APPPUNTI SISTEMI OPERATIVI

di Simone Magagna

Introduzione

Sistema operativo: un insieme di utilità progettate per offrire all'utente un'astrazione più semplice e potente della macchina Assembler e gestire in maniera ottimale le risorse fisiche e logiche dell'elaboratore.

Processo: programma in esecuzione e corrisponde a:

- insieme ordinato di **stati** assunti dal programma nel corso dell'esecuzione
- insieme ordinato delle **azioni** effettuate dal programma nel corso dell'esecuzione

Kernel: costituisce il nucleo del SO. Gestisce l'avanzamento dei processi tramite un ordinamento (scheduling), gestisce le interruzioni esterne (eventi di I/O, situazioni anomale...), consente ai processi di accedere a risorse di sistema e di attendere eventi.

Risorsa CPU: indispensabile per l'avanzamento di processi. A livello fisico corrisponde al processore, a livello logico (sotto gestione software) può essere vista come macchina virtuale.

Risorsa Memoria: risorsa ad accesso multiplo per lettura, individuale per scrittura. Viene virtualizzata dal software (utilizzandola assieme alla memoria secondaria) attribuendone l'accesso secondo particolari politiche.

Risorsa I/O: riutilizzabile, non prerilasciabile. Il software ne facilita l'impiego nascondendo le caratteristiche hardware e uniformando il trattamento.

Il SO può risiedere permanentemente in ROM, oppure nella memoria secondaria per essere caricato (in parte) in RAM all'attivazione del sistema (in ROM risiede solo il caricatore del sistema).

Vi sono due orientamenti di scambio dei processi:

- **Scambio cooperativo**: il processo in esecuzione decide quando passare il controllo al processo successivo

- **Scambio con prerilascio**: il processo in esecuzione viene rimpiazzato:

  - da un processo a priorità maggiore

  - all'esaurimento del suo quanto di tempo

Il prerilascio si realizza tramite un meccanismo esterno all'esecuzione dei processi: un meccanismo solleva un interruzione e un gestore software la identifica e, se necessario, la notifica allo Scheduler.

Scheduler: componente del nucleo che decide le modalità di ordinamento dei processi. Bisogna rendere il suo operato parametrico rispetto a specifici attributi assegnati ai processi, configurandoli opportunamente.

Dispatcher: componente che avvia i processi all'esecuzione. Opera su mandato dello Scheduler. Il Dispatcher 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 utilizzo CPU

- numero processi avviati all'esecuzione per unità di tempo

- durata di permanenza di un processo in stato di pronto

I processi si possono classificare in:

- **CPU-bound**: attività sulla CPU di durata molto lunga

- **I/O-bound**: attività di breve durata sulla CPU, intervallate da attività di I/O molto lunghe

Esistono due possibili politiche di ordinamento:

- **Politica a rotazione con priorità**: code per ciascuna categoria di processo (CPU-bound, I/O-bound) e ordinate a priorità al loro interno, stabilendo poi una politica di ordinamento tra code

- **Politica a priorità con rotazione**: code per ogni livello di priorità. Selezione prioritaria tra code e a rotazione equa dentro ciascuna di esse

Sincronizzazione tra processi

Molti processi condividono risorse e informazioni funzionali. Per gestire la loro condivisione servono meccanismi di sincronizzazione di accesso. La modalità di accesso indivisa ad una risorsa condivisa viene detta "**in mutua esclusione**": l'accesso consentito ad un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa. Una soluzione al problema della sincronizzazione tra processi è ammissibile se soddisfa le seguenti 4 condizioni:

- garantire accesso esclusivo

- garantire attesa finita

- non fare assunzioni sull'ambiente di esecuzione

- non subire condizionamenti dai processi esterni alla sezione critica

Struttura Semaforo:

- **semaforo binario**: struttura composta da un campo valore intero (il quale può assumere solo i valori 0 o 1) e da un campo coda che accoda tutti i **PCB** dei processi in attesa sul semaforo. Possono essere effettuate due azioni:

- richiesta di accesso **P(down)**: decrementa il contatore se questo non è già a 0, altrimenti accoda il chiamante

- avviso di rilascio **V(up)**: incrementa il contatore di 1 e chiede al Dispatcher di porre in stato di pronto il primo processo in coda sul semaforo

```
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();}
}
```

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

- **semaforo contatore**: stessa struttura del binario ma usa una logica differente per il campo valore:

  - valore > 0 denota disponibilità non esaurita

  - valore < 0 denota richieste pendenti

```
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();}
}
```

Il valore iniziale indica la capacità massima della risorsa.

Struttura Monitor: è un aggregato di sottoprogrammi, variabili e strutture dati. Solo i sottoprogrammi del monitor possono accedere alle variabili interne. Solo un processo alla volta può essere attivo entro il monitor. Due procedure operanti su variabili speciali 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

La primitiva Wait permette di bloccare il chiamante qualora le condizioni logiche della risorsa non consentano l'esecuzione del servizio (es.: contenitore pieno per il produttore, contenitore vuoto per il consumatore).

La primitiva Signal notifica il verificarsi della condizione attesa al (primo) processo bloccato, risvegliandolo. Il processo risvegliato compete col chiamante della Signal per il possesso della CPU. Il segnale di risveglio non ha memoria.

Problemi classici di sincronizzazione:

- Produttore-Consumatore

- Filosofi a cena

- Lettori e Scrittori

- Barbiere che dorme

Problemi pensati per rappresentare tipiche situazioni di rischio:

- Stallo con blocco (deadlock)

- Stallo senza blocco (starvation)

- Esecuzioni non predicibili (race condition)

Condizioni necessarie e sufficienti affinché si verifichi 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 dal processo (scambio cooperativo))

- condizione di attesa circolare (un processo attende una risorsa in possesso del successivo processo)

Produttore-consumatore: due processi condividono un buffer comune, di dimensione fissa. Il produttore mette informazioni nel buffer e il consumatore le preleva. Se il produttore vuole mettere un nuovo elemento nel buffer quando questo è già pieno, esso “andrà a dormire” e verrà risvegliato solo quando il consumatore avrà rimosso uno o più elementi. Analogamente, se il consumatore vuole rimuovere un elemento dal buffer e vede che il buffer è vuoto, “va a dormire” finché il produttore non inserisce qualcosa nel buffer e lo risveglia. L’algoritmo del produttore-consumatore può essere risolto tramite:

-**Semafori:**

```
#define N 100

typedef int semaphore;

semaphore mutex = 1;

semaphore empty = N;

semaphore full = 0;


void producer void{

 int item;

 while(TRUE){

  item =produce_item();

  down(&empty);

  down(&mutex);

  insert_item(item);

  up(&mutex);

 up(&full);
```

```
 }}


void consumer(void){
 int item;
 while(TRUE){
  down(&full);
  down(&mutex);
  item_remove(item);
  up(&mutex);
  up(&empty);
  consume_item(item);
 }
}
```

-**Monitor:**

```
 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;
```

-**Passaggio di messaggi:**

```
#define N 100

void producer(void){
 int item;
 message m;
 while(TRUE){
 item = produce_item();
 receive(consumer, &m);
 build_message(&m, item);
 send(consumer, &m);
 }
}
void consumer(void){
 int item, i;
 merssage m;
 for(i=0; i<N; i++) send(producer, &m);
 while(TRUE){
  receive(producer, &m);
  item = extract_item(&m);
  send(producer, &m);
  consume_item(item);
 }
}
```

Filosofi a cena: il problema dei filosofi a cena simula l'accesso esclusivo a risorse condivise da parte dei processi. I filosofi sono seduti in un tavolo circolare con un piatto di spaghetti per ciascuno e una forchetta alla propria sinistra. L'attività di un filosofo è intervallata da momenti di riflessione e da momenti in cui mangia. Ogni filosofo per mangiare ha bisogno di due forchette. Per evitare il deadlock del sistema utilizziamo l'algoritmo del "filosofo mancino". Può essere implementato tramite:

-**Semafori:**

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

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]);
   }
  }
```

-Monitor:

```
Monitor Tavolo{

boolean fork_used[S] = false;

condition filosofo[S];


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;

}


deposita(int n){

 fork_used[n] = false;

 fork_used[(n+1)%5] = false;

 filosofo[n].notify();
```

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

Lettori e scrittori: un insieme di processi devono leggere e scrivere in un database condiviso. Più lettori possono accedere contemporaneamtente al database mentre gli scrittori devono averene accesso esclusivo. I lettori hanno precedenza sugli scrittori. L'algoritmo può essere implementato attraverso:

-Semafori:

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void){
 while(TRUE){
  down(&mutex);
  rc = rc+1;
  if(rc == 1) down(&db);
  up(&mutex);
  read_data_base();
  down(&mutex);
  rc = rc-1;
  if(rc == 0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(void){
 while(TRUE){
  think_up_data();
  down(&db);
  write_data_base();
  up(&db);
 }
}
```

Politiche di ordinamento processi

Ogni processo è associato a un descrittore chiamato **Process Control Block** che ne specifica le caratteristiche distintive:

- identificatore del processo
- contesto di esecuzione del processo
- stato di avanzamento del processo
- priorità
- diritti di accesso alle risorse ed eventuali privilegi
- discendenza familiare
- puntatore alla lista delle risorse assegnate al processo

Una decisione di scheduling è necessaria:

- alla creazione di un processo
- alla terminazione di un processo
- quando un processo si blocca
- all'ocorrenza di un interrupt

I **meccanismi** per realizzare scelte di ordinamento e gestione dei processi risiedono nel nucleo.

Le **politiche** sono determinate fuori dal nucleo decise nello spazio delle applicazioni decidendo quali valori assegnare ai parametri di configurazione dei processi considerati dai meccanismi di gestione.

L'efficienza delle politiche scelte si misura in termini di:

- percentuale di impiego utile della CPU
- numero di processi avviati all'esecuzione per unità di tempo (**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**)

Diverse classi di sistemi concorrenti richiedono politiche di ordinamento di processi specifiche:

- **sistemi a lotti**: ordinamento predeterminato; lavori di lunga durata e limitata urgenza; prerilascio non necessario. Le politiche per tali sistemi devono garantire:

massimo prodotto per unità di tempo, massima rapidità di servizio per singolo lavoro e massimo utilizzo delle risirse di elaborazione.

- **sistemi interattivi**: grande varietà di attività; prerilascio essenziale. Le politiche per tali sistemi devono garantire: rapidità di risposta per singolo lavoro e soddisfazione delle aspettative generali dell'utente.

- **sistemi in tempo reale**: lavori di durata ridotta ma con elevata urgenza; l'ordinamento deve riflettere l'importanza del processo; prerilascio possibile. Le politiche per tali sistemi devono garantire: rispetto delle scadenze temporali e predicibilità di comportamento. Il valore corretto deve essere prodotto entro un tempo fissato: oltre tale limite il valore ha utilità decrescente, nulla o addirittura negativa.

Politiche di ordinamento per sistemi a lotti:

- **FCFS(First Come First Served)**: senza prerilascio e senza priorità. Ordine di esecuzione = ordine di arrivo.

- **RR(Round Robin)**: imponendo divisione di tempo sulla politica FCFS si ottiene una tecnica di rotazione detta Round Robin. Con prerilascio senza priorità.

- **SJF(Shortest Job First)**: senza prerilascio; esegue prima il lavoro più breve.

- **SRTN(Shortest Remaining Time Next)**: variante di SJF con prerilascio; tiene conto dei nuovi processi quando arivano.

Politiche di ordinamento per sistemi interattivi:

- Ordinamento a quanti

- Ordinamento a quanti con priorità

- Con garanzia per processo

- Senza garanzia

- Con garanzia per utente

Politiche di ordinamento per sistemi in tempo reale:

- Modello semplice (cyclic executive)

- Ordinamento a priorità fissa

- Ordinamento a priorità fissa con prerilascio

Ricapitolazione dei concetti di base

Gerarchia fisica di memoria:

- **Registri** interni alla CPU: ampi 32 o 64 bit

- **Cache** suddivisa in blocchi chiamati line con ampiezza tipica 64 B: L1 all'interno della CPU, L2 adiacente alla CPU.

- **Dischi magnetici**: capienza 100 volte superiore e costo/bit 100 volte inferiore rispetto alla RAM ma con tempo di accesso 1000 inferiore.

CMOS: memoria volatile ma alimentata da una piccola batteria. Memorizza l'ora, da quale disco fare il boot e le impostazioni BIOS diverse da quelle di default.

MMU: gestisce lo spazio di memoria virtuale dei processi; logicamente interposta tra CPU e RAM ed è sotto la responsabilità del SO.

Trattamento delle interruzioni: l'uso delle interruzioni evita il ricorso al polling (verifica ciclica del SO a tutte le unità/periferiche). Avviene in 8 passi:

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 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
5. All'arrivo dell'interruzione i registri PC e PSW sono posti sullo stack del processo corrente
6. La CPU passa al modo operativo protetto
7. Il parametro principale che denota l'interruzione serve come indice nel vettore delle interruzioni, così si individua il gestore designato a servire l'interruzione
8. La parte immediata del gestore esegue nel contesto del processo interrotto

Plug & Play: prima ogni scheda I/O aveva un livello di interrupt fisso e un indirizzo fisso per i regitri. Con Plug & Play , il sistema assegna centralmente i livelli di interrupt e gli indirizzi di I/O e poi li rivela alle schede.

BIOS: salvato in un chip ROM, contiene software a basso livello per la gestione di I/O. Viene caricato all'avvio del computer. Verifica la quantità di RAM e quali dispositivi di base sono presenti. Fa lo scan dei bus PCI e ISA per rilevare dispositivi ad essi connessi. Determina il dispositivo di boot dalla lista in memoria CMOS.

BOOT: indica, in generale, l'insieme dei processi che vengono eseguiti da un computer durante la fase di avvio, in particolare dall'accensione fino al completo

caricamento in memoria primaria del kernel del sistema operativo a partire dalla memoria secondaria. 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 sia attiva. Da tale tabella viene caricato un secondo boot loader che legge il SO dalla partizione attiva e lo esegue. Il SO interroga il BIOS per ottenere informazioni sulla configurazione del sistema. Successivamente attua varie inizializzazioni ed esegue il programma iniziale (login, GUI).

Chiamate di sistema: la maggior parte dei servizi del Sistema Operativo sono eseguiti in risposta a invocazioni esplicite di processi. 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 deve attivare il modo operativo privilegiato. Il meccanismo complessivo è simile a quello già visto per il trattamento delle interruzioni:

1. Il programma applicativo effettua una chiamata di sistema
2. Poi invoca la procedura di libreria corrispondente alla chiamata
3. Questa pone l'ID della chiamata in un luogo noto al SO
4. Poi esegue l'istruzione trap per passare all'esecuzione in modo operativo privilegiato
5. Il SO individua la chiamata da eseguire
6. La esegue
7. Poi ritorna al chiamante oppure ad un nuovo processo
8. Cancella dati nello stack facendo avanzare il puntatore

Struttura del Kernel:

Struttura Monolitica: l'approccio monolitico definisce un'interfaccia virtuale di alto livello sull'hardware, con un set di primitive o chiamate di sistema per implementare servizi di sistema operativo come *gestione dei processi*, *multitasking* e *gestione della memoria*, in diversi moduli che girano in *modalità supervisore*. Anche se ogni modulo che serve queste operazioni è separato dal resto, l'integrazione del codice è molto stretta e difficile da fare in maniera corretta e, siccome tutti i moduli operano nello stesso spazio, un bug in uno di essi può bloccare l'intero sistema. Tuttavia, quando l'implementazione è completa e sicura, la stretta integrazione interna dei componenti rende un buon kernel monolitico estremamente efficiente. 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

Struttura 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 utente: i

processi di sistema sono visti come serventi mentre quelli utente sono visti come clienti. Il ruolo del nucleo è di gestire i processi e supportare le loro comunicazioni. Se un servizio va in crash difficilmente lo farà tutto il sistema.

Monolitico vs Micro-Kernel:

- i micro-kernel consentono gestione più flessibile
- i kernel monolitici sono più semplici da realizzare e da mantenere
- i micro-kernel hanno problemi di sincronizzazione tra le varie componenti che ne rallentano lo sviluppo e mantenimento

Gestione della Memoria

Il gestore della memoria è la componente di SO 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. Per processi soggetti a migrazione da RAM a disco durante l'esecuzione

La memoria disponibile è in generale inferiore a quella necessaria per tutti i processi attivi.

Sistemi monoprogrammati: un solo processo per volta. La memoria disponibile è ripartita tra solo quel processo e il SO. La parte di SO ospitata in RAM è però solo quella che contiene l'ultimo comando invocato dall'utente.

Sistemi multiprogrammati: la forma più rudimentale crea una partizione per ogni processo staticamente all'avvio del sistema. Tali partizioni possono avere dimensione diversa. Il problema è assegnare dinamicamente processi a partizioni minimizzando la frammentazione interna.

- A ogni nuovo processo viene assegnata la partizione di dimensione più appropriata. Una coda di processi per partizione. Scarsa efficacia nell'uso della memoria disponibile.
- Una sola coda per tutte le partizioni: quando si libera una partizione questa viene assegnata al processo a essa più adatto e più avanti nella coda oppure assegnata al miglior processo scandendo l'intera coda. Vengono discriminati i processi più piccoli.

Utilizzo stimato CPU: (1-P^N) dove:

- P = percentuale di tempo dedicato a I/O di un processo
- N = numero processi simultaneamente in memoria

Rilocazione: interpretazione degli indirizzi emessi da un processo in relazione alla sua corrente posizione in memoria.

Protezione: assicurazione che ogni processo operi all'interno dello spazio di memoria a esso permissibile.

Swapping: tecnica per alternare processi in memoria principale senza garantire allocazione fissa. Trasferisce processi interi e assegna partizioni diverse nel tempo. Il processo rimosso viene salvato su memoria secondaria. Processi diversi richiedono partizioni di ampiezze diverse assegnate ad hoc. Questo comporta rischio di frammentazione esterna, perciò occorre ricompattare periodicamente la memoria principale. Le dimensioni di memoria di un processo possono variare nel tempo ma è difficile aumentare dinamicamente l'ampiezza della partizione assegnata, perciò è meglio assegnarle con margine.

Con memoria allocata così dinamicamente è essenziale traccia del suo stato d'uso. Due strategie:

- **Mappe di bit**: memoria vista come insieme di unità di allocazione (q bit per unità)
- **Liste collegate**: memoria vista a segmenti. Ogni elemento di lista rappresenta un segmento: ne specifica punto di inizio, ampiezza e successore. Liste ordinate per indirizzo di base.

Politiche di allocazione per le liste collegate:

- **First fit**: il primo segmento libero ampio abbastanza.
- **Next fit**: come First fit ma ripartendo dall'ultimo segmento visto.
- **Best fit**: il segmento libero più adatto.
- **Worst fit**: il segmento libero più ampio.

Memoria Virtuale: l'intera RAM è presto divenuta insufficiente per ospitare un intero processo. Nasce così il concetto di memoria virtuale. 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. Ogni processo ha un suo proprio spazio di memoria virtuale. Perciò 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. La CPU emette indirizzi logici non più verso il bus ma verso l'MMU, la quale poi li trasformerà in indirizzi fisici. Due tecniche alternative di gestione:

Paginazione

L'obbiettivo è consentire spazi di indirizzamento più grandi della RAM. La memoria virtuale è suddivisa in unità a dimensione fissa dette pagine. La RAM è suddivisa in unità cornici ampie come le pagine, dette page frame. Il trasferimento da e verso disco avvengono sempre in pagine. Di ogni pagina occorre sapere se sia presente in RAM oppure no. Se una pagina è assente quando riferita, si genera un evento page fault gestito dal SO tramite trap. La traduzione da virtuale a fisico avviene tramite una tabella delle pagine:

indirizzo fisico = f(indirizzo virtuale).

Ciascun processo ha la sua grande tabella delle pagine, poiché ha il suo spazio di indirizzamento virtuale. Ogni indirizzo emesso dal processo deve essere tradotto:

- Tramite vettore di registri (uno per ogni pagina virtuale) caricato ad ogni cambio di contesto
- Struttura sempre residente in RAM: un singolo registro punta all'inizio della page table

L'indirizzo di disco ove la pagina si trova quando non è in RAM non è nella tabella. La tabella delle pagine serve all'MMU. Il caricamento della pagina da disco viene effettuato dal SO all'occorrenza di un page fault. La tabella delle pagine è troppo grande per stare nei registri perciò sta in RAM. Serve una struttura supplementare più agile che funga da cache. Piccola memoria associativa che consente scansione parallela: **TLB.** Solitamente interna alla MMU, ricerca su tutte le righe

simultaneamente. Ogni indirizzo emesso verso l'MMU viene prima trattato con la TLB. Se la sua pagina è presente e l'accesso è permesso la traduzione avviene tramite TLB, senza accedere alla page table. Se non è presente si ha un cache miss e le informazioni richieste vengono caricate in TLB dalla tabella delle pagine, rimpiazzandone una cella. Nei sistemi a 64 bit però le tabelle delle pagine assumono dimensioni proibitive. La soluzione adottata impiega una **Tabella Invertita:** non più una riga per pagina ma per page frame in RAM. La traduzione da virtuale a fisico diventa molto onerosa poiché la pagina potrebbe risiedere in qualunque page frame. Ricerca velocizzata dall'uso della TLB e realizzando la tabella come una tabella hash: i dati relativi alle pagine i cui indirizzi virtuali indicizzano una stessa riga di tabella vengono collegati in lista. Quando si produce un page fault il SO deve rimpiazzare una pagina, salvando su disco la pagina rimossa. Vi sono molteplici politiche di rimpiazzo:

- **Optimal replacement:** rimpiazza la pagina in memoria che non sarà utilizzata da maggior tempo. Non è realizzabile.
- **NRU (Not Recently Used):** per ogni page frame vengono aggiornati: bit M (modified), bit R (referenced) posto a 0 periodicamente. I page frame vengono classificati in:
  1. Classe 0: non riferita, non modificata
  2. Classe 1: non riferita, modificata
  3. Classe 2: riferita, non modificata
  4. 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
- **Second Chance:** corregge FIFO rimpiazzando solo le pagine con bit R=0, altrimenti il page frame viene considerato come appena caricato, posto in fondo alla coda con R=0.
- **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):** rimpiazza la pagina che da più tempo non viene riferita. Richiede la lista aggiornata ad ogni riferimento a memoria
- **NFU (Not Frequently Used):** per ogni page frame aggiorna periodicamente un contatore C che cresce di più se R=1. Il problema è che non dimentica nulla perciò una pagina riferita molte volte in passato e ora non più in uso non viene sostituita.
- **Aging:** NRU modificato: per ogni page frame aggiorna un contatore C che cresce di più se R=1 ma non incrementa C con R: gli inserisce R a sinistra. Usando N bit per C perde memoria dopo N aggiornamenti

Working set: è l'insieme delle pagine che un processo ha in uso in un dato intervallo di tempo. Se la memoria non basta ad accogliere il WS si crea un fenomeno di thrashing (un programma che causa errori di pagina ogni poche istruzioni). Se il WS viene caricato prima dell'esecuzione si ha prepaging (evitando page fault).

Anomalia di Belady: la frequenza di un page fault non sempre decresce al crescere dell'ampiezza della RAM. Gli **stack algorithms** sono immuni all'anomalia poiché soddisfano tale proprietà:

$M(m, r) \subseteq M(m+1, r)$

dove m rappresenta il numero di page frame, mentre r sono i riferimenti. La proprietà dice: assumendo gli stessi riferimenti, le pagine caricate con m page frame sono un

sottoinsieme di quelle caricate con m+1 page frame. LRU e Optimal Replacement sono stack algorithms.

Per rimpiazzare una pagina occorre scegliere tra:

- **Politiche locali:** rimpiazzo nel WS del processo che ha causato il page fault
- **Politiche globali:** la scelta avviene tra page frame senza distinzione di processo

Le politiche globali sono più efficienti con prestazioni superiori.

Vi sono due scelte di grandezza delle pagine:

- **Pagine piccole:** maggiore ampiezza della tabella delle pagine
- **Pagine ampie:** maggiore rischio di frammentazione interna (in media ogni processo lascia inutilizzata metà del suo ultimo page frame)

Il valore ottimo può essere definito:

spreco per processo: $f(\pi)$: $(\sigma/\pi) \times \xi + \pi/2$

derivata prima: $(\sigma \times \xi)/ \pi^2 +1/2$

ponendola uguale a zero si ha che il minimo di $f(\pi)$ si ha per: $\pi = (2\sigma\xi)^{1/2}$

dove:

- σ B dimensione media di un processo
- π B dimensione media di pagina
- ξ B per riga in tabella delle pagine

Trattamento di page fault:

1. l'hw fa un trap al kernel e salva il PC sullo stack
2. un programma assembler salva i dati nei registri e poi chiama il sistema operativo
3. il SO scopre il page fault e cerca di capire quale pagina lo abbia causato
4. ottenuto l'indirizzo virtuale causa del page fault, il SO verifica che si tratti di indirizzo valido e cerca page frame rimpiazzabile
5. se il page frame è dirty, si imposta il suo spostamento su disco
6. quando il page frame è libero, vi copia la pagina richiesta
7. all'arrivo dell'interrupt del disco, la page table è aggiornata e il frame è indicato come normale
8. il PC viene reimpostato per puntare all'istruzione causa page fault
9. il processo causa del page fault è pronto per l'esecuzione e il SO ritorna al programma assembler che lo aveva chiamato
10. il programma assembler ricarica i registri e altre info; poi torna in user space per continuare l'esecuzione

Area di Swap: area del disco riservata per ospitare le pagine temporaneamente rimpiazzate. Ogni processo ne riceve in dote una frazione: i puntatori a questa zona devono essere mantenuti nella tabella delle pagine del processo. 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. I processi non hanno dimensione costante perciò l'area è frazionata per codice e per dati.

La tecnica di paginazione può causare **frammentazione interna**.

Segmentazione

Spazi di indirizzamento completamente indipendenti gli uni dagli altri per dimensione e posizione in RAM. Rappresentano entità logiche note al programmatore e destinate a contenere informazioni coese (codice di procedure, dati di inizializzazione di un processo, stack di processo…). La tecnica si presta a schemi di protezione specifica ma causa **frammentazione esterna.** Consente di separare e distinguere tra codice e dati, di gestire contenuti di dimensione variabile nel tempo, di condividere parti di programmi tra processi. L'obbiettivo principale è la separazione logica tra aree dei processi e la loro protezione specifica. I segmenti possono essere paginati.

LDT(Local descriptor Table): descrive i segmenti del processo

GDT(Global Descriptor Table): descrive i segmenti del SO

File System

Il file system è il servizio di SO progettato per soddisfare 3 esigenze principali:

- Persistenza dei dati
- Possibilità di condividere dati tra applicazioni distinte
- Nessun limite di dimensione fissato a priori

Il termine file system designa la parte di SO che si occupa di: organizzazione, gestione, realizzazione, accesso di e ai file. All'utente applicativo deve offrire:

- Modalità di accesso ai file
- Struttura logica e fisica dei file
- Operazioni ammissibili su file

Ciò deve essere realizzato in modo pratico ed economico garantendo la massima indipendenza dall'architettura fisica.

File: designa un insieme di dati correlati, residenti in memoria secondaria e trattati unitariamente. 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. Le caratteristiche distintive di un file sono:

- **Attributi:** nome, dimensione corrente, data di creazione, data di ultima modifica, creatore e possessore, permessi di accesso.

- **Struttura dei dati:** la struttura dei dati all'interno di un file può essere considerata da tre punti di vista:
  1. **Livello utente:** il programma applicativo associa autonomamente significato al contenuto grezzo del file.
  2. **Livello di struttura logica:** il SO organizza i dati grezzi in strutture logiche per facilitarne il trattamento. Le possibili strutture logiche di un file sono:
     - sequenze di byte: 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: l'accesso ai dati è sequenziale e utilizza un puntatore al record corrente. Lettura e scrittura operano su record singoli. Il SO deve conoscere la struttura interna del file.
     - record di lunghezza e struttura variabile: la struttura di ogni record viene descritta e identificata univocamente da una chiave posta in posizione fissa e nota entro il record. Le chiavi vengono raccolte in una tabella a se stante, ordinata per chiave, contente anche i puntatori all'inizio di ciascun record. L'accesso ai dati avviene per chiave.
  3. **Livello di struttura fisica:** il SO mappa le strutture logiche sulle strutture fisiche della memoria secondaria disponibile.

- **Operazioni ammesse:** creazione, apertura, cerca posizione, cambia nome, distruzione, chiusura, lettura/scrittura, trova attributi, modifica attributi. Azioni più complesse si ottengono tramite combinazione di operazioni di base. Si può accedere in uso solo ad un file già aperto. Dopo l'uso il file dovrà essere chiuso.

Modalità di accesso:

- **Accesso sequenziale:** viene trattato un gruppo di byte alla volta. Un puntatore indirizza il record corrente e avanza a ogni lettura o scrittura (la lettura può avvenire in qualunque posizione del file, la quale però deve essere raggiunta sequenzialmente, mentre la scrittura può avvenire solo in coda al file). 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.
- **Accesso indicizzato:** per ogni file una tabella di chiavi ordinate contenenti gli offset dei rispettivi record nel file (informazioni di navigazione non più nei record ma in una struttura a parte ad accesso veloce).

Il FS può trattare diversi tipi di file:

- **file regolari:** sui quali l'utente può operare normalmente
- **file catalogo:** 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 o a blocco

Il SO può mappare un file in memoria virtuale: all'indirizzo di ogni suo dato in memoria secondaria corrisponde un indirizzo in memoria virtuale. Le operazioni su file avvengono in memoria principale:

chiamata di indirizzo □ page fault □ caricamento □ operazione □ rimpiazzo di pagina □ salvataggio in memoria secondaria

Directory: ogni FS usa directory o folder per tener traccia dei suoi file regolari. Le directory possono essere classificate rispetto all'organizzazione di file che esse consentono:

- **A livello singolo:** tutti i file sono elencati su un'unica lista lineare ciascuno con un proprio unico nome
- **A due livelli:** una Root Directory contiene una User File Directory per ciascun utente di sistema. L'utente registrato può vedere solo la propria UFD. Efficiente nella ricerca. Libertà di denominazione (ma non di riferimenti multipli allo stesso file). Non dà libertà di raggruppamento.
- **Ad albero:** numero arbitrario di livelli. Il livello superiore viene detto radice. Ogni directory può contenere file regolari o directory di livello inferiore. Il cammino può essere assoluto (espresso rispetto alla radice) o relativo (espresso rispetto alla posizione corrente). Ricerca efficiente. Libertà di denominazione (ma non di riferimenti multipli allo stesso file). Libertà di raggruppamento.
- **A grafo:** l'albero diventa grafo consentendo allo stesso file di appartenere simultaneamente a più directory. La forma generalizzata consente riferimenti ciclici e dunque riferimenti circolari (multigrafo poiché consente "spigoli paralleli").

Hard Link: un puntatore diretto ad un file regolare viene inserito in una directory ad esso remota, la quale però deve risiedere nello stesso FS del file. Questo crea due vie d'accesso distinte ad uno stesso file. Più efficiente a livello prestazionale.

Symbolic 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. Mantiene una sola via d'accesso al file originario. Possono collegare file anche su macchine remote, varcando i confini dei dischi.

I file system sono memorizzati su disco. I dischi possono essere partizionati perciò ogni partizione può contenere un suo FS distinto.

L'unità informativa su disco è il settore, i quali però vengono letti e scritti a blocchi: 1 blocco = N settori.

A livello fisico un file è un insieme di blocchi su disco. Vi sono tre strategie di allocazione di blocchi a file:

- **Allocazione contigua:** memorizza i file su blocchi consecutivi. Ogni file è decritto dall'indirizzo del suo primo blocco e dal numero di blocchi utilizzati. Consente sia accesso sequenziale che diretto. Vi è rischio di frammentazione esterna.
- **Allocazione a lista concatenata:** file come lista concatenata di blocchi. File identificato dal puntatore al suo primo blocco. Ciascun blocco deve contenere il puntatore al blocco successivo (o un marcatore di fine lista). Consentito sia accesso sequenziale che diretto (ma lento).
- **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. Non causa frammentazione esterna. Consente accesso sequenziale e diretto. Non richiede di conoscere preventivamente la dimensione massima di ogni nuovo file. Due strategie di organizzazione:
  1. **FAT** (forma tabulare): tabella ordinata di puntatori. Un puntatore per ogni blocco del disco. La porzione di FAT relativa ai file in uso deve sempre risiedere interamente in RAM: consente accesso diretto ai dati seguendo sequenzialmente i collegamenti. Un file è una catena di indici.
  2. **Nodi indice** (forma indicizzata): una struttura indice (i-node) per ogni 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. Un i-node contiene un numero limitato di puntatori a blocchi. Per i file di piccola dimensione gli indirizzi dei blocchi dei dati sono ampiamente contenuti in un singolo i-node. Per i file di dimensione maggiore esistono molteplici livelli di indirezione: un campo dell'i-node principale punta a un livello di blocchi i-node intermedi che a loro volta possono puntare ai blocchi dati o ad un successivo livello di blocchi i-node, e così via.

Gestione dei file condivisi:

- Per ogni file condiviso porre nella directory remota un symbolic link verso il file originale. Esiste così un solo descrittore 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 ma 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 intenzione di cancellarlo.

Gestione dei blocchi liberi:

- Vettori 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.

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 ad un falso file che occupi tutti i blocchi danneggiati.

Salvataggio del FS:

- Su nastro.
- Su disco: con partizione di back-up oppure mediante architettura RAID.

La directory fornisce informazioni su: nome, collocazione, attributi di file appartenenti a quel particolare catalogo. File e directory risiedono in aree logiche distinte.

La ricerca di un file correla il nome alle informazioni necessarie all'accesso: nome e directory di appartenenza del file sono determinati dal percorso indicato dalla richiesta. La ricerca lineare in directory è di realizzazione facile ma di esecuzione onerosa. La ricerca mediante tabelle hash è più complessa ma più veloce:

F(nome) = posizione in tabella puntatore al file

Si può anche creare in RAM una cache di supporto alla ricerca.