minimo sforzo per il S/O
– L'accesso ai dati utilizza un puntatore relativo all'inizio del file
– Lettura e scrittura operano a blocchi di byte

Record di lunghezza e struttura fissa
– Gli spazi non utilizzati sono riempiti da caratteri speciali (p.es.: NULL o SPACE)
– Il S/O deve conoscere la struttura interna del file
• Per muoversi al suo interno
– L'accesso ai dati è sequenziale e utilizza un puntatore al record corrente
– Lettura e scrittura operano su record singoli
– Scelta ormai obsoleta e legata a specifiche limitazioni dell'architettura di sistema

Accesso sequenziale
– Viene trattato un gruppo di byte (oppure un record) alla volta
– Un puntatore indirizza il record (o gruppo) corrente e avanza a ogni lettura o scrittura
• La lettura può avvenire in qualunque posizione del file, la quale però deve essere raggiunta sequenzialmente
– Come su un nastro
  • La scrittura può avvenire solo in coda al file (Append)
– Sul file si può operare solo sequenzialmente
  • Ogni nuova operazione fa ripartire il puntatore dall'inizio

Accesso diretto
– Opera su record di dati posti in posizione arbitraria nel file
  • Posizione determinata rispetto alla base (offset = 0)
• Accesso indicizzato
– Per ogni file una tabella di chiavi ordinate contenenti gli offset dei rispettivi record nel file
  • Informazione di navigazione non più nei record ma in una struttura ad accesso veloce (hash)
– Ricerca binaria della chiave e poi accesso diretto
– Denominato ISAM (indexed sequential access method) da IBM
  • Consente accesso sia indicizzato che sequenziale

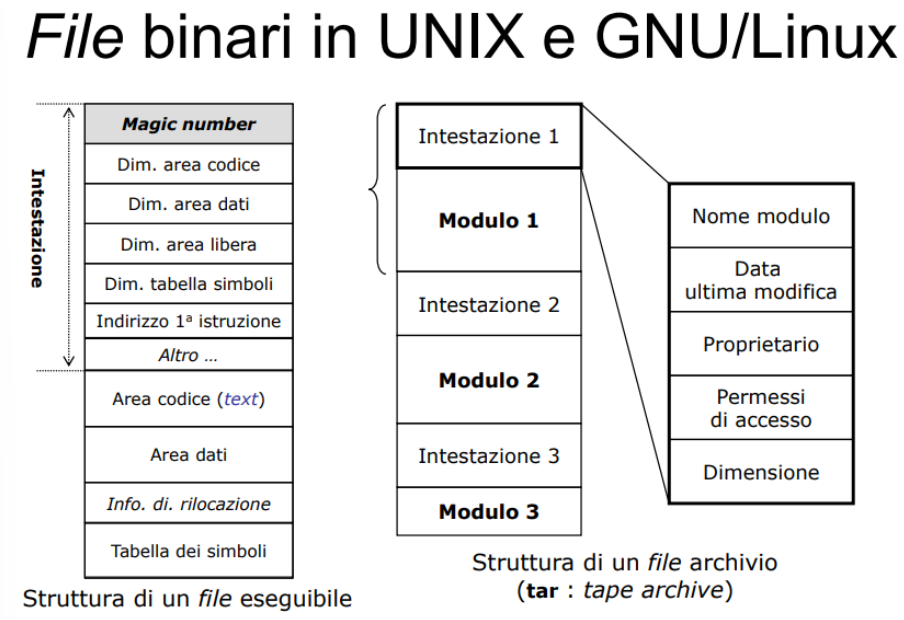
Il FS può trattare diversi tipi di file, con classificazione distinta da quella dell'utente
– File regolari (regular)
  • Sui quali l'utente può operare normalmente (contenuto ASCII (testo) o binario (eseguibile))
– File catalogo (directory)

Scritto da Gabriel

• Tramite i quali il FS permette di descrivere l'organizzazione di gruppi di file

– File speciali

• Con i quali il FS rappresenta logicamente dispositivi orientati a carattere (p.es.: terminale) o a blocco (p.es.: disco)



Le operazioni ammesse sono:

• Creazione

– File inizialmente vuoto; inizializzazione attributi

• Apertura

– Deve precedere il 1° uso; predispone le informazioni utili all'accesso

• Cerca posizione (seek)

– Solo per accesso casuale

• Cambia nome

– Rename (può implicare spostamento nella struttura logica del FS)

• Distruzione

– Rilascio della memoria occupata

• Chiusura

– Rilascio delle strutture di controllo usate per l'accesso ed il salvataggio dei dati

• Lettura. Scrittura

– Read, write, append

• Trova attributi (per make)

• Modifica attributi (permessi)

• Sessione d'uso di un file

– Si può accedere in uso solo a un file già aperto

– All'apertura del file il S/O ne predispone uno specifico strumento di accesso (handle)

– Dopo l'uso il file dovrà essere chiuso

– UNIX (→ GNU/Linux) mantiene una tabella dei file aperti a due livelli

• Livello I: informazioni sul file comuni a "famiglie" di processi

• Livello II: dati specifici del particolare processo

Esempio d'uso con "chiamate di sistema"

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc,
         char *argv[]){

  FILE *fp;
  char dato;

  if (argc != 2) {
    printf("Nome del file?");
    exit(1);
  }

  // continua …
```

1/2

```c
  if ((fp = fopen(argv[1],"w"))
      == NULL){
   printf("File non aperto.\n");
   exit(1);
  }
  do {
   dato = getchar();
   if (EOF == putc(dato, fp)) {
    printf("Errore di lettura.\n");
   break;
   };
 } while (dato != 'c');
 fclose(fp);
}
```

2/2

Il S/O può mappare un file in memoria virtuale
- – Il file continua a risiedere in memoria secondaria
- – All'indirizzo di ogni suo dato corrisponde un indirizzo di memoria virtuale (base + offset)

• Con segmentazione si potrebbe avere (file = segmento) potendo così usare lo stesso offset per entrambi
- – Le operazioni su file avvengono in memoria principale

• Chiamata di indirizzo → page fault → caricamento → operazione → rimpiazzo di pagina → salvataggio in memoria secondaria
- – A fine sessione tutte le modifiche effettuate in memoria primaria devono essere riportate in memoria secondaria

• Riduce gli accessi a disco ma comporta problemi con la condivisione e con i file di enorme dimensione
- – Dove trovare la versione corrente dei dati: RAM o disco?

Ogni FS usa directory (catalogo) o folder(cartella) per tener traccia dei suoi file regolari.
Le *directory* possono essere classificate rispetto all'organizzazione di file che esse consentono
- – A livello singolo
- – A due livelli
- – Ad albero
- – A grafo aperto
- – A grafo generalizzato (con cicli)

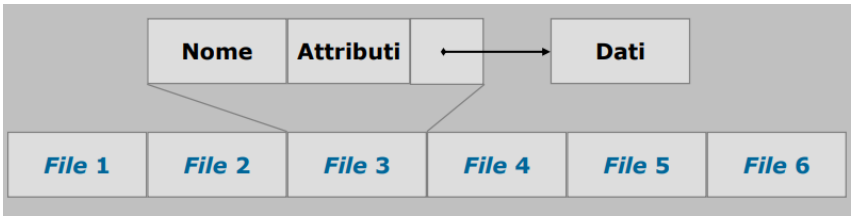
Requisiti fondamentali a livello utente
– Efficienza
- • Realizzare un file deve essere semplice
- • Trovare un file deve essere facile e veloce

– Libertà di denominazione
- • Più utenti devono poter ciascuno usare lo stesso nome per un file loro proprio
- • Lo stesso file deve poter essere "chiamato" con nomi diversi da utenti diversi

– Libertà di raggruppamento
- • Creare gruppi logici di file sulla base di proprietà significative per l'utente

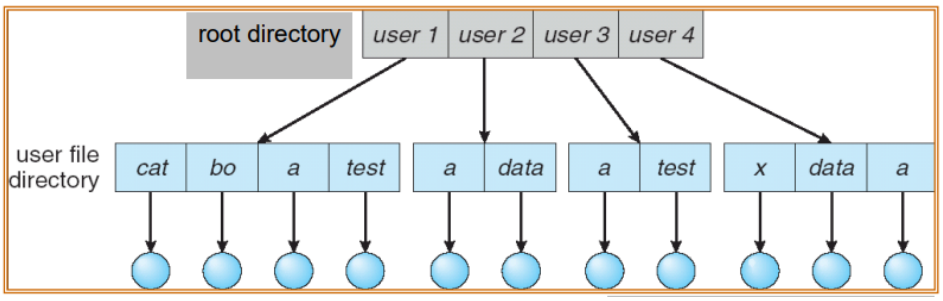
Directory a livello singolo

– Tutti i file sono elencati su un'unica lista lineare ("root directory" ?), ciascuno con il proprio nome
• I nomi dei file devono pertanto essere unici
– Semplice da capire e da realizzare
– File facili da trovare
– Gestione onerosa all'aumentare dei file



Directory a due livelli

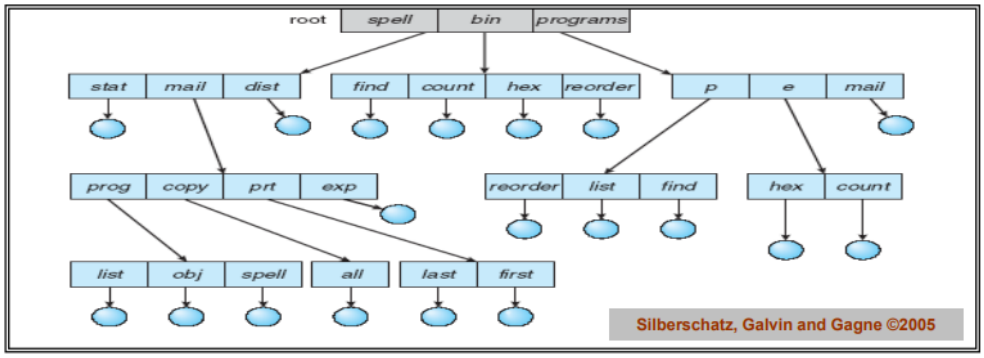
– Una Root Directory contiene una User File Directory (UFD) per ciascun utente di sistema
– L'utente registrato può vedere solo la propria UFD
  • Le UFD di altri solo se esplicitamente autorizzato
  • Buona soluzione per isolare utenti in sistemi multiprogrammati
– I file sono localizzati tramite percorso (path name)
– I programmi di sistema possono essere copiati su tutte le UFD, oppure (meglio) posti in una directory di sistema condivisa e lì localizzati mediante cammini di ricerca predefiniti (search path)



Requisiti parzialmente soddisfatti
– Efficienza di ricerca
– Libertà di denominazione, ma non di riferimenti multipli allo stesso file
Requisiti non soddisfatti
– Libertà di raggruppamento

Directory ad albero

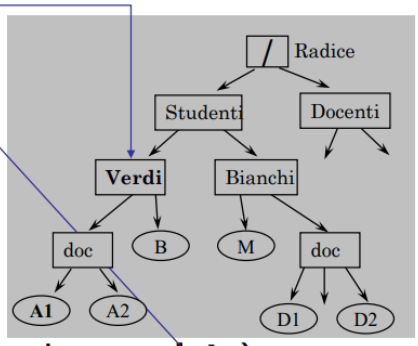
– Numero arbitrario di livelli
– Il livello superiore viene detto radice (root)
– Ogni directory può contenere file regolari o directory di livello inferiore
– Ogni utente ha la sua directory corrente che può cambiare con comandi di sistema
– Se non si specifica il cammino (path) si assume come riferimento la directory corrente
– Il cammino può essere assoluto
• Espresso rispetto alla radice del FS
– Oppure relativo
• Rispetto alla posizione corrente



Requisiti parzialmente soddisfatti
– Ricerca efficiente – Libertà di denominazione
• Ma non di riferimenti multipli allo stesso file
– Libertà di raggruppamento

Esempio di *directory* ad albero
(/ per UNIX/GNU/Linux, \ per MS Windows)

Livello corrente: *directory* **Verdi** = . (*dot*)
Livello superiore (*directory* padre) = ..
Livello inferiore (*directory* figlio) = ./ (*slash*)
Il *file* **A1** identificato come
`[./]doc/A1` (cammino **relativo**)
`/studenti/Verdi/doc/A1` (cammino **assoluto**)
Il *file* **D1** di un altro ramo (purché condiviso)
`../Bianchi/doc/D1` (relativo)
`/studenti/Bianchi/doc/D1` (assoluto)

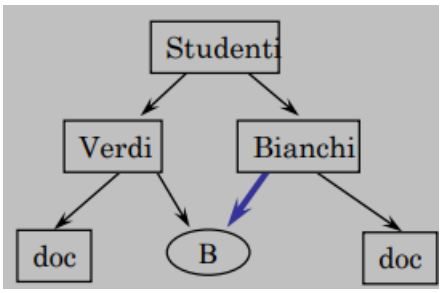


Directory a grafo

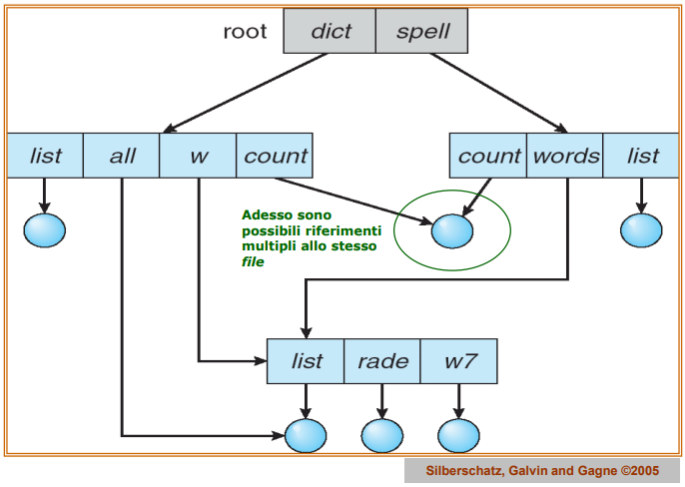
- Aciclico oppure ciclico (generalizzato)

– L'albero diventa grafo consentendo allo stesso file di appartenere simultaneamente a più directory
– UNIX e GNU/Linux utilizzano collegamenti simbolici (link) tra il nome reale del file e la sua presenza virtuale
– La forma generalizzata consente collegamenti ciclici e dunque riferimenti circolari
– Un S/O potrebbe duplicare gli identificatori di accesso al file (handle) → nomi distinti

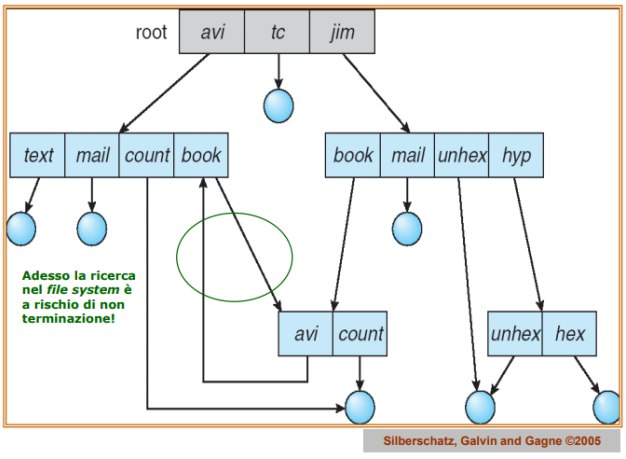
- Questo però rende più difficile assicurare la coerenza del file



Struttura a grafo aciclico



Struttura a grafo generalizzato



Hard link
– Un puntatore diretto a un file regolare viene inserito in una directory a esso remota
    • Che deve risiedere nello stesso FS del file
– Questo crea 2 vie d'accesso distinte dirette a uno stesso file

Symbolic (soft) link
– Viene creato un file speciale il cui contenuto è il cammino del file originario
    • Il file originario può avere qualunque tipo e risiedere anche in un FS remoto
– Questo riferimento mantiene 1 sola via d'accesso al file originario

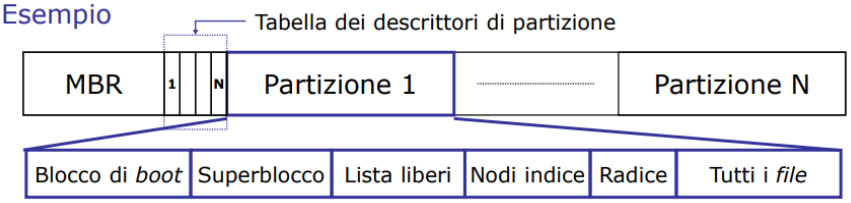
Operazioni su *directory* GNU/Linux

Azione	Nome comando	Chiamata di sistema
Crea *directory*	`mkdir`	Create
Cancella *directory*	`rmdir`	Delete
Cambia nome a *directory*	`mv`	Rename
Apri, chiudi, leggi *directory*		Opendir, Closedir, Readdir
Crea collegamento a *file*	`ln`	Link
Rimuovi collegamento a *file*	`rm`	Unlink

I file system (FS) sono memorizzati su disco
    – I dischi possono essere partizionati
    – Ogni partizione può contenere un FS distinto
• Il settore 0 del disco contiene le informazioni di inizializzazione del sistema
    – Master Boot Record
    – L'inizializzazione è eseguita dal BIOS
    – L'MBR contiene (in 512 B) una descrizione delle partizioni che identifica quella attiva
    – Il primo blocco di informazione di ogni partizione contiene le sue specifiche informazioni di inizializzazione (boot block)

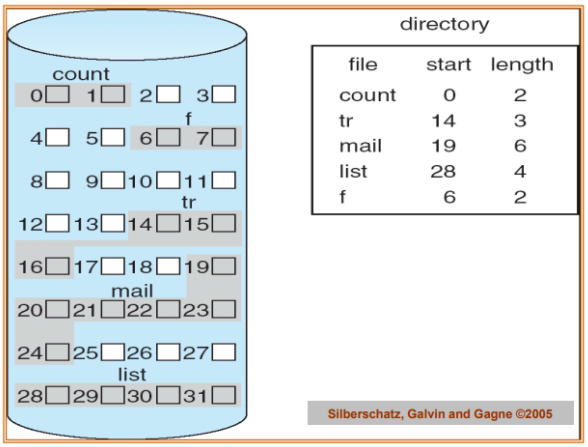
L'unità informativa su disco è il *settore*
• I dischi vengono però letti e scritti a blocchi (cluster per Microsoft)
– 1 blocco = N settori (N ≥ 1)
– Rischio consapevole di frammentazione interna
• La struttura interna di partizione è specifica del FS

Esempio

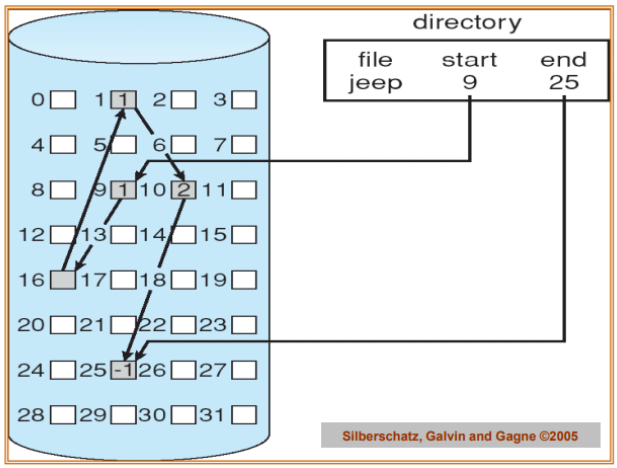


A livello fisico un file è un insieme di blocchi di disco ed occorre decidere quali blocchi assegnare a quale file e come tenerne traccia. 3 strategie di allocazione di blocchi a file
– Allocazione contigua
– Allocazione a lista concatenata (linked list)
– Allocazione a lista indicizzata

Allocazione contigua
– Cerca di memorizzare i file su blocchi consecutivi
– Ogni file è descritto dall'indirizzo del suo primo blocco e dal numero di blocchi utilizzati
– Consente sia accesso sequenziale che diretto
– Un file può essere letto e scritto con un solo accesso al disco
    • Ideale per CD-ROM e DVD
– Ogni modifica di file comporta il rischio di frammentazione esterna
    • Ricompattazione periodica molto costosa
    • L'alternativa richiede l'utilizzo dei gruppi di blocchi liberi
        – Mantenere la lista dei blocchi liberi e la loro dimensione
            » Possibile ma oneroso
– Conoscere in anticipo la dimensione massima dei nuovi file per farli stare in un blocco libero
    » Stima difficile e rischiosa



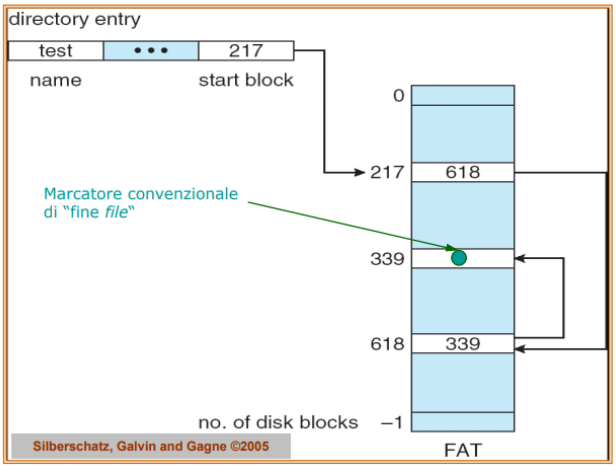
Allocazione a lista concatenata
– File come lista concatenata di blocchi
– File identificato dal puntatore al suo primo blocco
    • Per alcuni S/O anche dal puntatore all'ultimo blocco del file
– Ciascun blocco di file deve contenere il puntatore al blocco successivo (o un marcatore di fine lista)
    • Questo sottrae spazio ai dati
– L'accesso sequenziale resta semplice ma può richiedere molte operazioni su disco
    • Accesso diretto molto più complesso e oneroso (lento)
– Un solo blocco guasto corrompe l'intero file



Allocazione a lista indicizzata
– Si pongono i puntatori ai blocchi in strutture apposite
    • Ciascun blocco contiene solo dati
– Il file è descritto dall'insieme dei suoi puntatori
– 2 strategie di organizzazione
    • Forma tabulare (FAT, File Allocation Table)
    • Forma indicizzata (nodo indice, i-node)
– Non causa frammentazione esterna
– Consente accesso sequenziale e diretto
– Non richiede di conoscere preventivamente la dimensione massima di ogni nuovo file

• File Allocation Table
    – La scelta progettuale di MS-DOS, base di MS Windows
• FAT = tabella ordinata di puntatori
    – Un puntatore ∀ blocco (cluster) del disco
    – La tabella cresce con l'ampiezza della partizione
• La porzione di FAT relativa ai file in uso deve sempre risiedere interamente in RAM
    – Accesso diretto ai dati seguendo sequenzialmente i collegamenti ma senza accessi a disco
    – Es. disco da 200 GB, blocchi da 1 KB, serve FAT di 200 M righe ciascuna di 3-4 Bytes: 6-800 MB di memoria impiegati!
• Un file è una catena di indici, infatti.

Struttura **FAT**



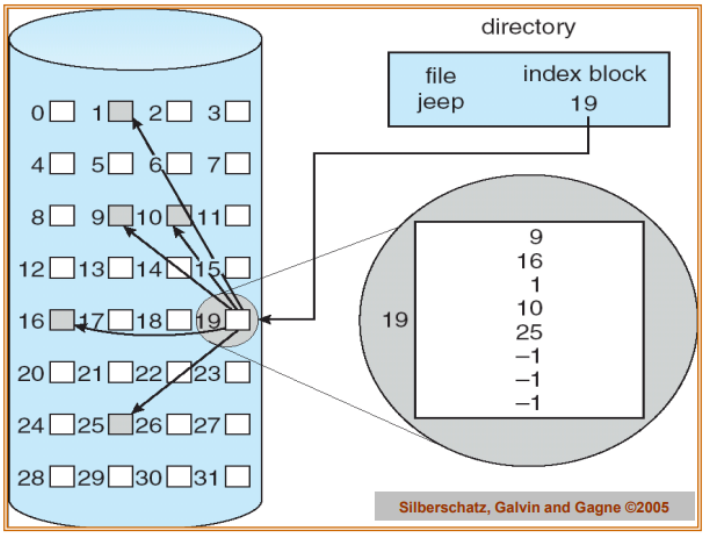
Nodi indice (UNIX → GNU/Linux)
– Una struttura indice (i-node) ∀ file con gli attributi del file e i puntatori ai suoi blocchi
    • L'i-node è contenuto in un blocco dedicato
– In RAM una tabella di i-node per i soli file in uso
    • La dimensione massima di tabella dipende dal massimo numeri di file apribili simultaneamente
– Non più dalla capacità della partizione
– Un *i-node* contiene un numero limitato di puntatori a blocchi

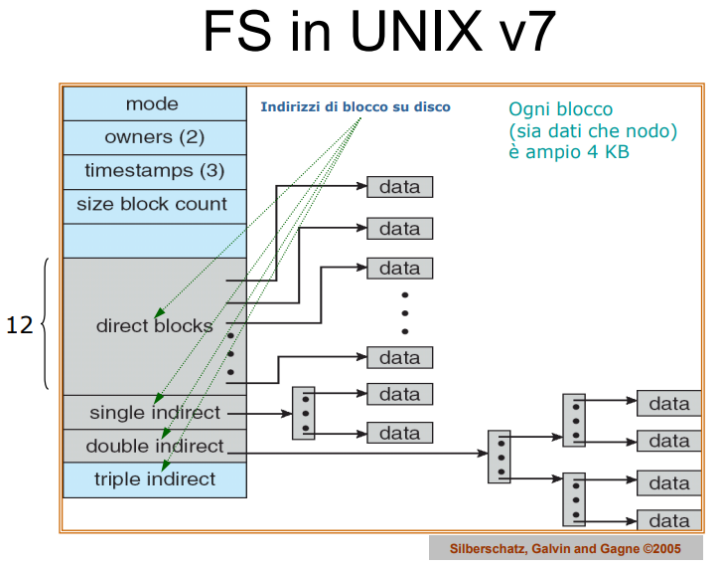
Quale soluzione per file composti da un numero maggiore di blocchi?
– File di piccola dimensione
- Gli indirizzi dei blocchi dei dati sono ampiamente contenuti in un singolo i-node
  – Tipicamente con un po' di frammentazione interna

– File di media dimensione
- Un campo dell'i-node punta a un nuovo blocco i-node

– File di grandi dimensioni
- Un campo dell'i-node principale punta a un livello di blocchi inode intermedi che a loro volta puntano ai blocchi dei dati

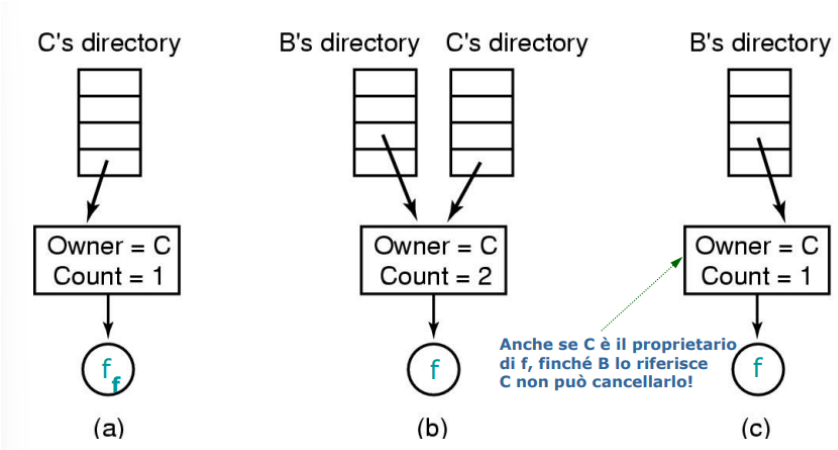
– Per file di dimensioni ancora maggior basta aggiungere un ulteriore livello di indirezione





Gestione dei file condivisi
– Come preservarne la consistenza senza costi eccessivi
- Non porre blocchi di dati nella directory di residenza del file
- ∀ file condiviso porre nella directory remota un symbolic link verso il file originale
  – Esiste così 1 solo descrittore (i-node) del file originale
  – L'accesso condiviso avviene tramite cammino sul FS

• Altrimenti si può porre nella directory remota il puntatore diretto (hard link) al descrittore (i-node) del file originale
  – Più possessori di descrittori dello stesso file condiviso
  – Un solo proprietario effettivo del file condiviso
  – Il file condiviso non può più essere distrutto fin quando esistano suoi descrittori remoti anche se il suo proprietario avesse inteso cancellarlo



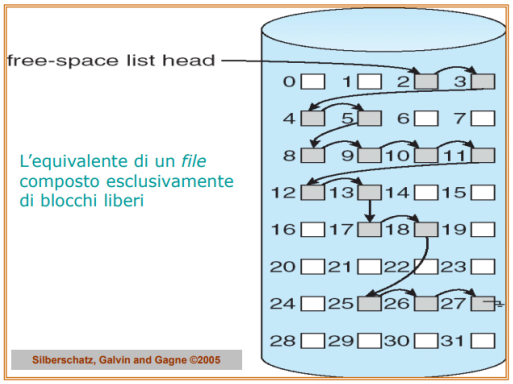
Scritto da Gabriel

Lista concatenata dei blocchi liberi

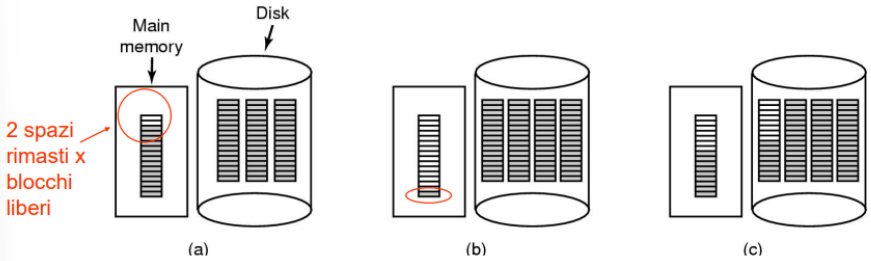
Gestione dei blocchi liberi

– Vettore di bit (bitmap) dove ogni bit indica lo stato del corrispondente blocco
- 0 = libero
- 1 = occupato

– Lista concatenata di blocchi sfruttando i campi puntatore al successivo
- Questa è la scelta nell'architettura FAT



Gestione Spazio su Disco



(a) In RAM: Blocco di puntatori a blocchi liberi su disco
- Altri su disco (non c'è bisogno di averli sempre tutti in RAM)

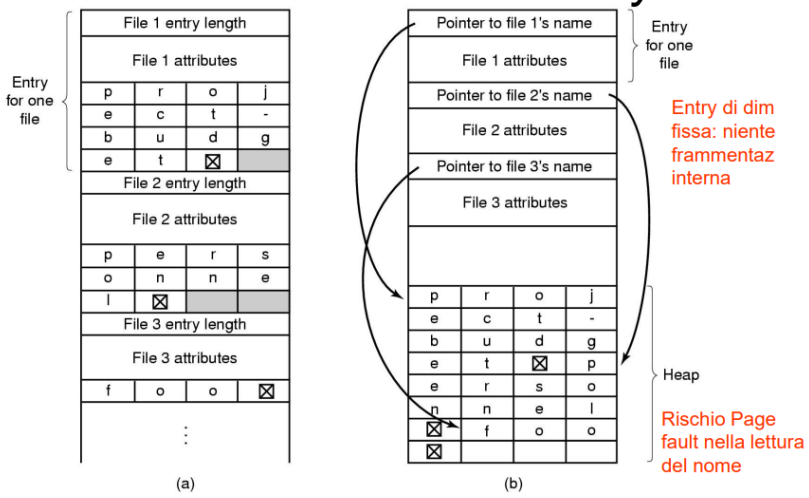
(b) Dopo aver cancellato un file di 3 blocchi
- ma se poi riscrivo 3 blocchi? (devo ricaricare blocco di prima…)

(c) Strategia alternativa per la gestione dei blocchi
- Una volta riempito il blocco, si dividono i puntatori in due parti: metà in blocco in RAM e metà in blocco su disco
- Riempito quello in RAM, si fa scambio

Realizzazione delle *directory* – 2

La directory fornisce informazioni su
– Nome
– Collocazione
– Attributi
• Di file appartenenti a quel particolare catalogo
• File e directory risiedono in aree logiche distinte
• Conviene minimizzare la complessità gestionale della struttura interna di directory
– Meglio una struttura a lunghezza fissa
- Per quanto il suo contenuto sia di ampiezza intrinsecamente variabile

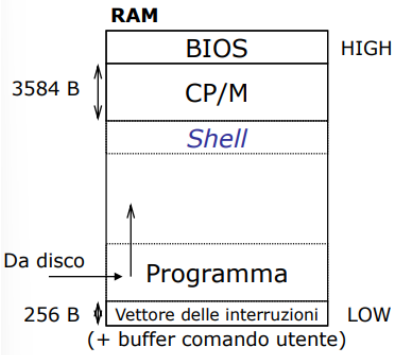
– [Nome + attributi] oppure
– [Nome + puntatore a nodo indice con attributi]



- Frammentazione interna trascurabile per nomi di file fino a 8 caratteri + 3 di estensione
- Il problema diventa però più serio per nomi lunghi

Diamo una prospettiva storica, con:
• CP/M (1973-1981)

CP/M (*Control Program for Microcomputers*)



- BIOS minimo
  - 17 I/O calls (massima portabilità)
- Sistema multiprogrammato
  - Ogni utente vede solo i propri *file*
- *Directory* singola con dati a struttura fissa (32 B entry)
  - In RAM solo quando serve
- *Bitmap* in RAM per blocchi di disco liberi
  - Distrutta a fine esecuzione
- Nome *file* limitato a 8 + 3 caratteri
  - Dimensione inizialmente limitata a 16 blocchi da 1 KB
    - Puntati da *directory*

• MS-DOS & Windows 95 (1981 → 1997)

MS-DOS (*Microsoft Disk Operating System*)

- **Non** multiprogrammato
  - Ogni utente vede **tutto** il FS
- FS **gerarchico** senza limite di profondità e **senza condivisione**
  - Fino a 4 partizioni per disco (C: D: E: F:)
- *Directory* a lunghezza variabile con *entry* di 32 B
  - Nomi di *file* a 8+3 caratteri (normalizzati a maiuscolo)
- Allocazione *file* a lista (FAT)
  - **FAT-X** per **X** = numero di *bit* per indirizzo di blocco ($12 \leq$ **X** <32)
  - Blocchi/Cluster di dimensione multipla di 512 B;
    - Max partition size è 2^{12} x 512B = 2MB
    - Estendendo blocchi fino a 4KB si arriva a 16 MB max
  - **FAT-16** : *File* e partizione limitati a 2 GB
    - 2^{16} = 64K (puntatori a) blocchi di 32 KB ciascuno = 2 GB
  - **FAT-32** : blocchi da 4 ÷ 32 KB e indirizzi da 28 *bit* (!)
    - Perché 2 TB è il limite **intrinseco** di capacità per partizione Win95
      - 2^{32} settori (*cluster*) da 512 B = $2^2 \times 2^{30} \times 2^9$ B = 2^{41} B = 2 TB
      - 2^{28} blocchi da **8** KB = $2^8 \times 2^{20} \times 2^3 \times 2^{10}$ B = 2^{41} B = 2 TB

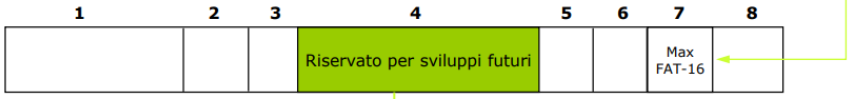
Il FS in MS-DOS 7.0 – 1

Struttura di *directory entry* (32 B)

1. Nome *file*	8 B	5. Ora modifica	2 B
2. Estensione *file*	3 B	6. Data modifica	2 B
3. Attributi	1 B	7. Indice I blocco	2 B
4. Riservati	10 B	8. Dimensione	4 B

(unsigned)
5 *bit* × ore [0-23]
6 *bit* × minuti [0-59]
5 *bit* × ~2 secondi [0-29]

(unsigned)
7 *bit* × anno+1980 [-2107]
4 *bit* × mese [1-12]
5 *bit* × giorno [1-31]



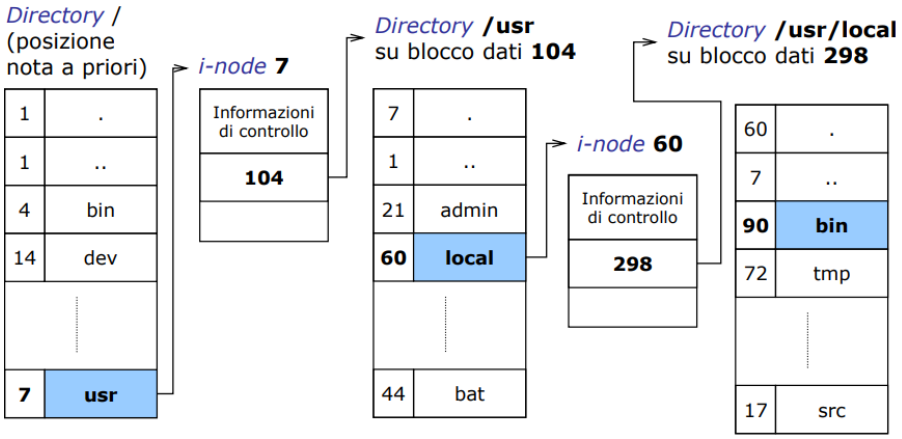
Usato per Windows 98 (FAT-32, orario accurato, nomi *file* lunghi e *case sensitive*)

• Windows 98 (1998-1999)

• UNIX v7 (1979)

Concepito e realizzato tra il 1969 e il 1979 da Ken Thompson e Dennis Ritchie

- Struttura ad albero con radice e condivisione di *file*
  - Grafo **aciclico**
- Nomi di *file* fino a 14 caratteri ASCII (escluso / e NUL)
- *Directory* contiene nome *file* e puntatore (su 2 B) al suo *i-node* descrittore
  - Max 64 K *file* per FS (2^{16} *i-node* distinti)
- L'*i-node* (64 B) contiene gli attributi del *file*
  - Incluso il contatore di *directory* che puntano al *file* tramite un *link* di tipo *hard*
    - Se contatore = 0, il nodo e i blocchi del *file* diventano liberi



Esecuzione parziale del comando "cd /usr/local/bin/"

In merito invece all'*integrità del File System*, si hanno varie tecniche:

Gestione dei blocchi danneggiati
– Via hardware
    • Creando e mantenendo in un settore del disco un elenco di blocchi danneggiati e dei loro sostituti
– Via software
    • Ricorrendo a un falso file che occupi tutti i blocchi danneggiati

Salvataggio del FS
– Su nastro
    • Tempi lunghi, anche per incrementi
– Su disco
    • Con partizione di back-up
    • Oppure mediante architettura RAID, che è una tecnica che utilizza una combinazione di più dischi invece di un singolo disco per aumentare le prestazioni, la ridondanza dei dati o entrambi.

Consistenza del FS
– Un file viene aperto, modificato e poi salvato
– Se il sistema cade tra la modifica e il salvataggio il contenuto del file su disco diventa inconsistente

Consistenza dei blocchi
– 2 liste di blocchi con un contatore ∀ blocco
    • Lista dei blocchi in uso dei file
    • Lista dei blocchi liberi
– Consistenza
    • Ciascun blocco appartiene a una e una sola lista
– Perdita
    • Un blocco non appartiene ad alcuna lista
– Duplicazione
    • Il contatore del blocco è >1 in una delle due liste

Una porzione di memoria principale viene usata come cache di (alcune migliaia di) blocchi
    – Per ridurre la frequenza di accesso ai dischi
    – L'accesso ai blocchi localizzati in "cache" avviene tramite ricerca hash
    – La gestione richiede specifica politica di rimpiazzo blocchi
• Occorre però garantire la consistenza dei dati su disco
    – MS-DOS
        • I blocchi modificati vengono copiati immediatamente su disco (Write through)
    • Alto costo ma consistenza sicura (specie con dischi rimovibili)
– UNIX → GNU/Linux
    • Un processo periodico (sync) effettua l'aggiornamento dei blocchi su disco
    • Basso costo e basso rischio con dischi fissi affidabili

Esercizio Blocchi Liberi - Bitmap

Quesito 3. Un *file system* utilizza un vettore di *bit* (*bitmap*), con posizioni numerate da sinistra verso destra, per indicare i blocchi liberi presenti in una data partizione di disco. La formattazione della partizione libera tutti i blocchi tranne il 1°, che viene assegnato alla *directory* radice. In tale sistema, l'assegnazione di blocchi liberi a *file* considera sempre prima i blocchi di indice minore, trascurando ogni considerazione di contiguità. Si mostri il contenuto della parte iniziale di tale vettore dopo ciascuna delle seguenti operazioni:

- Scrittura del *file* A, ampio 6 blocchi
- Scrittura del *file* B, ampio 5 blocchi
- Rimozione del *file* A
- Scrittura del *file* C, ampio 8 blocchi
- Rimozione del *file* B

Si mostri poi l'evoluzione del contenuto di tale vettore a fronte della medesima sequenza di operazioni nel caso in cui il sistema volesse assicurare la massima contiguità dei blocchi assegnati ad ogni *file*.

Soluzione

Soluzione 3 (punti 5). Seguiamo l'evoluzione del contenuto della maschera nel primo caso, concentrandoci sulle sue prime posizioni:

Dopo la formattazione:	100000000000000 …
Scrittura del *file* A:	111111100000000 …
Scrittura del *file* B:	111111111111000 …
Rimozione del *file* A:	100000011111000 …
Scrittura del *file* C:	111111111111110 …
Rimozione del *file* B:	111111100000110 …

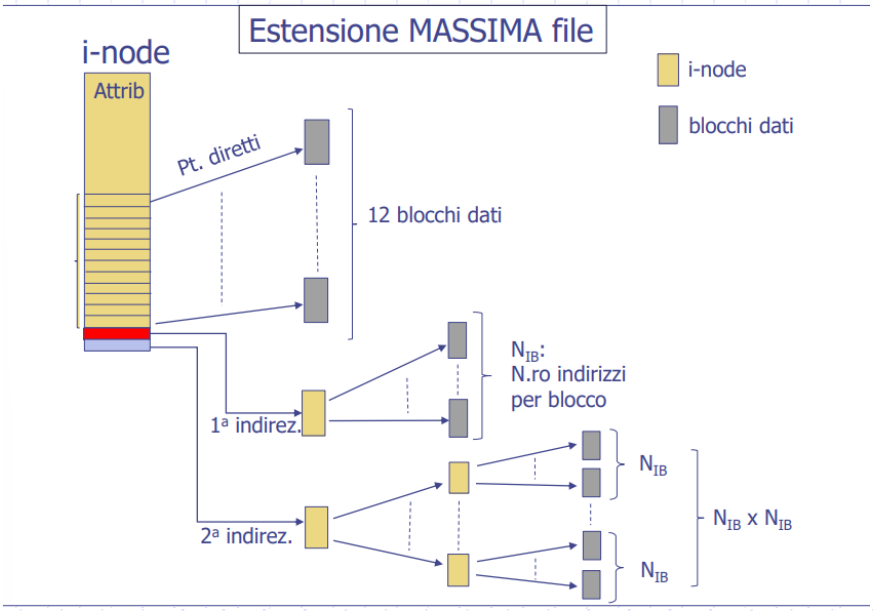
Vediamone ora l'evoluzione nel secondo caso, nel quale prevalgono considerazioni di contiguità dei blocchi assegnati ai *file*:

Dopo la formattazione:	100000000000000 …	
Scrittura del *file* A:	111111100000000 …	allocazione contigua senza frammentazione esterna
Scrittura del *file* B:	111111111111000 …	allocazione contigua senza frammentazione esterna
Rimozione del *file* A:	100000011111000 …	
Scrittura del *file* C:	10000011111111111110 …	allocazione contigua con frammentazione esterna
Rimozione del *file* B:	10000000000111111110 …	

Esercizio i-node

Quesito 4. Il progettista di un sistema operativo ha deciso di usare nodi indice (*i-node*) per la realizzazione del proprio *file system*, stabilendo che essi abbiano la stessa dimensione di un blocco, fissata a 512 *byte*. Il progettista ha poi deciso che un nodo indice primario contenga 12 campi di indirizzo di blocchi di disco e 2 campi puntatori a nodi indice di primo e secondo livello di indirezione rispettivamente. Sapendo che gli indirizzi di blocco sono espressi su 32 *bit*, si vuole allocare un *file* logicamente composto da 10.000 *record* da 80 *byte* ciascuno, imponendo che un *record* non possa essere suddiviso su due blocchi. Calcolare quanti blocchi verranno utilizzati per allocare il *file* dati e quanti per gestire la sua allocazione tramite nodi indice. Determinare inoltre l'occupazione totale in memoria secondaria risultante da tale strategia di allocazione.

Soluzione



Soluzione 4 (punti 5). Richiamiamo i dati del problema:
N_R = numero di *record* che compongono il *file* = 10.000
D_R = dimensione di un *record* = 80 *byte*
D_I = dimensione di un indirizzo = 4 *byte*
D_B = dimensione di un blocco = 512 *byte*

N_{RB} = numero di *record* per blocco = intD_B/DR) = int512/80) = int6,4) = 6
N_{BF} = numero di blocchi occupati dal *file* = 1+intN_R/N_{RB}) = 1+int10000/6) = 1.667
N_{IB} = numero di indirizzi in un blocco = D_B/D_I = 512/4 = 128
N_{ID} = numero di indirizzi in X blocchi a doppia indirezione = X^2

I blocchi da indirizzare sono N_{BF} = 1.667

- di questi, 12 possono essere indirizzati direttamente dal nodo indice principale
- dei rimanenti 1.667-12 = 1.655, N_{IB} (cioè 128) sono indirizzabili ad indirezione singola tramite l'indirizzo ad indirezione singola del nodo indice principale
- dei rimanenti 1.655-128 = 1.527, si possono utilizzare blocchi indiretti di 128 con indirezione doppia, come mostrato in figura.

1 + int1527/128 = 12 blocchi

Con le indicazioni riportate in figura possiamo concludere che:

- per allocare il *file* dati sono necessari NBF blocchi, cioè 1.667 blocchi
- per gestire l'allocazione del *file* sono necessari:
  - 1 blocco per il nodo indice principale
  - 1 blocco di indirizzi ad indirezione singola
  - 1+12 blocchi per l'indirezione doppia

  per un totale di 1+1+1+12 = 15 blocchi
- l'occupazione totale in memoria secondaria vale 1.667 + 15 = 1.682 blocchi da 512 *byte*, per un totale di 1.682 o 512 = 861.184 = *byte* = 841 kB

Quesito

Si consideri un *file system* residente su una partizione di disco con dimensione dei blocchi logici e fisici di 512 B, dimensione dei *file* non superiori a 512 blocchi, e con tutte le informazioni su ciascun *file* già presenti in memoria principale.
Per ciascuno dei tre metodi di allocazione visti a lezione (**contigua, concatenata, indicizzata**):

1. si illustri come gli indirizzi logici vengono fatti corrispondere agli indirizzi fisici
2. assumendo che l'ultimo accesso sia stato fatto al blocco logico 10, si determini quanti blocchi fisici debbano essere letti dal disco per accedere al blocco logico 4.

Soluzione 1/3

Allocazione contigua: il *file* è denotato dall'indice del primo blocco fisico e dalla sua ampiezza in blocchi; vista la corrispondenza di ampiezza tra blocchi logici e fisici, ogni posizione interna al *file* (blocco logico e *offset* in esso) ha una corrispondenza diretta sul disco (blocco fisico e *offset*).

Allocazione concatenata: il *file* è denotato dagli indici del primo e dell'ultimo blocco fisico; una parte dei dati di ogni blocco contiene il puntatore al blocco successivo. La posizione interna al *file* espressa in (blocco logico *i*, *offset 0*) viene dunque tradotta mediante l'attraversamento di *i* posizioni nella lista concatenata a partire dalla testa.

Soluzione 2/3

Allocazione indicizzata: il *file* è denotato da un blocco speciale (detto appunto "indice"), che contiene gli indici dei blocchi fisici ove risiedono i dati. La posizione interna al *file* espressa in (blocco logico *i*, *offset o*) viene dunque tradotta localizzando il blocco fisico denotato dalla posizione *i* entro il blocco indice e la posizione *o* al suo interno. (Come noto, il blocco indice può essere realizzato come una tabella concatenata, tipo FAT, oppure come un blocco contiguo dedicato, tipo *i-node*.)

Soluzione 3/3

Blocchi fisici acceduti per procedere dal blocco 10 al blocco 4 :

Allocazione contigua: 1 (direttamente il blocco 4).

Allocazione concatenata: 4 (fino al blocco 4 a partire dalla testa della lista).

Allocazione indicizzata: 1 (direttamente il blocco 4, ma solo in virtù dell'ipotesi favorevole del quesito per la quale la dimensione massima del *file* sia interamente rappresentabile con un singolo blocco indice).

UNIX-Linux

Il sistema operativo Unix è un sistema operativo multitasking e multiutente sviluppato alla fine degli anni Sessanta. AT&T Bell Labs è lo sviluppatore del sistema operativo Unix. Originariamente progettato per l'uso esclusivo da parte dei programmatori grazie alla sua flessibilità, potenza e portabilità, Unix è stato ampiamente utilizzato in vari sistemi informatici, tra cui desktop, server e computer portatili.
Questo lo ha reso anche il sistema operativo che è stato maggiormente portato su più famiglie di macchine, con conseguente identificazione con il concetto di sistemi aperti. Unix coordina l'utilizzo delle risorse del computer, stabilendo una sorta di "divisione del lavoro". Ad esempio, mentre una persona esegue un programma di controllo ortografico, un'altra può creare un documento, mentre un'altra persona modifica il documento, un'altra formatta il documento e così via, contemporaneamente, senza che ciascuno debba sapere cosa sta facendo l'altro.

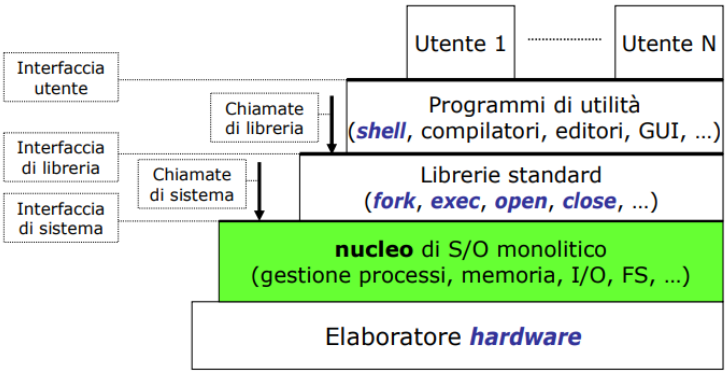
Questo è possibile perché Unix controlla tutti i comandi del computer, dalle tastiere ai dati generati, ecc. e quindi ogni utente crede di essere l'unico a lavorare sul computer in quel momento. Questa capacità è chiamata condivisione in tempo reale e ha probabilmente reso Unix il sistema operativo più potente in assoluto. Unix è stato inizialmente progettato per computer di medie dimensioni, ma sin dalla sua creazione è stato utilizzato in computer mainframe più potenti e più grandi e persino in personal computer.

Scritto da Gabriel

Le caratteristiche principali del sistema operativo Unix sono la portabilità, la flessibilità, il multitasking e la multiutenza. È inoltre dotato di un file system gerarchico e di un'ampia libreria di software.
Ci furono molte versioni di UNIX nel corso del tempo, ma la standardizzazione si ebbe nel 1986 con POSIX (Portable Operating System Interface for UNIX) , che definisce l'insieme standard di procedure di libreria utili per operare su S/O compatibili. La maggior parte contiene chiamate di sistema e ha servizi utilizzabili da linguaggi ad alto livello.

• Scelte architetturali per cloni UNIX
– Micro-kernel : MINIX (Tanenbaum, 1987)
    • Nel nucleo solo processi e comunicazione (affidabile e efficiente)
    • Il resto dei servizi (p.es. : FS) realizzato in processi utente
        – MINIX non copre tutti i servizi UNIX
– Nucleo monolitico : GNU/Linux (Linus Torvalds, da v0.01 nel maggio 1991 a v3.8 di oggi)
• Clone completo aderente a POSIX con qualche libertà
• Modello open-source (scritto nel C compilato da gcc – GNU C compiler

In generale, UNIX è un sistema multi-programmato multi-utente con un'architettura a livelli gerarchici.



UNIX nasce con I/F per linea di comando (shell)
– Più potente e flessibile di GUI ma destinato a utenti esperti
– Una shell per ogni terminale utente (xterm)
    • Lettura dal file "standard input"
    • Scrittura sul file "standard output " o "standard error "
– Inizialmente associati al terminale stesso (visto come file)
• Possono essere re-diretti
    – < per stdin, > per stdout (linguaggio C)
– Caratteri di prompt (%, $) indicano dove editare il comando
– Comandi compositi possono associare uscita di comandi ad ingresso di altri mediante | (pipe) e combinati in sequenze interpretate (script)
– In modalità normale la shell esegue un comando alla volta
– Comandi possono essere inviati all'esecuzione liberando la shell(&, background)

La principale entità attiva nel sistema UNIX è il *processo*
– Inizialmente definito come sequenziale
    • Ossia dotato di un singolo flusso di controllo interno
– Concorrenza a livello di processi
    • Molti processi attivati direttamente dal sistema (daemon)
    • Creazione mediante fork()
        – Clone con stessa memoria all'inizio e accesso a file aperti
    • La discendenza di un processo costituisce un "gruppo"
    • Comunicazione mediante scambio messaggi (pipe) e invio di segnali (signal) entro un gruppo
– Processi figli hanno memoria identica quella del processo genitore solo all'inizio
    • Poi indipendente alla prima modifica

Scritto da Gabriel

Ci sono processi con più flussi di controllo interni, detti *thread.*
La creazione di un thread gli assegna identità, attributi, compito e argomenti e ciascun thread condividono tutte le risorse logiche e fisiche del processo genitore (inclusi valori di variabili in uso).

- Completato il proprio lavoro il thread termina sé stesso volontariamente
  - Invocando la procedura *pthread_exit*
- Un thread può sincronizzarsi con la terminazione di un suo simile
  - Invocando la procedura *pthread_join*
- L'accesso a risorse condivise viene sincronizzato mediante semafori a mutua esclusione
  - Tramite le procedure *pthread_mutex{_init, _destroy}*
- L'attesa su condizioni logiche (p.es. : risorsa libera) avviene mediante variabili speciali simili a condition variables (ma senza monitor)

Le gestione dei processi avviene con una tabella dei processi, che sta permanentemente in RAM e per tutti i processi.

- Parametri di ordinamento (es. priorità, tempo di esecuzione cumulato, tempo di sospensione in corso, …)
- Descrittore della memoria virtuale del processo
- Lista dei segnali significativi e del loro stato
- Stato, identità, relazioni di parentela, gruppo di appartenenza

Codice semplificato di un processo *shell*

```
while (TRUE) {
   type_prompt(); // mostra prompt sullo schermo
   read_command(cmd, par); // legge linea comando
1  pid = fork();
   if (pid < 0) {
      printf("Errore di attivazione processo.\n");
      continue; // ripeti ciclo
   };
   if (pid != 0) {
     waitpid(-1, &status, 0); // attende la terminazione
                              // di qualunque figlio
   } else {
2    execve( cmd, par, 0);
   }
}
```

Codice eseguito dal padre

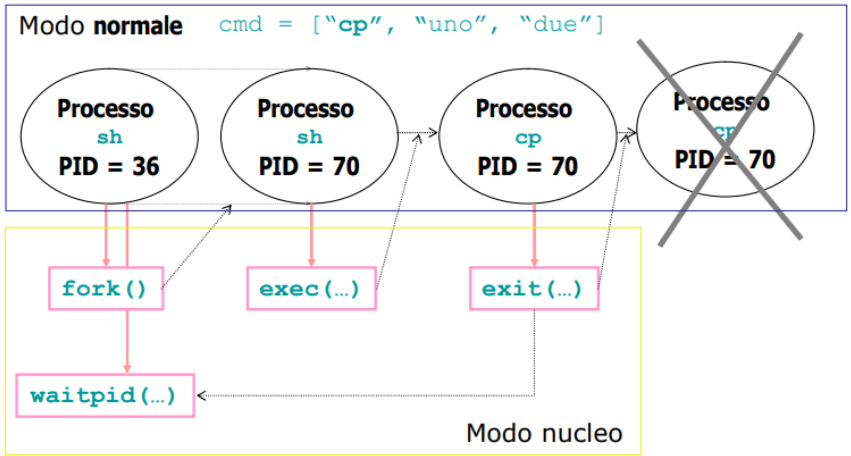
Codice eseguito dal figlio

Esecuzione di comando di *shell* – 2

1. Il processo chiamante passa in **modo nucleo** e prova a inserire i propri dati per il figlio nella **Tabella dei Processi** (incluso il PID). Se riesce, alloca memoria per ***stack*** e dati del figlio. A questo punto il codice del figlio è ancora **lo stesso** del padre.

2. La linea di comando emessa dall'utente viene passata al processo figlio come ***array*** di stringhe. La **exec**, che opera in **modo nucleo**, localizza il programma da eseguire e lo sostituisce al codice del chiamante, passandogli la linea di comando e le "**definizioni di ambiente**" specifiche per il nuovo processo

Esecuzione di comando di *shell* – 3

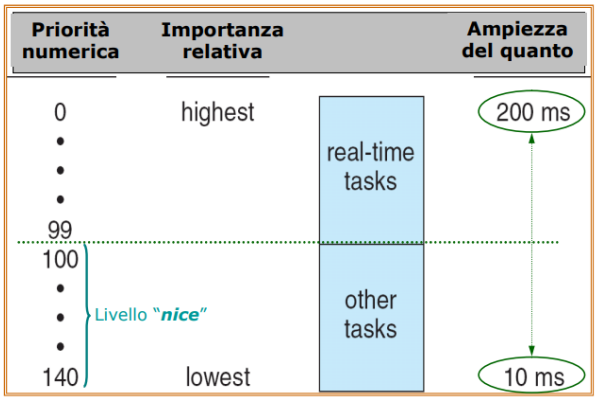


• fork() duplica il processo chiamante creando un processo figlio uguale ma distinto
– Che accade se questi include più thread?
• Vi sono 2 possibilità
1) Tutti i thread del padre vengono clonati
• Difficile gestire il loro accesso concorrente ai dati e alle risorse condivise con thread del padre
2) Solo un thread del padre viene clonato
• Possibile sorgente di inconsistenza rispetto alle esigenze di cooperazione
con le thread non clonate
• Dunque, il multi-threading aggiunge gradi di complessità
– Al FS
• Più difficile assicurare consistenza nell'uso concorrente di file

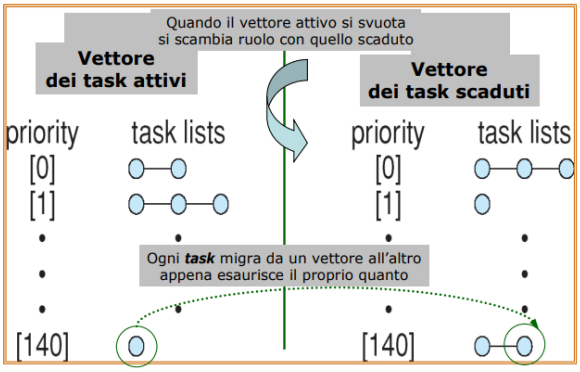
Maggior granularità nel trattamento della condivisione di strutture di controllo nella creazione di processi e thread figli
• Chiamata di sistema alternativa a fork()
pid = clone(function, stack_ptr, ctrl, arg);
– function → programma da eseguire nel nuovo "task" (processo o thread) con argomento par
– Stack_ptr → indirizzo dello stack assegnato al nuovo task
– ctrl → grado di condivisione desiderato tra il nuovo task e l'ambiente del chiamante
• Spazio di memoria, FS, file, segnali, identità
• Es. Solo copia o stesso address space?

• I thread sono gestiti direttamente dal nucleo
– Ordinamento per task (thread o processo indistintamente)
– Selezione distinta tra classi distinte
– Prerilascio per fine quanto o per attesa di evento
• 3 classi di priorità di task
– Tempo reale con politica FCFS a priorità senza prerilascio
• A priorità uguale viene scelto il task in attesa da più tempo
– Tempo reale con politica RR a priorità
• Prerilascio per quanti con ritorno in fondo alla coda
– Divisione di tempo RR a priorità (Timesharing)
• Priorità dinamica con premio o penalità rispetto al grado di interattività con I/O (alta → premio, bassa → penalità)
• Nuovo valore assegnato al task all'esaurimento del suo quanto corrente

Ordinamento dei processi (GNU/Linux) – 2

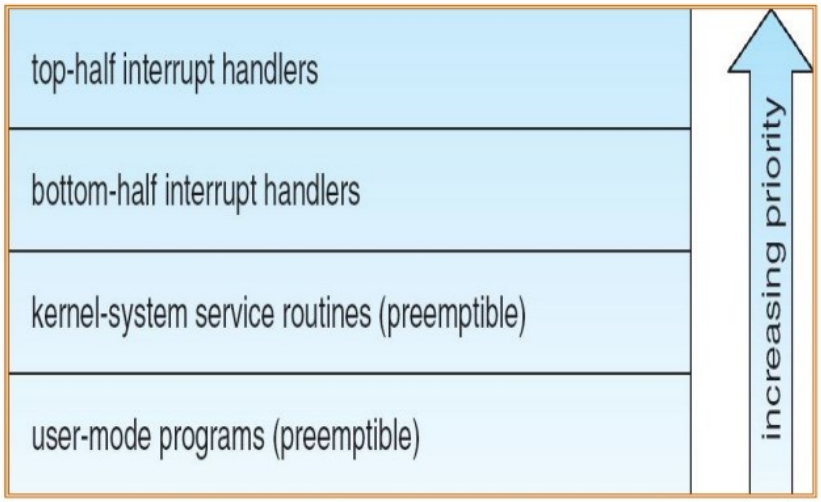


Ordinamento dei processi (GNU/Linux) – 3



Per versione < 2.6 l'attività dei processi in modo nucleo non ammetteva prerilascio
- – Ma questo naturalmente causava pesanti problemi di inversione di priorità

• Con versione ≥ 2.6 si usa granularità più fine
- – Inibizione selettiva di prerilascio
  - • Per sezioni critiche corte
- – Uso di semafori convenzionali
  - • Per sezioni critiche lunghe
- – Uso minimale di disabilitazione delle interruzioni
  - • La parte immediata (top half) dei gestori disabilita
  - • La parte differita (bottom half) non disabilita ma il completamento della sua esecuzione viene preferito a ogni altra
    - – Tranne che di altre parti immediate



L'inizializzazione di GNU/Linux è composta in questo modo:
• BIOS carica l'MBR (Master Boot Record) da primo settore su disco di boot in RAM e lo "esegue"
- – MBR = 1 settore di disco = 512 B

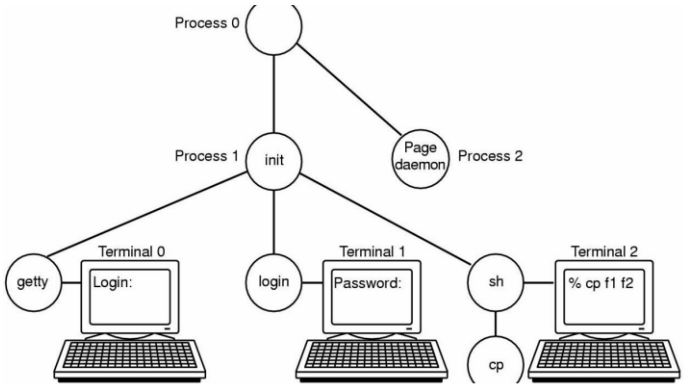
• L'MBR carica il programma di boot dal corrispondente blocco della partizione attiva
- – Lettura della struttura di FS, localizzazione e caricamento del nucleo di S/O

• Il programma di inizializzazione del nucleo è scritto in assembler (specifico per l'elaboratore!)
- – Poche azioni di configurazione di CPU e RAM
- – Il controllo passa poi al main di nucleo
  - • Configurazione del sistema con caricamento dinamico dei gestori dei dispositivi rilevati
  - • Inizializzazione e attivazione del processo 0

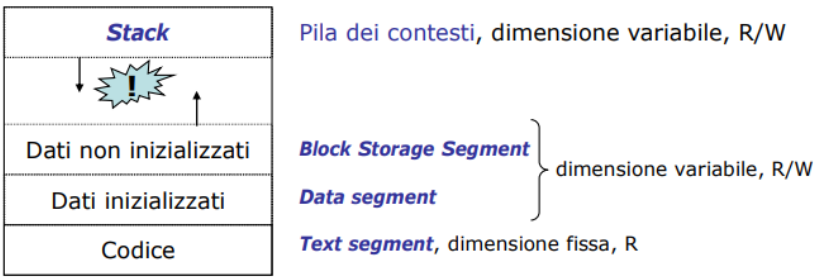
Processo 0
- – Configurazione degli orologi, installazione del FS di sistema, creazione dei processi 1 (`init`) e 2 (*daemon* delle pagine)

Processo 1
- – Configurazione modo utente (singolo, multi)
  - Esecuzione *script* di inizializzazione `shell` (`/etc/rc` etc.)
  - Lettura numero e tipo terminali da `/etc/ttys`
  - `fork()` ∀ terminale abilitato ed `exec("getty")`

Processo `getty`
- – Configurazione del terminale e attivazione del *prompt* di *login*
- – Al *login* di utente, `exec("[/usr]/bin/login")` con verifica della *password* d'accesso (criptate in `/etc/passwd`)
- – Infine `exec("shell")`



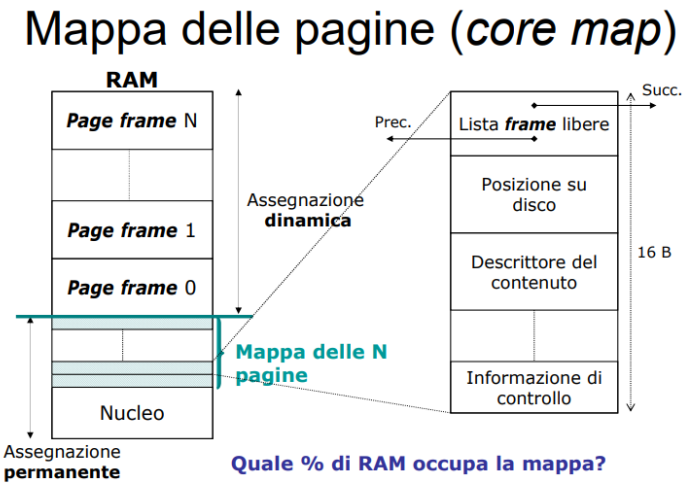
- Ogni processo possiede un proprio spazio di indirizzamento privato (**memoria virtuale**)
  - Suddiviso in 4 sezioni



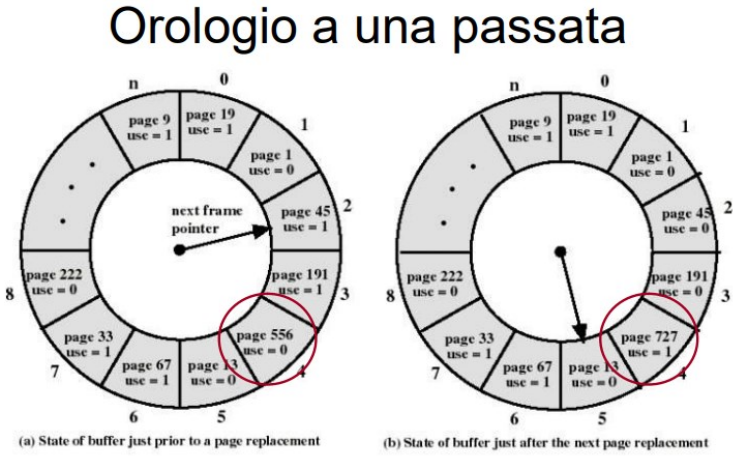
• Il segmento dati varia in dimensione a seconda delle attività del programma (cf. malloc() di C)
  – POSIX non definisce queste chiamate di sistema
  – Una parte del segmento dati può ospitare file mappati in memoria
• Il segmento stack contiene l'ambiente d'esecuzione corrente (record di attivazione) e cresce in direzione opposta al segmento dati
• Il segmento codice può essere condiviso tra più processi
  – Ma non gli altri segmenti tranne per processi duplicati da fork() finché non vengono modificati
• Copy-on-write
  – contiene linguaggio macchina (da compilazione)

In origine l'allocazione di memoria principale
avveniva mediante swap di processi
– Rimpiazzo di interi processi quando una particolare esecuzione rilevava mancanza di memoria
  • A seguito di fork()
  • A causa di allocazione esplicita richiesta dal programma
  • Per allocazione implicita conseguente a chiamata di procedura
– Il gestore (swapper) creava lo spazio necessario salvando su disco i processi sospesi con più tempo d'esecuzione recente e minor priorità

In seguito, fu introdotta paginazione con modalità a richiesta (paging on demand)
– Un processo è eseguibile se il suo descrittore e la sua tabella delle pagine si trovano in RAM
  • Il suo spazio di indirizzamento è caricato da disco per ogni riferimento che richiede dati non presenti in RAM
  • Nessun caricamento anticipato di pagine (no working set)
– Il processo "2" gestisce lo stato dei page frame in RAM
  • Page daemon
  • Tenendo nucleo di S/O e "mappa delle pagine" (core map) sempre in RAM
  • Il resto è paginato e ciascuna page frame indica il proprio uso
    – Codice, dati, stack, tabella delle pagine (altrimenti in lista pagine libere)



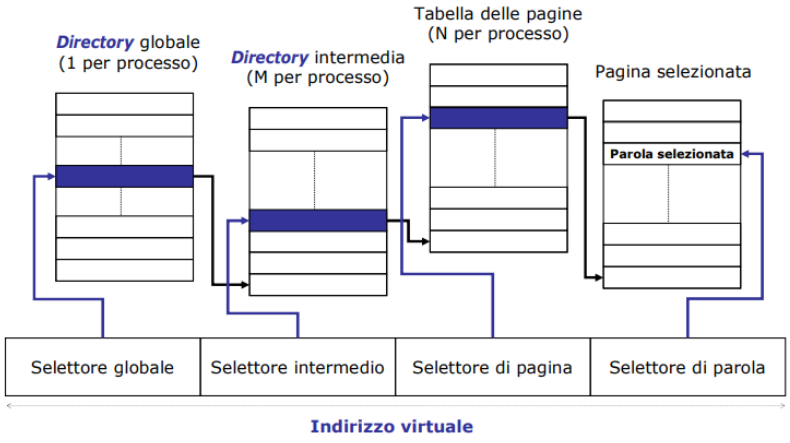
Page daemon verifica con periodico 1/4 s che in RAM vi siano ≥ *lotsfree* pagine libere
– Se ne mancano ne libera quante ne servono salvandone il contenuto corrente su un'area di disco specifica per pagina
– La selezione delle pagine in uscita usa un algoritmo "a doppia passata"
- Two-handed clock algorithm
- Lista circolare delle pagine
- La 1a passata pone a 0 il bit di riferimento
- La 2a passata, a distanza programmabile, rimuove le pagine nel frattempo, non riferite (bit vale 1 altrimenti)



Se vi è spazio libero il page daemon riporta in RAM processi pronti selezionati con una euristica di "valore"
– Caricando solo il descrittore di processo e la sua tabella delle pagine
- Lasciando che il resto sia caricato via paging on demand

Per architetture a 32 bit la memoria virtuale di processo è ampia 4 GB
– 1 GB riservato e invisibile al modo operativo normale (user mode) per la tabella delle pagine del processo e per altri dati di controllo a uso del nucleo
- Spazio suddiviso in regioni = sequenze contigue di pagine
- Ogni regione ha un descrittore noto al nucleo
– Info su contenuto, pagine attive/inattive, aree libere

La fork() di GNU/Linux replica per il figlio l'intera lista di descrittori del padre. Le pagine del figlio sono fisicamente duplicate solo in caso di modifica (copy on write)
– La regione è marcata R/W
– Le sue pagine dati sono inizialmente marcate R
– Ogni richiesta di scrittura causa eccezione così il nucleo duplica la pagina richiesta e marca la copia come R/W



• Il nucleo rimane sempre in RAM
• Ad ogni istante nella RAM rimanente possono trovarsi:
– Le pagine attive dei processi utente
– Un insieme di pagine utente inattive ma presenti
– Una cache di blocchi di file usata dal file system
• Dimensione variabile organizzata per pagine
• *kswapd* è il page daemon (periodo 1 s) che cerca pagine da spostare su disco
• Il daemon *bdflush* gestisce la riscrittura delle pagine modificate

La RAM è allocata in maniera dinamica e variabile e si usa un algoritmo di allocazione primario (buddy). Gli schemi di partizione statici soffrono della limitazione di avere un numero fisso di processi attivi e l'utilizzo dello spazio può anche non essere ottimale. Il sistema buddy è un algoritmo di allocazione e gestione della memoria che gestisce la memoria in incrementi di due. Supponiamo che la dimensione della memoria sia 2U, supponiamo che sia richiesta una dimensione S.

- Se $2^{U-1} < S <= 2^{U}$: Allocare l'intero blocco
- Altrimenti: Dividere ricorsivamente il blocco in parti uguali e testare la condizione ogni volta; quando è soddisfatta, allocare il blocco e uscire dal ciclo.

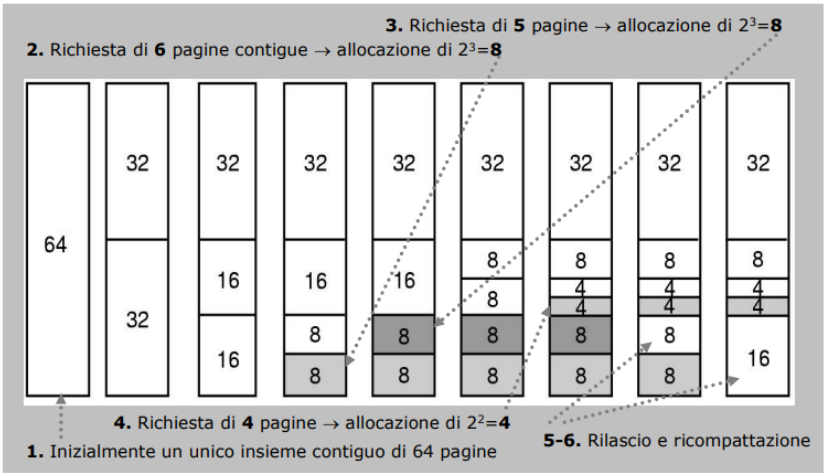
Il sistema tiene anche traccia di tutti i blocchi non allocati e può unire questi blocchi di dimensioni diverse per creare un unico grande blocco.

Vantaggi:

- Facile implementazione del sistema buddy
- Allocazione di blocchi di dimensioni corrette
- È facile unire blocchi adiacenti
- Veloce nell'allocare e deallocare la memoria

Svantaggi:

- Richiede che tutte le unità di allocazione siano potenze di due
- Porta alla frammentazione interna



UNIX tratta i dispositivi di I/O come file di tipo speciale, ciascun con posizione specifica nel FS
– Per esempio /dev/…
• Un gestore (device driver) è associato in modo esclusivo a ciascun dispositivo o a famiglia di dispositivi dello stesso tipo
– Una coppia di indici <maggiore, minore> identifica precisamente ciascun dispositivo di I/O
• Maggiore: tipologia
• Minore: specifico di quel dispositivo

GNU/Linux consente invece caricamento dinamico dei moduli di gestione dei dispositivi
– Soluzione molto preferibile alla configurazione statica che richiede ogni volta una nuova compilazione dell'intero nucleo
• Inevitabile a fronte della grande varietà di hardware attuale
• Il caricamento dinamico richiede al nucleo di effettuare diverse azioni di configurazione
– [1] Rilocazione dello spazio di indirizzamento del modulo
– [1-2] Allocazione delle risorse necessarie
• P. es.: interruzione assegnata al dispositivo
– [2] Configurazione del vettore delle interruzioni
– [2] Attivazione e inizializzazione del gestore

Scritto da Gabriel

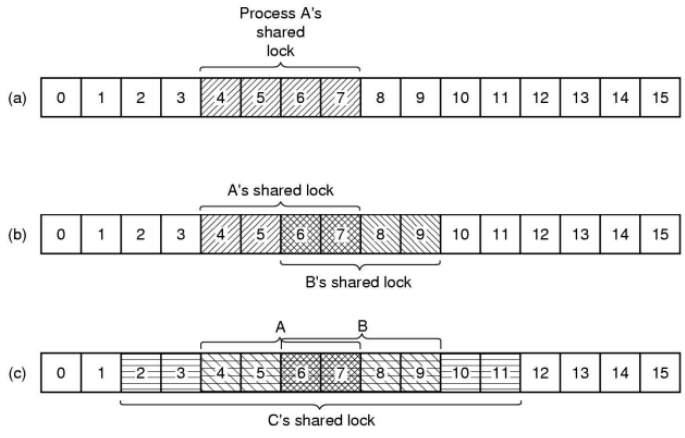
• Un file speciale (detto socket) viene utilizzato per la connessione di rete e i relativi protocolli
  – Può essere creato e distrutto dinamicamente
  – Un socket è associato a uno specifico indirizzo di rete
• Tre tipi di connessione con scelta alla creazione
  – Connessione affidabile a flusso di caratteri (~ TCP)
    • Il gestore garantisce la correttezza della trasmissione
    • Invio e ricezione per blocchi di dimensione variabile
  – Connessione affidabile a flusso di pacchetti (TCP)
    • Come sopra, ma con invio e ricezione solo per pacchetti
  – Trasmissione inaffidabile di pacchetti (UDP)
    • L'utente deve occuparsi di trattare gli eventuali errori

Per quanto riguarda il *file system*, con le seguenti caratteristiche:

• File visto da FS come sequenza di byte di significato arbitrario
  – Fissato dal programma applicativo
• File regolari, file repertorio (directory) e file speciali che mappano dispositivi di I/O
• Nome inizialmente limitato a 14 caratteri (UNIX v7)
• Poi esteso fino a 255 (UNIX BSD □ GNU/Linux)
  – Estensione non obbligatoria
  – Convenzione di estensione a scelta del programma applicativo e/o dell'utente

• File designato mediante cammino (path) assoluto o relativo
– Il cammino relativo richiede la nozione di directory di lavoro corrente
  • *pwd* per visualizzarne la posizione assoluta
  • *cd* per cambiare posizione
– Un intero FS B posto su una partizione visibile può essere ritenuto come parte di un FS A mediante mount
  • La radice di B viene designata con un nome (cammino) specifico in A detto mount point

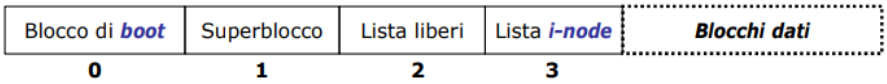
Controllo di accessi concorrenti (locking) - POSIX
  – A grana grossa (per directory o per file)
    • Mediante uso esplicito di semafori convenzionali
  – A grana fine (per gruppi di byte in un file)
    • Mediante meccanismi dedicati
Due distinte modalità d'uso
– Accesso simultaneo condiviso (shared lock)
  • Più accessi R alla stessa zona ma anche a zone solo parzialmente sovrapposte
– Accesso esclusivo (exclusive lock)
  • Consente un solo accesso per zona selezionata



Disponibili all'utente solo indirettamente tramite incapsulazione in procedure di libreria
– *lseek*
- Fissa l'indice di posizione all'interno di un file
  - Come offset espresso in byte rispetto ad un riferimento dato
- Accesso diretto

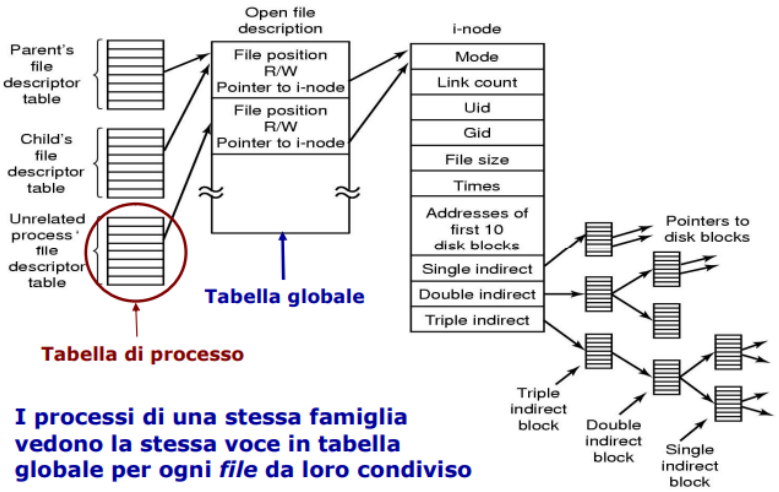
– *stat*
- Fornisce informazioni su file prelevandole dall'i-node corrispondente
- Chiamata incapsulata dal comando stat di shell

• Struttura di partizione secondo UNIX v7
• Il super-blocco (1) indica tra l'altro il # di i-node e di blocchi nel FS e fornisce il puntatore alla lista dei blocchi liberi (2)
• Gli i-node (3) sono numerati 1..N
• Directory come insieme variabile e non ordinato di unità informative (entry)
– Ampie 16 B
- 14 B (codifica ASCII) per nome di file
- 2 B per numero di i-node



In nucleo usa due strutture di controllo
– Un insieme di tabelle di processo contiene "descrittori utente" dei file attualmente in uso a ciascun processo
- A ogni descrittore utente deve corrispondere l'attuale posizione di R/W
- Però ogni processo deve avere il suo proprio indice di posizione sui propri file aperti
  - Possono esistere più posizioni di R/W su uno stesso file condiviso
  - L'indice non può essere ritenuto nell'i-node che è unico per file !

– Una tabella globale mantiene la corrispondenza tra tutti i file aperti e i loro i-node
- Ciascuna voce nella tabella di processo punta a una voce nella tabella globale che specifica diritti e posizione di R/W corrente nel file
  - La stessa voce file condiviso da processi di una stessa famiglia
  - Voce diversa per stesso file per processi non apparentati

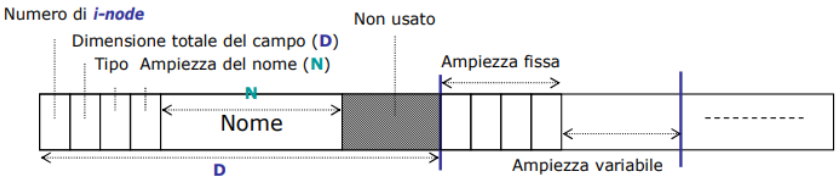


• L'i-node principale del file contiene (tra l'altro) l'indirizzo dei suoi primi 12 blocchi dati
– 1 i-node ha la dimensione di 1 frazione di blocco (64 B)
• Per file più grandi 1 campo dell'i-node principale punta a 1 i-node secondario che contiene puntatori ad altri blocchi dati
– I-node principale con campo single-indirect
• Per file ancora più grandi l'i-node secondario contiene puntatori a nodi single-indirect
– I-node principale con campo double-indirect
• È previsto anche un campo triple-indirect

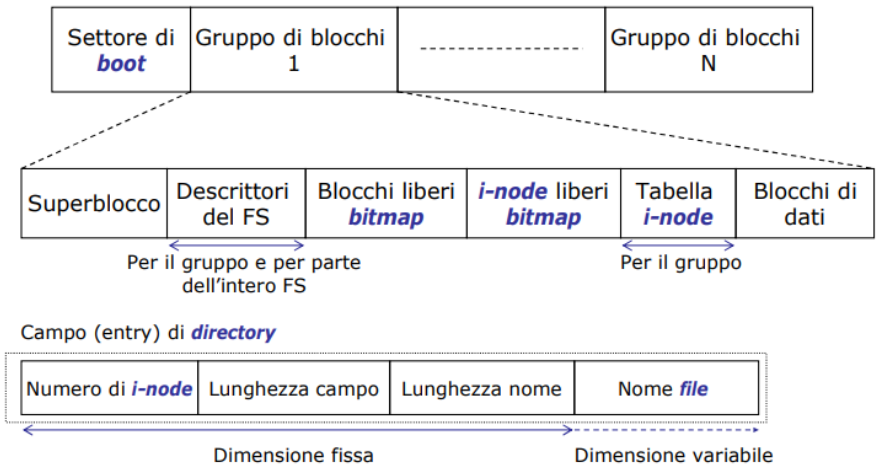
Esempio d'uso di *i-node* (UNIX v7)

- **Ipotesi**
  - Blocco dati di capienza 4 KB
  - *i-node* ampio 64 B
  - Indici di blocco espressi su 4 B
- **Esempio 1** (con uso di campo *single-indirect*)
  - Max dimensione di *file* rappresentabile
    - (12 + 64 B / 4 B) × 4 KB = (12 + 16) × 4 KB = 112 KB
- **Esempio 2** (con uso di campo *double-indirect*)
  - Max dimensione di *file* rappresentabile
    - 112 KB + 16^2 × 4 KB = 1 MB + 112 KB
- **Esempio 3** (con uso di campo *triple-indirect*)
  - Max dimensione di *file* rappresentabile
    - 1 MB + 112 KB + 16^3 × 4 KB = 17 MB + 112 KB

• La versione BSD (è la variante originaria del sistema operativo Unix, sviluppata presso l'Università di Berkeley in California, alla base di una delle due famiglie principali di sistemi operativi liberi attualmente più diffusi, tra cui gli esponenti più noti sono FreeBSD, OpenBSD, ecc.) introduce alcune migliorie importanti
– Estensione del nome di file fino a 255 caratteri
– Directory di dimensione multipla di blocco
• Facilita e velocizza la scrittura su disco
• Comporta frammentazione interna



Inizialmente basato sul FS di MINIX però subito abbandonato per le eccessive limitazioni
– Limitazioni imposte da MINIX
• Nomi ≤ 14 caratteri
• Indirizzi di blocco su 2 B per blocchi ampi 1 KB
• Ampiezza massima di file ≤ 64 MB (perché?)
ext2 diviene presto la versione di riferimento
– Basata sulle scelte BSD con diversa struttura fisica
– La maggiore innovazione è stata la suddivisione della partizione in gruppi di blocchi
• Distribuzione uniforme delle directory su disco
• i-node e relativi blocchi dati sono tenuti vicini tramite preallocazione di alcuni blocchi al momento della creazione di un file
– Riduce frammentazione



Frammentazione interna del blocco

- Dimensione di i-node estesa a 128 B
  - Indirizzi di blocco ampi 4 B
    - Per denotare fino a 232 = 4 G blocchi
- Blocchi di dimensione 1, 2, 4 KB scelta in fase di configurazione del FS
  - Partizione di dimensione ≥ 4 TB
  - 12 indirizzi diretti + 3 indiretti (single, double, triple)
  - Informazioni di controllo
  - Una parte riservata per uso futuro
  - Ogni aggiunta a file viene realizzata quanto più localmente possibile entro lo stesso gruppo
    - Località tra file correlati tramite gruppi
    - Località entro file mediante preallocazione di N ≤ 8 blocchi contigui

Microsoft Windows

Parentesi storica:

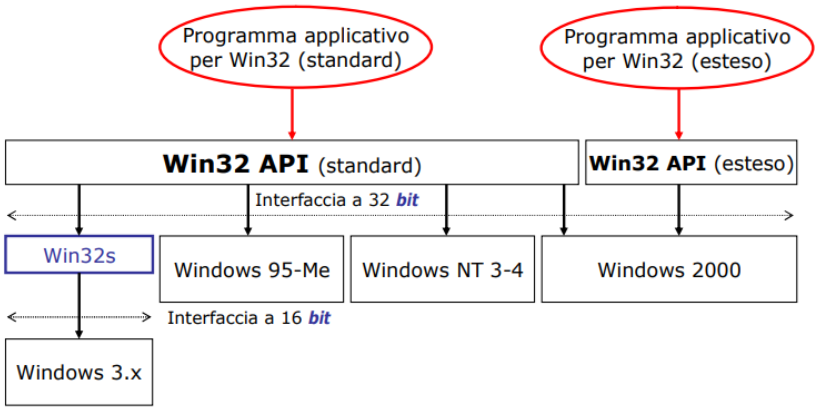
- MS-DOS (Microsoft Disk Operating System)
  - Mono-utente in modalità command line
  - Non multi-programmato
  - Inizialmente ispirato a CP/M
    - 1981 : 1.0 (8 KB codice) □ PC IBM 8088 (16 bit)
    - 1986 : 3.0 (36 KB) □ PC IBM/AT (i286 @ 8 MHz, □ 16 MB)
- Windows 1ª generazione
  - Modalità GUI solo come rivestimento di MS-DOS
  - Interfaccia copia del 1° modello Macintosh di Apple
    - 1990 - 1993 : 3.0, 3.1, 3.11 □ i386 (32 bit)
- GUI (Graphical User Interface)
  - Introdotto dal modello Macintosh di Apple il 24 gennaio 1984
- Finestre (windows), icone (icons), menu e dispositivi di puntamento (pointing)
- Realizzabile
  - Sia come programma in spazio utente (GNU/Linux)
  - Che come parte del S/O (Windows)

• Windows 2ª generazione
– Vero e proprio S/O multiprogrammato ma sempre mono-utente con FS su modello FAT
    • 1995 : Windows 95 (MS-DOS 7.0)
    • 1998 : Windows 98 (MS-DOS 7.1)
– Nucleo a procedure Incapaci di più esecuzioni simultanee
– Ogni accesso a nucleo protetto da semaforo a mutua esclusione
    » Scarsissimi benefici di multiprogrammazione
– ¼ dello spazio di indirizzamento di processo (4 GB totali) condiviso R/W con gli altri processi; ¼ condiviso R/W con il nucleo
    » Scarsissima integrità dei dati critici
• 2000 : Windows ME (ancora MS-DOS)

• Windows 3ª generazione
– Progetto NT: abbandono della base MS-DOS
• Architettura a 16 bit
– Enfasi su sicurezza e affidabilità
– FS di nuova concezione (NTFS)
    • 1993 : Windows NT 3.1 → fiasco commerciale per la mancanza di programmi di utilità
    • 1996 : Windows NT 4.0 → reintroduzione di interfaccia e programmi Windows 95
        – Scritto in C e C++ per massima portabilità al costo di grande complessità (16 M linee di codice!)
        – Molto superiore a Windows 95/98 ma privo di supporto per plug-and-play gestione batterie e emulatore MS-DOS
– Architettura di NT 3.1 a microkernel e modello client-server
    • La maggior parte dei servizi è incapsulata in processi di sistema eseguiti in modo utente e offerti ai processi applicativi tramite scambio messaggi
– Elevata portabilità
    • Dipendenze hardware localizzate nel nucleo
– Bassa velocità
    • Più costosa l'esecuzione in modo privilegiato
– Architettura di NT 4.0 a nucleo monolitico
    • Servizi di sistema riposizionati entro il nucleo
– 1999 : Windows 2000 (alias di NT 5.0)
    • Il S/O esegue in modo nucleo
        – Lo spazio di indirizzamento dei processi è interamente privato e distinto dal modo nucleo
    • Periferiche rimuovibili
        – Plug-and-play
    • Internazionalizzazione (configurabile per lingua nazionale)
    • Alcune migliorie a ntfs
    • MS-DOS completamente rimpiazzato da una shell di comandi che ne riproduce e estende le funzionalità
    • Enorme complessità: oltre 29 M linee di codice C[++]
– 2001: Windows XP
    • Migliorie grafiche rispetto a Windows 98 da cui ci si vuole definitivamente staccare
        – Un successo
    • Non è previsto utilizzo come tipo server
– 2007: Windows Vista
    • Windows Aero
    • Flop (lento e pesante)
        – Seppure più sicuro e con meno buffer overflow
    • Supporto WinXP esteso fino ad aprile 2014

– 2009: Windows 7
  • A meno di 3 anni dall'uscita del predecessore
    – Corsa ai ripari per insuccesso Vista
  • Miglioramenti in efficienza e interfaccia grafica
    – Supporto multitouch
  • In alcune versioni include un emulatore di XP
    – Per assicurare assoluta compatibilità e facilitare transizione
– 2012: Windows 8
  • Integrazione PC, Smartphone, Tablet
  • Windows Store per distribuzione app
  • Interfaccia con Tiles
  • Efficienza energetica (consumo batteria dispositivi mobili

Basato su principio speculare a quello adottato da UNIX e GNU/Linux
– Interfaccia di sistema non pubblica
– Procedure di libreria pubblicate in Win32 API (Application Programming Interface) a uso del programmatore ma controllata da Microsoft
– Alcune procedure includono chiamate di sistema
– Altre svolgono servizi di utilità eseguiti interamente in modo utente
• Nessun sforzo di evitare ridondanza o rigore gerarchico



Tutte le informazioni vitali di configurazione del sistema sono raccolte in una specie di FS detto *registry* salvato su disco in file speciali detti *hives*
– Directory → key
– File → entry = {nome, tipo, dati}
• 6 directory principali con prefisso *HKEY_*
  – Per esempio: HKEY_LOCAL_MACHINE con entry descrittive dell'hardware e delle sue periferiche (HARDWARE) dei programmi installati (SOFTWARE) e con informazioni utili per l'inizializzazione (SYSTEM)

Key	Description
HKEY_LOCAL_MACHINE	Properties of the hardware and software
HARDWARE	Hardware description and mapping of hardware to drivers
SAM	Security and account information for users
SECURITY	System-wide security policies
SOFTWARE	Generic information about installed application programs
SYSTEM	Information for booting the system
HKEY_USERS	Information about the users; one subkey per user
USER-AST-ID	User AST's profile
AppEvents	Which sound to make when (incoming email/fax, error, etc.)
Console	Command prompt settings (colors, fonts, history, etc.)
Control Panel	Desktop appearance, screensaver, mouse sensitivity, etc.
Environment	Environment variables
Keyboard Layout	Which keyboard: 102-key US, AZERTY, Dvorak, etc.
Printers	Information about installed printers
Software	User preferences for Microsoft and third party software
HKEY_PERFORMANCE_DATA	Hundreds of counters monitoring system performance
HKEY_CLASSES_ROOT	Link to HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
HKEY_CURRENT_CONFIG	Link to the current hardware profile
HKEY_CURRENT_USER	Link to the current user profile

Per quanto riguarda la *gestione dei processi,* facciamo le opportune distinzioni:

- ***Job*** **= {processi gestiti come singola unità con limiti risorse}**
- **Processo = possessore di risorse, con ≥ 1 *thread***

  ID unico, 4 GB di spazio di indirizzamento (2 in modo utente e 2 in modo nucleo), inizialmente con singolo *thread*, simile al processo UNIX; non ha stato di avanzamento
- ***Thread*** **= flusso di controllo gestito dal nucleo**

  Esegue per conto e nell'ambiente del processo (che non ha stato di avanzamento), con ID localmente unico, 2 *stack* (1 per modo)
- ***Fiber*** **= suddivisione di *thread* ignota al nucleo**

  Esegue nell'ambiente del *thread* e viene gestita interamente a livello di servizi offerti dal sottosistema Win32

I thread hanno vari modi per sincronizzarsi tra loro tramite oggetti di ordinamento

– Semafori binari (mutex) o contatori

– Sezioni critiche limitate allo spazio di indirizzamento del thread che le crea

– Eventi (oggetti del kernel)

- Thread attendono che si verifichino certi eventi
  - – Manual-reset events (rilasci manuali)
  - – Auto-reset events (al verificarsi dell'evento uno e uno solo viene rilasciato)

I thread hanno vari modi per comunicare senza bisogno di sincronizzarsi:

– Pipe : canali bidirezionali come in UNIX e GNU/Linux a sequenza di byte senza struttura oppure per messaggi (sequenze con struttura)

– Mailslot : canali unidirezionali anche su rete

– Socket : come pipe ma per comunicazioni remote

– RPC (chiamata di procedura remota) : per invocare procedure nello spazio di altri processi e riceverne il risultato localmente

– Condivisione di memoria : usando (porzioni di) file mappati in memoria

A loro volta, occorre un'opportuna *politica di ordinamento*.

- Ordinamento con prerilascio a priorità

– Effettuato da azioni esplicite del thread eseguite in modo nucleo → nessuna entità attiva dedicata di sistema

- Thread si blocca ad un semaforo, I/O, etc.
  - – Già in nucleo
- Thread segnala un oggetto (es. fa up di semaforo)
  - – Già in nucleo
- Al completamento del proprio quanto di tempo
  - – passa in modo nucleo tramite DPC per concludere l'interrupt handler

– Causato da attività esterne eseguite nel contesto del thread corrente

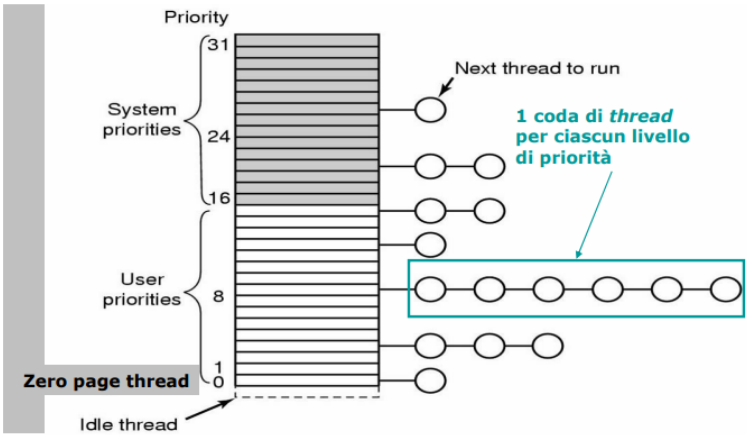
- Azioni di ordinamento programmate come DPC (Deferred Procedure Call) associate al trattamento di eventi asincroni

– Completamento operazione I/O

– Scadenza timer

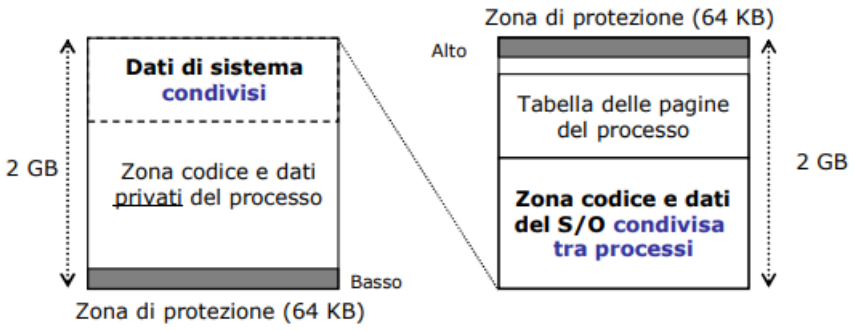
- 6 classi di priorità per processo
  - Realtime, high, above-normal, normal, below-normal, idle
- 7 classi di priorità per thread
  - Time-critical, highest, above-normal, normal, below-normal, lowest, idle
- 32 livelli di priorità (31 .. 0)
  - Ciascuno associato a una coda di thread pronti
  - Thread non distinti per processo di appartenenza
  - 31 .. 16 priorità di sistema; 15 .. 0 priorità ordinarie
- Ricerca per priorità decrescente
- Selezione dalla testa della coda
- Priorità più elevate per processi I/O bound e interattivi

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

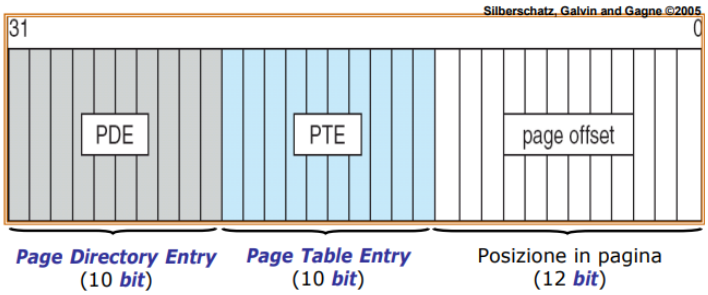


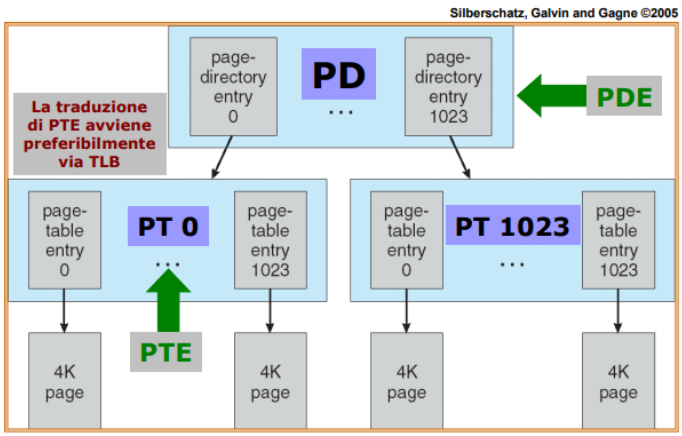
- Ciascun thread ha una priorità base iniziale e una corrente che varia nel corso dell'esecuzione
  - Entro la fascia della classe di priorità del processo di appartenenza
- La priorità corrente si eleva quando il thread
  - Completa un'operazione di I/O
    - Per favorire maggior utilizzazione delle periferiche
    - Insieme a un ampliamento temporaneo della durata del quanto
  - Ottiene un semaforo o riceve un segnale d'evento
    - Per ridurre il tempo di attesa dei processi interattivi
- La priorità corrente decresce a ogni quanto consumato
  - Il quanto può essere più ampio se è riferito alla finestra in primo piano
- Usa una tecnica brutale per mitigare il problema di inversione di priorità
  - Un thread pronto non selezionato per un certo tempo riceve un incremento di priorità per 2 quanti

In merito alla *gestione della memoria*, ogni processo dispone di uno spazio di indirizzamento virtuale paginato ampio 4 GB e suddiviso in 2 zone adiacenti ampie 2 GB ciascuna, con indirizzi virtuali espressi su 32 bit.



Scritto da Gabriel





Una pagina virtuale può essere

– R (lettura) / W (scrittura) / E (esecuzione)

– Libera (free): non riferita da alcun PTE

- Tutte le pagine di un processo sono inizialmente libere (pagingon-demand)
- Page fault

– Assegnata (committed): in uso per codice o dati

- Viene riferita tramite indirizzo virtuale e caricata da disco ove non fosse già presente in RAM

– Prenotata (reserved): non ancora in uso, ma non libera

- Per agevolare l'assegnazione di pagine contigue a processi

– Alla creazione di un nuovo processo 1MB è riservato per lo stack

Più processi possono condividere l'accesso a pagine di uno stesso file mappato in memoria

– Un libreria condivisa DLL (Dynamic Link Library) è un tipico esempio di file mappato in memoria

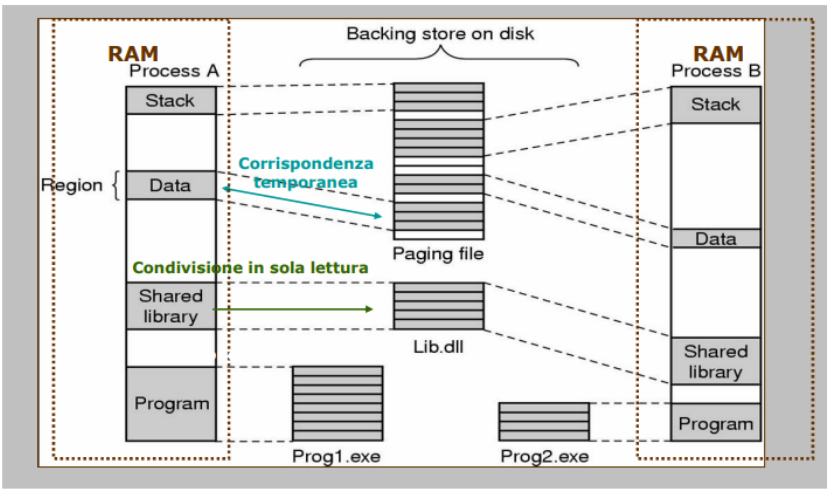
- Codice condiviso in sola lettura
- Dati statici R/W copiati per ciascun processo (copy-on-write)

– Ogni processo che accede a un file possiede specifici diritti di accesso che il S/O si preoccupa di far rispettare

La stessa posizione nel file può corrispondere ad indirizzi virtuali diversi per processi distinti

– Gli indirizzi riferiti nel codice condiviso di DLL devono pertanto, essere espressi in modo relativo

– A cura del compilatore



- Il caricamento di una nuova pagina in RAM può richiedere il rimpiazzo locale di un pagina "vecchia"

– Solo se non vi sono abbastanza pagine libere

– Il sistema mantiene una lista delle pagine libere

– A ogni processo i si associa l'insieme delle sue pagine attualmente in RAM (Working Set), la cui ampiezza può variare solo entro limiti prefissati

- Politica di rimpiazzo locale

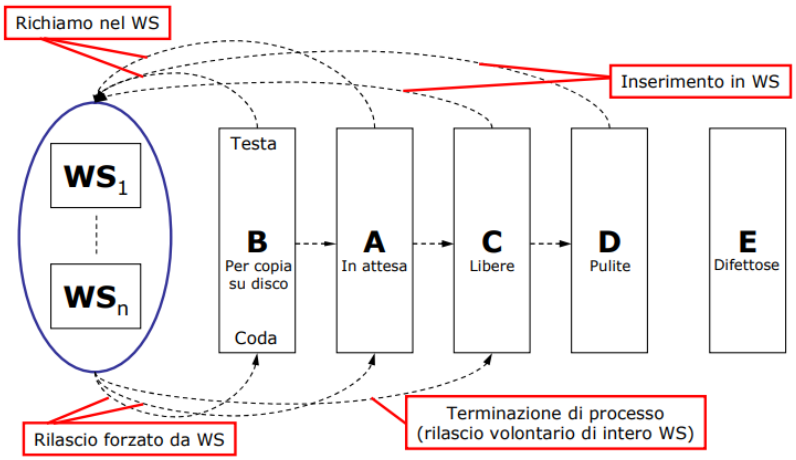
- Si ha rimpiazzo globale se e solo se un particolare processo deve scambiare proprie pagine tra RAM e disco troppo spesso

Scritto da Gabriel

• Anche il S/O stesso è visto come un processo con un proprio WS con pagine rimpiazzabili
  – Min set iniziale nell'ordine di 20-50 pagine
  – Max set iniziale nell'ordine di 45-345 pagine
  – Solo alcune pagine del S/O sono inamovibili
• Un daemon di kernel con periodo 1 s accerta che vi siano sufficienti pagine libere
  – Balance set manager
• Se insufficienti il daemon attiva un thread del Memory manager che esamina con una euristica i WS dei processi per rilasciarne pagine
  – Working set manager
  – Processi non recentemente attivi con WS ampi vengono scrutinati prima degli altri
  – Le pagine necessarie si prelevano dagli WS di ampiezza vicina al massimo e con scarso uso recente

• Ciascuna page frame in RAM può essere
  – In uso e appartenere a 1 WS (≥1 se condivisa)
  – Rilasciata e appartenere a 1 e 1 sola lista tra:
    • [A] In attesa: pagina recentemente rimossa dal WS di un processo ma ancora associata a esso e non modificata
      – Può essere riassegnata e sovrascritta senza problemi
    • [B] Da copiare su disco: ~ A ma se rimpiazzata deve essere riportata su disco
    • [C] Libera: ~ A ma non più associata ad alcun processo
    • [D] Azzerata: ~ C ma con contenuto obliterato a zero per consentire riassegnazione senza travaso di info privata
    • [E] Difettosa: pagina che non può più essere utilizzata a causa di difetti nella zona di memoria fisica

Lo swapper thread (daemon) del Memory manager porta in [A] o [B] le pagine dello stack dei processi i cui thread siano stati tutti recentemente inattivi. Altri 2 daemon assicurano che vi siano abbastanza pagine in [C] salvando su disco quelle in [B] e poi accodandole in [A]. Un WS che cresce preleva pagine libere da [C] se le sovrascrive interamente (senza conservare dati precedenti) da [D] altrimenti.
Un daemon dedicato che opera per conto del kernel azzera periodicamente il contenuto di pagine in [C] e le pone in [D].



• Euristiche complesse e non garantite governano le scelte effettuate dalle varie attività di gestione delle liste [A] – [D]
– L'amministratore di sistema può influenzare alcune euristiche mediante parametri di configurazione
• Lo stato della RAM viene mantenuto in una tabella dedicata acceduta per indice di pagina fisica (page frame database)
– Pagina valida/invalida, contatore dei riferimenti, WS di appartenenza, lista di appartenenza, etc.

Il file system di Windows NT (NTFS) offre una combinazione di prestazioni, affidabilità e compatibilità non riscontrabile nel file system FAT. È stato progettato per eseguire rapidamente operazioni di file standard come lettura, scrittura e ricerca, e anche operazioni avanzate come il ripristino del file system, su dischi rigidi molto grandi.

supporta l'intera gamma dei FS Windows e **anche** `ext2fs` di GNU/Linux

- **FAT-16**
  - Limite **logico** all'ampiezza di partizione
  - $\leq 2^{16}$ **blocchi** di ampiezza massima 32 KB $\rightarrow$ 2 GB
- **FAT-32**
  - Limite **fisico** all'ampiezza di partizione
  - $\leq 2^{32}$ **settori** da 512 B $\rightarrow$ 2 TB
  - Limite logico : 2^{28} blocchi da 8 KB $\rightarrow$ 2 TB
- **NTFS**
  - **Nuova concezione** con indici espressi su 64 *bit*

La formattazione di un volume con il file system NTFS comporta la creazione di diversi file di sistema e della Master File Table (MFT), che contiene informazioni su tutti i file e le cartelle del volume NTFS.
La prima informazione su un volume NTFS è il settore di avvio della partizione, che inizia dal settore 0 e può essere lungo fino a 16 settori.

Il file system NTFS include le caratteristiche di sicurezza necessarie per i file server e i personal computer di fascia alta in un ambiente aziendale. Il file system NTFS supporta anche il controllo dell'accesso ai dati e i privilegi di proprietà, importanti per l'integrità dei dati critici.
Mentre alle cartelle condivise su un computer Windows NT vengono assegnate particolari autorizzazioni, ai file e alle cartelle NTFS possono essere assegnate autorizzazioni indipendentemente dal fatto che siano condivise o meno. NTFS è l'unico file system di Windows NT che consente di assegnare autorizzazioni a singoli file.

Il file system NTFS ha un design semplice ma molto potente. Fondamentalmente, tutto ciò che si trova sul volume è un file e tutto ciò che si trova in un file è un attributo, dall'attributo dei dati, all'attributo della sicurezza, all'attributo del nome del file.
Ogni settore allocato su un volume NTFS appartiene a un file. Anche i metadati del file system (informazioni che descrivono il file system stesso) fanno parte di un file.

Nome di file fino a 255 caratteri in codifica UNICODE (2– 4 B/carattere)
  – Un nome espresso come cammino (relativo o assoluto) non può eccedere (32 K – 1) caratteri
  – Distinzione tra maiuscolo e minuscolo, ma senza effetto per buona parte di Win32 API (backward compatibility)
- File come aggregato di attributi rappresentati come sequenza di caratteri (byte stream)
  – Esempio: sequenza breve contenente il nome del file e l'indirizzo dell'oggetto associato (+ thumbnail preview) + sequenza lunga (fino a 264 B !) contenente i dati del file
- FS ad architettura gerarchica (come ext2)
  – \ invece di / come separatore nelle espressioni di cammino
  – Supporto per entrambe le forme di link
- Servizi di FS resi tramite procedure di libreria Win32 API

NTFS è una collezione di volumi logici
- – Un volume logico può mappare su più partizioni e anche su più dischi
- – Volume = sequenza lineare di blocchi (cluster) di ampiezza fissa
- • Volumi diversi possono avere dimensione di blocco diverse
  - – Tra 512 B e 4 KB
  - – Teoricamente fino a 64 KB
- • Blocco piccolo ridotta frammentazione interna
- • Blocco grande meno accessi a disco ma più frammentazione

La principale struttura dati è la MFT (Master File Table)
- – Una per volume
- – Fisicamente realizzata come un file
  - • Perciò può essere salvata ovunque nel volume
    - – Soluzione più robusta
- – Logicamente strutturata come una sequenza lineare di ≤ 248 record di ampiezza da 1 a 4 KB
  - • Ciascun record descrive 1 file identificato da un indice ampio 48 bit
  - • Gli altri 16 bit servono come numero di sequenza (contatore di riuso)

Ciascun record contiene un numero variabile di coppie *<descrittore di attributo, valore>*
– Il 1° campo della coppia specifica la struttura dell'attributo (che tipo di attributo è, quanto è lungo il suo valore)
- • Esistono 13 attributi di base con struttura predefinita
- • Possono esistere altri attributi aggiuntivi a struttura libera

– Il 2° campo denota il valore dell'attributo
- • Se possibile il valore è rappresentato interamente nel record
  - – Attributo residente
- • Altrimenti rappresentato da un puntatore al suo record remoto
  - – Attributo non residente

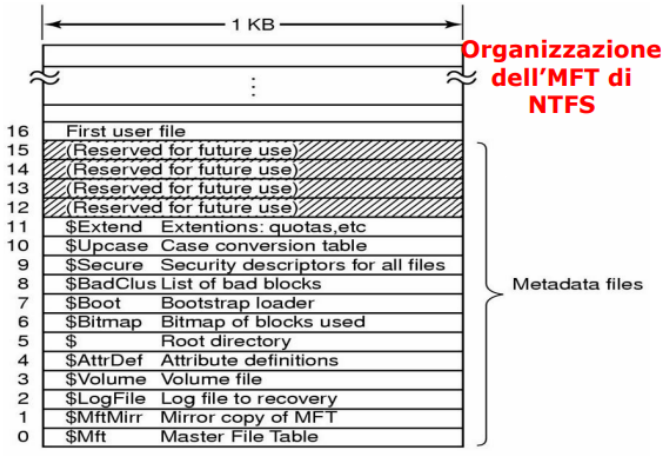
– Il valore dell'attributo dati rappresenta il contenuto effettivo del file

• I primi 16 record dell'MFT sono riservati per "file trascendenti" di sistema (metadata)
- – Questi record descrivono l'organizzazione dell'intero volume

• Il 1° record (0) descrive l'MFT stesso
• Il 2° (1) replica i primi 16 record in modo non residente ponendone il contenuto in fondo al volume (mirror file)
- – Facilita il ripristino del volume in caso di corruzione dell'MFT

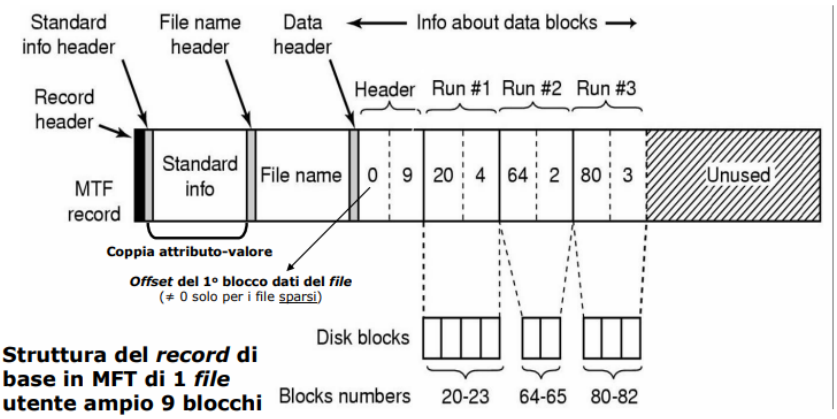
• Il 4° (3) caratterizza il volume (nome logico, versione di FS, data di creazione, etc.)
• Il 5° (4) descrive gli attributi usati nel volume
- – Gli attributi non residenti sono denotati da un puntatore di 48 bit a un record remoto e un codice di controllo di 16 bit che deve coincidere con quello del record di base in MFT (64 bit in tutto)

Inoltre: puntatore alla radice del FS; bitmap dei blocchi liberi; copia del codice di boot di volume o suo puntatore; etc.



• Il campo *<descrittore di attributo>* per attributi residenti ha ampiezza 24 B
  – Quello per attributi non residenti è più ampio
• Non tutti i 13 attributi di sistema applicano a tutti i file
  – Gli attributi previsti per i file corrispondono a quelli che GNU/Linux pone negli i-node con l'aggiunta dell'identificatore dell'oggetto corrispondente
  – Il contenuto dati di file di ampiezza < 1 KB viene memorizzato interamente entro un record di MFT
    • Immediate file (rari)
  – Per file più grandi il valore dell'attributo dati diventa la lista ordinata dei corrispondenti blocchi su disco
• NTFS cerca di assegnare allo stesso file sequenze di blocchi contigui piuttosto che singoli blocchi isolati
  – Strategia analoga a quella di ext2fs
  – Nel caso peggiore i dati di un file possono trovarsi su sequenze di blocchi singoli non adiacenti
• Esiste 1 record base in MFT ∀ file sequenziale presente nel volume
  – La struttura interna del record dipende dalla dimensione del file e dalla contiguità dei suoi blocchi
  – File con zone interne non utilizzate (e.g. poste a 0 e riservate per uso futuro) sono chiamati file sparsi e sono gestiti diversamente

Record base senza estensioni – 1



Struttura del *record* di base in MFT di 1 *file* utente ampio 9 blocchi

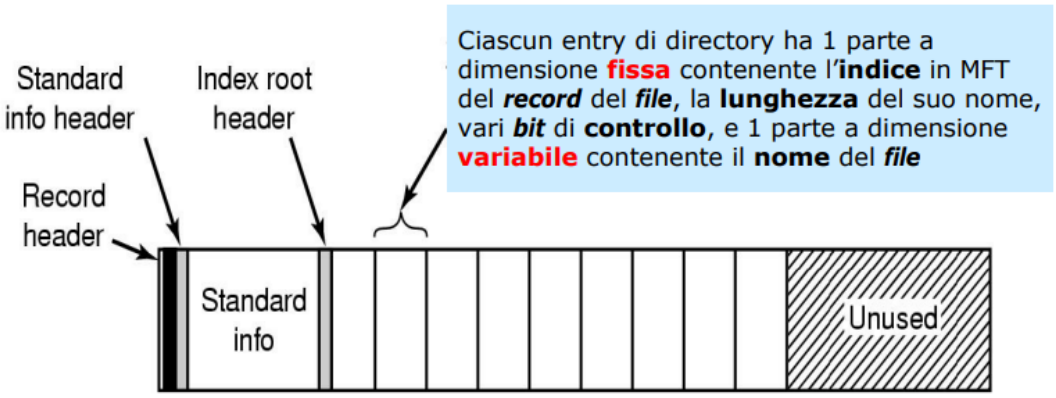
Nella figura un singolo descrittore basta a contenere l'intera lista di sequenze (run) di blocchi contigui di dati del file
– 9 blocchi dati in totale suddivisi in 3 sequenze
ciascuna descritta come :
  • Indirizzo su disco del 1° blocco della sequenza
  • Ampiezza in blocchi della sequenza
– L'intestazione dell'attributo dati specifica il # di sequenze presenti nel descrittore (3 in questo caso)
– La prima coppia di attributi dati specifica l'offset entro il file del 1° blocco coperto dal descrittore e l'offset del 1° blocco non coperto (= ampiezza)

La strategia NTFS consente di rappresentare file di ampiezza virtualmente illimitata. Il numero di record necessari per i dati di 1 singolo file dipende dalla contiguità dei suoi blocchi piuttosto che dalla sua ampiezza

– 1 file da 20 GB costituito da 20 sequenze di 1 M blocchi da 1 KB ciascuno richiede 20+1 coppie di valori espressi su 64 bit ovvero 21 x 2 x 8 B x 336 B

– 1 file da 64 KB costituito da 64 sequenze di 1 blocco ciascuna richiede (64+1) x 2 x 8 B x 1040 B

Il *record* base in MFT per una *directory* di piccole dimensioni



Per il *record con estensioni*, la rappresentazione di file può richiedere più record

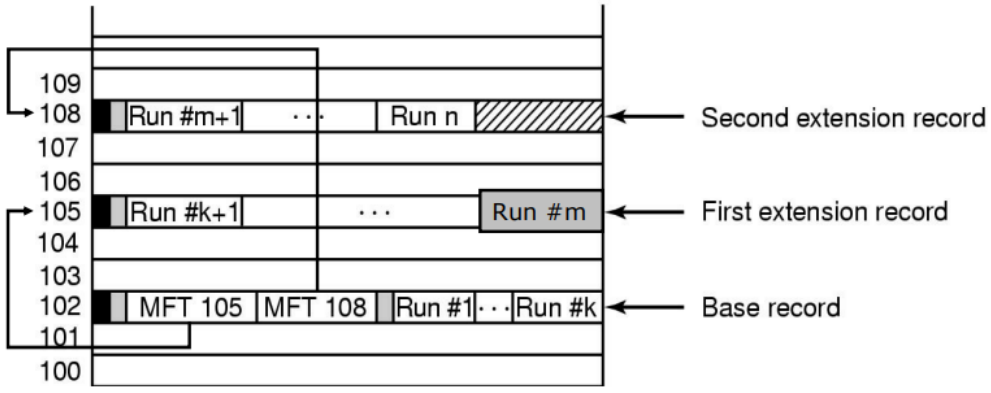
• NTFS usa per questo una tecnica a continuazioni analoga a quella usata dagli i-node di UNIX e GNU/Linux

– Il record base in MFT contiene un puntatore a N − 1 record secondari in MFT che descrivono la sequenza di blocchi del file

– Lo spazio rimanente del record base può descrivere le prime sequenze di blocchi dati del file

• Se non vi fosse abbastanza spazio in MFT per i record secondari di un dato file la loro intera lista verrebbe trattata come un attributo non residente e posta in un file dedicato denotato da un record posto in MFT

Un *file* composto da **n** sequenze di blocchi dati con descrizione specificata su 1 *record* base e 2 *record* di estensione in MFT



Esercizi ricapitolazione

Quesito 1 (punti 8). Sia data una memoria secondaria di ampiezza 64 GB, organizzata in blocchi di ampiezza 1 kB. Dopo aver calcolato la dimensione minima di un indice di blocco per tale memoria, sotto il vincolo che essa debba essere un multiplo di 8 (*bit*), si determini la dimensione massima di *file* ottenibile nel caso pessimo di contiguità nulla sotto le seguenti ipotesi:

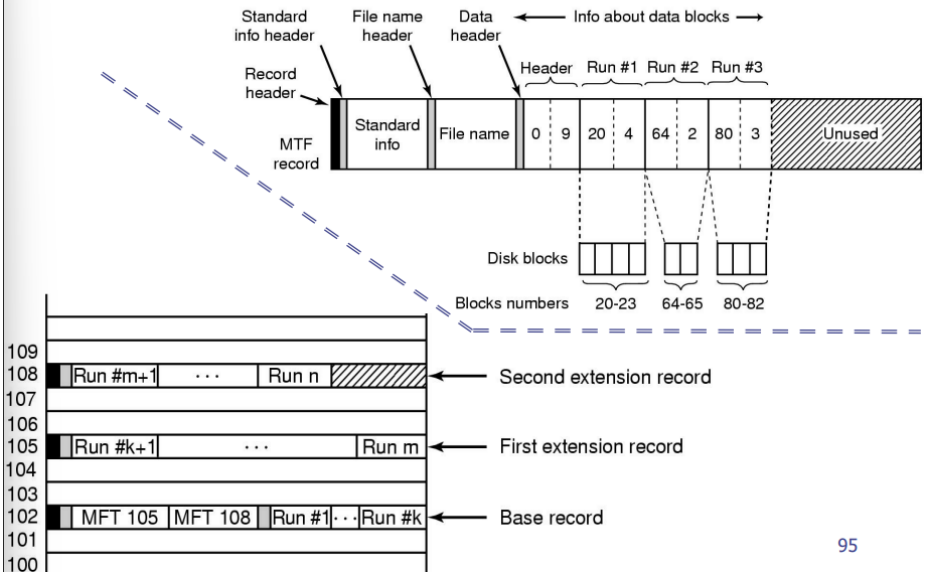
1. *file system* di tipo NTFS, con *record* ampi 1 kB, 408 B riservati all'attributo dati nel *record* principale ed 800 B nei *record* di estensione, utilizzando esattamente 2 *record*;
2. *file system* di tipo Extfs, con *i-node* ampi 1 kB, nodo principale contenente 16 indici di blocco ed 1 indice di I e di II indirezione, utilizzando l'intero *i-node* principale.

Calcolate le dimensioni richieste, si determini per ciascun tipo di *file system*, il rapporto inflattivo determinato dalla sua organizzazione strutturale, ossia l'onere proporzionale dovuto alla memorizzazione delle strutture di rappresentazione rispetto a quella dei dati veri e propri.

Soluzione 1 (punti 8). Essendo la memoria secondaria ampia 64 GB e i blocchi ampi 1 kB, è immediato calcolare che siano necessari $\lceil \frac{64\ GB}{1\ kB} \rceil = 64\ M = 2^6 \times 2^{20} = 2^{26}$ indici, la cui rappresentazione binaria banalmente richiede 26 *bit*. Stante il vincolo che la dimensione dell'indice debba essere un multiplo di 8 *bit*, la dimensione dell'indice deve salire a 32 *bit* (4 B). Vediamo ora quale possa essere la dimensione massima di *file* ottenibile sotto le ipotesi fissate dal quesito.

***File system* di tipo** NTFS : Dei 408 B riservati all'attributo dati nel *record* principale, $2 \times 4 = 8$ B saranno riservati alla coppia {base, indice}, mentre i rimanenti $408 - 8 = 400$ B potranno essere utilizzati per indicare le sequenze contigue di caso peggiore (dunque tutte ampie 1 blocco). Poiché ciascuna sequenza richiede una coppia di indici {inizio, fine}, pari a $2 \times 4 = 8$ B, il *record* principale potrà ospitare $\lfloor \frac{400\ B}{8\ B} \rfloor = 50$ sequenze ampie 1 blocco. Il *record* di estensione dispone invece di 800 B per la memorizzazione di $\lfloor \frac{800\ B}{8\ B} \rfloor = 100$ ulteriori sequenze. Ne segue che, sotto le ipotesi del quesito, la dimensione massima di *file* consentita da NTFS è pari a: $(50 + 100)$ blocchi $\times$ 1kB/blocco = 150 kB, al costo di 2 *record*, ciascuno ampio 1 kB. Il rapporto inflattivo in questo caso è dunque pari a: $\frac{2\ kB}{150\ kB} = 1.33\%$.

Soluzione (descrizione *record* MFT)



95

Soluzione

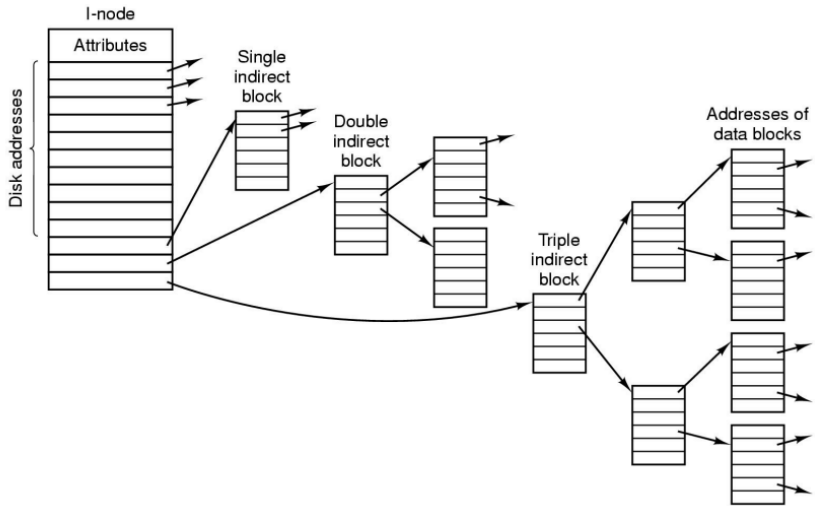
***file system* di tipo** Extfs : In questo caso, utilizzando tutti i campi dell'*i-node* principale, abbiamo a disposizione:

- 16 indici diretti di blocco, al costo di 1 blocco poiché un *i-node* occupa lo stesso spazio di un blocco;
- 1 indice di I indirezione, il quale punta ad un *i-node* interamente utilizzato per contenere indici diretti di blocco, che consente di esprimere fino a: $\lfloor \frac{1\,kB}{4\,B} \rfloor = \lfloor \frac{2^{10}}{2^2} \rfloor = 2^8 = 256$ indici di blocco, al costo di 1 ulteriore blocco;
- 1 indice di II indirezione, il quale punta ad un *i-node* speciale, interamente utilizzato per contenere puntatori ad *i-node* di I livello, che dunque consente di esprimere 256 puntatori a strutture ciascuna contenente fino a 256 indici diretti di blocco, per un totale di $256^2 = (2^8)^2 = 2^{16} = 65.536$ blocchi, al costo di $1 + 256 = 257$ ulteriori blocchi.

Conseguentemente, Extfs consente di rappresentare *file* di dimensione massima pari a: $(16 + 256 + 65.536) \times 1\,kB = 65.808\,kB = 64\,MB + 272\,kB$ (438,72 volte maggiore di quanto ottenuto con NTFS) per un rapporto inflattivo pari a: $\frac{(1+1+257)\times 1\,kB}{65.808\,kB} = 0,39\%$ (3,4 volte inferiore a quanto ottenuto con NTFS).

La considerazione ovvia da trarre da queste considerazioni è che NTFS) è particolarmente penalizzato dalle condizioni di scarsa o nulla contiguità. (Per contro, ad Extfs la contiguità non giova in alcun modo.)

Soluzione (descrizione *i-node*)



Esercizio "*Keeping Track of Free Blocks*"

Sia dato un disco di 16 GB diviso in blocchi ampi 1 KB. Si considerino due possibili strutture per tener traccia dei blocchi liberi: lista concatenata e *bitmap*. Nel primo caso, ogni elemento della lista è costituito a sua volta da un blocco, il quale contiene indici di blocco (di 32 *bit* ciascuno), dei quali l'ultimo è riservato per l'indicazione del prossimo blocco di lista libera. Nel secondo caso l'uso di un *bit* 1 o 0 definisce se il corrispondente blocco sia libero o utilizzato.
Si calcoli l'occupazione di memoria delle due strutture.

Soluzione

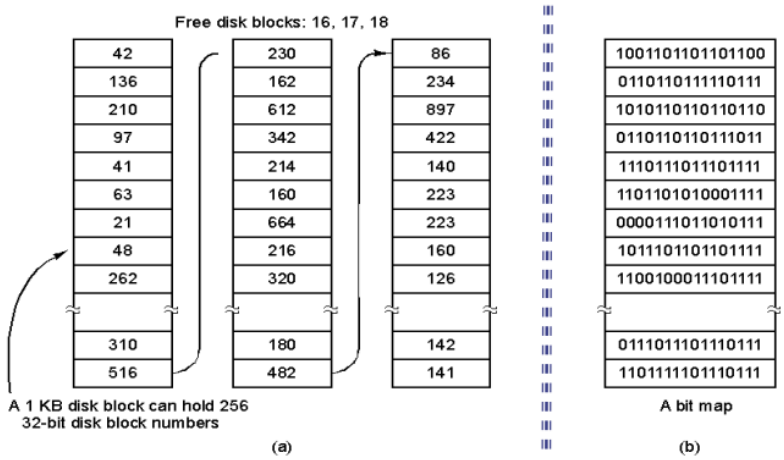
16 GB = 2^{34} B diviso in blocchi da 1 KB = 2^{10} B/blocco
ovvero 2^{34} B / 2^{10} B/blocco = 2^{24} blocchi = 16 M blocchi.

Ogni blocco può contenere 1 KB / 4 B/indice = 256 indici di blocco di cui 1 viene usato come collegamento al "blocco di indici" successivo nella lista. Ne rimangono dunque 255 utilizzabili per rappresentare i blocchi liberi.

Per rappresentare una lista di **massima ampiezza** servono dunque:
16 M indici / 255 indici/blocco = 65793,0039.. ≈ **65794** blocchi
cioè poco più di 64 K × 1 KB = **64 MB**

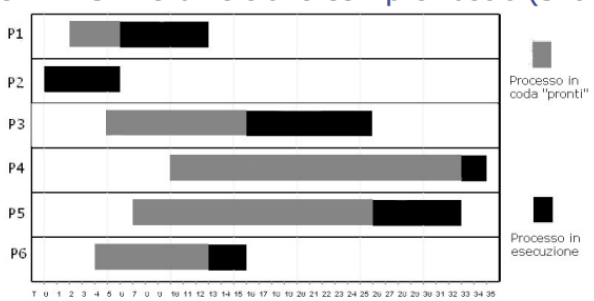
Con la struttura a *bitmap* sono invece **sempre** necessari 2^{24} *bit* = 2^{21} B =
= 2^{11} KB = **2 MB**

Soluzione



Esercizio - Simulatore

Considerando i processi P1, P2, P3, P4, P5 e P6, aventi un ordine di arrivo e di esecuzione su una macchina monoprocessore così come in figura si determini quale/i tra le seguenti politiche di scheduling senza priorità esplicite possa essere stata utilizzata:
1. FIRST IN FIRST OUT (Sì o No)
2. ROUND ROBIN (in caso di risposta positiva, indicare un esempio di ampiezza del quanto temporale)
3. SHORTEST JOB FIRST nella versione **con** prerilascio (Sì o No)



Soluzione

1. FIFO: Si'

2. RR: Si', con qualsiasi quanto temporale di ampiezza maggiore o uguale al massimo tempo di esecuzione fra i processi considerati, ovvero 10 u.t. (per P3)

3. SHORTEST JOB FIRST con prerilascio: No

Problema

- Numerosi operai in una fabbrica preparano un unico prodotto con l'utilizzo di **nA** quantità del componente **A**, **nB** quantità del componente **B**. Un fattorino viene chiamato a riempire le quantità di **A** e **B** fino a **totA** e **totB** ogni volta che le loro quantità residue scendono sotto **nA** ed **nB**. Prima di iniziare a comporre il prodotto, ogni operaio si assicura di avere le quantità necessarie dei due componenti; viceversa chiama il fattorino e attende che arrivi con la scorta. Una volta terminato il suo compito il fattorino resta in attesa di essere richiamato. Una volta completato un prodotto ogni operaio inizia a prepararne un altro. Inizialmente le scorte di **A** e **B** sono piene e nessun prodotto è stato ancora preparato.

- Scrivere le seguenti procedure che propongono una soluzione a questo problema utilizzando i costrutti dei monitor.

Soluzione

Il monitor potrebbe utilizzare i seguenti dati:

operaio: variabile condition sulla quale gli operai si sospendono in attesa di essere riforniti di un determinato componente;
fattorino: variabile condition sulla quale il fattorino si sospende in attesa di essere mandato a comprare le scorte di un componente;
operai_in_attesa: intero che indica il numero degli operai in attesa di un componente;
nA, **nB**: constanti che indicano la quantità necessaria di ogni componente per la preparazione di un prodotto;
totA, **totB**: costanti che indicano il numero massimo di scorte per ogni componente;
qA, **qB**: interi che indicano la quantità di componenti **A** e **B** attualmente presenti in fabbrica.

```
monitor Fabbrica {
condition operaio, fattorino;
int qA, qB;
int operai_in_attesa;
```

Soluzione

Funzione invocata da ciascun operaio all'infinito

```
void prepara_prodotto {
   while (true) {
      if (qA > nA) and (qB > nB) {
         qA = qA - nA;
         qB = qB - nB;
      else {
         fattorino.signal();
         operai_in_attesa++;
         operaio.wait();
      }
   }
}
```

Funzione invocata da fattorino

```
void scorta_ingrediente {
      qA = totA;
      qB = totB;
      while (operai_in_attesa > 0 ) {
            operaio.signal();
            operai_in_attesa --;
      }
      fattorino.wait();
}
```

Problema

- Sincronizzazione di 3 Processi coi Semafori.
- Si considerino i processi A, B e C che si sincronizzano come mostrato nel seguito attraverso i semafori Sem1, Sem2 e Sem3 (inizializzati a 0) e che operano sulle variabili condivise x, y e z, che sono inizializzate come segue: x = 1; y = 2; z = 1.
- Con quale ordine i processi stampano i valori delle tre variabili?
- Qual è il valore delle tre variabili che viene infine stampato?

Problema

Inizializzaz.

```
x = 1;
y = 2;
z = 1;
Sem1 = 0;
Sem2 = 0;
Sem3 = 0;
```

Process A {	Process B {	Process C {
P(Sem1);	P(sem2);	V(Sem2);
z = (x - z)*y ;	x = x + y;	P(Sem3);
x = x+z+y;	z = x - z;	x = x / y ;
V(Sem3);	y=(y-z)+x;	y = 2z + x ;
P(Sem1);	Print(y);	V(Sem1);
x = x + y;	V(sem1);	z = x + z;
Print(x);		Print(z);
}	}	}

Problema

Inizializzaz.

```
x = 1;
y = 2;
z = 1;
Sem1 = 0;
Sem2 = 0;
Sem3 = 0;
```

Process A {	Process B {	Process C {
P(Sem1);	P(sem2);	V(Sem2);
z = (x - z)*y ; =3	x = x + y; =3	P(Sem3);
x = x+z+y; =9	z = x - z; =2	x = x / y ; =3
V(Sem3);	y=(y-z)+x; =3	y = 2z + x ; =9
P(Sem1);	Print(y); «3»	V(Sem1);
x = x + y; ?? =12	V(sem1);	z = x + z; ??
Print(x); «12»		Print(z); «??»
}	}	}

Soluzione

- Il processo B stampa la variabile y per primo.
- Non possiamo fare affermazioni su quale processo stampa per secondo e per terzo tra A e C.
- Il risultato finale sarà x = 12 e y = 3 ma non possiamo dire se sarà z = 6 oppure z = 15 (a seconda di quale processo avanza per primo tra A e C)

Problema

- Si consideri una variante FAT-15 dell'architettura di file system nota come FAT-16, nella quale l'unica differenza dalla versione base sia che 1 bit dell'indice FAT non sia utilizzabile, tutti gli altri attributi di architettura rimanendo inalterati. Si specifichi la dimensione massima di file possibile con tale variante, e la dimensione massima di partizione.

Soluzione

- Come sappiamo l'indice X dell'architettura di file system FAT-X denota il numero di bit utilizzati per esprimere l'indice di blocco. Per le ipotesi del quesito abbiamo X = 15, il che significa che l'intera partizione potra' constare di 2^{15} = 32.768 blocchi. Poiche' il quesito richiede che tutte le altre caratteristiche dell'architettura in esame restino uguali allo standard FAT-16, i blocchi su disco saranno ampi al max 32 KB = $2^5 \times 2^{10}$ B = 2^{15} B. La dimensione massima di file cosi' come la dimensione massima di partizione saranno dunque fissate a:
  2^{15} blocchi × 2^{15} B / blocco = 2^{30} B = 1 GB,
  ovvero meta' della dimensione max ottenibile con FAT-16.

Problema

In un sistema le pagine hanno una dimensione di 1kB (1024), e la RAM è suddivisa in 256 frame. Nelle page table, un numero di pagina è scritto su 2 byte, e la page table più grande ammessa dal sistema occupa completamente un frame della RAM (per semplicità non consideriamo dirty bit e bit di validità). Rispondere alle seguenti domande:

a) il sistema usa un algoritmo di rimpiazzamento delle pagine?
b) qual è la lunghezza in bit di un indirizzo fisico?
c) il fatto che in un generico sistema lo spazio di indirizzamento fisico sia più piccolo dello spazio di indirizzamento logico è condizione necessaria perché il sistema soffra del problema del *thrashing*?

Soluzione

a) Si. Infatti una *page table* contiene al massimo 1024/2=512 *entries*, mentre il numero di frame della RAM è 256.

b) dimensione pagina=2^{10} byte; numero di frame = 2^8 => indirizzo fisico di 18 bit

c) No. Infatti può succedere che il sistema implementi la memoria virtuale e se la dimensione globale di tutti i processi attivi nel sistema eccede lo spazio di indirizzamento fisico allora è possibile il *thrashing*

Problema

In un sistema con paging, pagine di 2^6 bytes e la seguente page table

	In/Out	*Frame*
0	in	00101
1	out	01011
2	out	00001
3	out	11010
4	in	00011
5	out	10101
6	out	11111
7	in	10101

dire se i seguenti indirizzi logici genereranno un *page fault*. In caso negativo, scrivere l'indirizzo fisico corrispondente:
a) 0000001001001
b) 0000011010110
c) 0000100000101
d) 0000000111100

Soluzione

Visto che la dimensione della pagina è 2^6 bytes, gli ultimi 6 bit dell'indirizzo virtuale rappresentano l'*offset*, mentre i rimanenti sono il numero di pagina.

a) 0000001 001001: pagina 1 (cominciando a contare da zero nella page table). *Page fault.*

b) 0000011 010110: pagina 3. *Page fault.*

c) 0000100 000101: pagina 4. Pagina valida.
Diventa: 00011000101

d) 0000000 111100: pagina 0. Pagina valida.
Diventa: 00101111100

Problema Sincronizzazione Semafori

Si considerino tre processi (A, B e C) i quali devono eseguire alcune operazioni sulle variabili x e y e poi stamparne il risultato finale. A questo proposito, si consideri la seguente sequenza di avvenimenti:

- I processi A e C devono essere in attesa di essere risvegliati
- Il processo B sveglia il processo A, che a sua volta sveglia il processo C.
- I tre processi accedono concorrentemente (ma in mutua esclusione) alle variabili condivise x e y, poi B e C si bloccano in attesa che A termini la computazione su x e y (B e C avranno eseguito la loro computazione su x e y prima di bloccarsi)
- Terminata la computazione, è A stesso a svegliare B che a sua volta sveglia C.
- B stampa il valore di x e C stampa il valore di y (senza un ordine prestabilito).

[A] Assumendo che i semafori SemA, SemB e SemC siano inizializzati a 0, che il semaforo Mutex sia inizializzato a 1, e che inizialmente si abbia x = 2 e y = 1, si correggano e/o completino i frammenti di codice riportati nella slide seguente.

[B] Si discuta inoltre i possibili valori finali delle variabili x e y.

Process A {	Process B {	Process C {
	P(semB);	V(semA)
V(mutex);		
y= y + 2x ;	x= x + 1;	x= y * 3;
	V(mutex);	
P(SemA);	V(SemB);	V(SemC);
	Print(x);	Print(y);
}	}	}

Soluzione [A]

Process A {	Process B {	Process C {
P(semB); V(semA);	V(semB);	P(semA);
P(mutex); y = y + 2*x V(mutex);	P(mutex); x = x + 1 V(mutex);	P(mutex); x = y*3; V(mutex);
V(semA);	P(semA); V(semC);	P(semC);
	Print(x);	Print(y);
}	}	}

Soluzione [B]

I tre processi potrebbero entrare e uscire dalla sezione critica limitata da P(mutex) e V(mutex) in un ordine qualsiasi dunque per determinare il valore finale di x e y occorre provare tutte le combinazioni.

- Ordine A, B, C: risultato x = 15 e y = 5
- Ordine A, C, B: risultato x = 16 e y = 5
- Ordine B, A, C: risultato x = 21 e y = 7
- Ordine B, C, A: risultato x = 3 e y = 7
- Ordine C, A, B: risultato x = 4 e y = 7
- Ordine C, B, A: risultato x = 4 e y = 9

Problema – Paginazione e Hard Disk

Si consideri un elaboratore dotato di un sistema di memoria virtuale paginata e di un processore la cui frequenza di ciclo di clock sia 1 MHz. Si assuma che detto processore impieghi un ciclo di clock per eseguire istruzioni che non comportino riferimenti a pagine di memoria diverse da quella corrente. Si assuma inoltre che valgano le seguenti ipotesi:

• l'accesso a una pagina di memoria diversa da quella corrente comporti un costo temporale aggiuntivo di 1 μs;
• ciascuna pagina di memoria sia composta di 1.000 B;
• il disco fisso ruota alla velocità di 6.000 rpm (rotazioni/minuto);
• il trasferimento tra disco e memoria avvenga a 1.000.000 B/s;
• il tempo medio per spostare la testina dalla posizione corrente fin sopra la traccia dove si trova il punto di lettura/scrittura su disco sia di 10 ms;
• 1 blocco su disco corrisponda in dimensione a 1 pagina di memoria virtuale;
• 98% delle istruzioni eseguite facciano riferimento alla pagina corrente;
• 80% delle pagine accedute (diverse dalla corrente) si trovi già in memoria;
• quando una nuova pagina debba essere caricata in memoria, nel 50% dei casi quella rimpiazzata sia stata precedentemente modificata.

Si calcoli il tempo medio effettivo di esecuzione di ciascuna istruzione su tale elaboratore assumendo che il sistema stia eseguendo un unico processo e che il processore rimanga inattivo (stato *idle*) durante i trasferimenti di dati.

Soluzione – Parte 1/2

- Il processore considerato esegue una istruzione ogni µs quando essa non comporti riferimenti a pagine di memoria diverse da quella corrente.

- Per ipotesi, il 2% delle istruzioni richiede 1 µs in più per accedere ad aree di memoria diverse da quella corrente; di queste, il 20% richiede tempo ulteriore per leggere dal disco un pagina di memoria virtuale e, nel 50% dei casi, la pagina rimpiazzata dovrà essere copiata su disco, aumentando quindi il tempo di esecuzione della istruzione corrispondente. Il tempo medio di esecuzione risulta quindi dalla somma delle seguenti componenti:
  - tempo di esecuzione sul processore: 1 µs per tutte le istruzioni;
  - tempo di accesso a un'area di memoria diversa da quella corrente: 1 µs per il 2% delle istruzioni;
  - tempo di lettura da disco della pagina referenziata: tempo medio di posizionamento, rotazione e trasmissione, per il 20% delle istruzioni che fanno riferimento a pagine diverse dalla corrente;
  - tempo di scrittura su disco della pagina rimpiazzata: come per punto precedente, ma solo nel 50% dei casi in cui si accede a disco;

Soluzione – Parte 2/2

- Il disco ruota a 6.000 rpm, ovvero 100 rotazioni / secondo; una rotazione completa richiede quindi 10 ms.
  Statisticamente, quando la testina del disco si muove per leggere o scrivere su un'altra traccia, si troverà a mezzo giro di distanza dalla posizione cercata, aggiungendo quindi un ulteriore ritardo medio di 5 ms (equivalente a mezza rotazione) a ogni spostamento di testina.
  A ogni lettura o scrittura su disco occorre dunque aggiungere il tempo di spostamento della testina (per ipotesi, in media 10 ms), il tempo di rotazione del disco necessario per posizionare la testina sul punto iniziale di lettura o scrittura (in media 5 ms) e infine il tempo di trasferimento della pagina, pari a 1.000 B / 1.000.000 B/s = 1 ms.

  In totale quindi, ogni accesso a disco comporta un ulteriore ritardo di: 5 ms + 10 ms + 1 ms = 16 ms. In conclusione, il tempo medio di esecuzione di un'istruzione risulta essere: 1 µs + 0,02 × [1 µs + 0,2 × (16 ms +0,5 × 16 ms)] = 97,02 µs

Domanda – Hard Disk Infinito

- Gli hard disk sono componenti molto importanti di un computer che permettono di immagazzinare in modo permanente un insieme moderatamente grande di informazioni. Il sistema operativo si occupa di gestire anche queste componenti hardware permettendo, ad esempio, operazioni su file quali memorizzazione, recupero, cancellazione, ecc.
  I computer moderni sono dotati di hard disk di capacità sempre maggiore fornendo dunque un vantaggio in termini di spazio di memorizzazione agli utenti ma anche nuove complessità di gestione per il sistema operativo.

- Lo studente illustri, in massimo una pagina, le implicazioni (es. problematiche e possibili soluzioni, ma anche semplificazioni che diventerebbero possibili) per le varie componenti e strutture di un sistema operativo che si trovasse a dover gestire un hard disk di capacità infinita.

Risposta

- Molte funzioni del Sistema Operativo sarebbero coinvolte (e stravolte) nel dover gestire un hard-disk di dimensione infinita. Lo studente è invitato a rivisitare criticamente il programma del corso provando a riflettere sulle modifiche necessarie.

- Consideri ad esempio le implicazioni riguardo a:
  - dimensione degli indirizzi
  - dimensione strutture dati
  - utilizzabilità dei file system noti
  - reperire un file
  - contiguità file
  - lista blocchi liberi
  - importanza o meno della frammentazione
  - … (molto altro ancora)

Lettori e scrittori MULTIPLI

- Si consideri una versione modificata del problema dei lettori-scrittori in cui
  - Più lettori possono accedere contemporaneamente a db
  - Più scrittori possono accedere contemporaneamente a db
  - se uno o più scrittori chiedono di accedere mentre uno o più lettori stanno accedendo a db, lo scrittore deve attendere che i lettori abbiano finito
  - se uno o più lettori chiedono di accedere mentre uno o più scrittori stanno accedendo a db, il lettore deve attendere che gli scrittori abbiano finito

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base( );      /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc – 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read( );       /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data( );       /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base( );     /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

SOLUZIONE DELLA VERSIONE CLASSICA


```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base( );      /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc – 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read( );       /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data( );       /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base( );     /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

SOLUZIONE NUOVA VERSIONE

La nuova procedura ***writer*** sarà molto simile a ***reader***:

- *wc* al posto di *rc*

- *write_data_base()* al posto di *read_data_base()*