Appunti di sistemi operativi

Carlo Rosso

16 marzo 2022

Contents

1 Introduzione

Utilizzare un PC al pieno delle sue capacità è molto complesso. Aggiungiamo anche una stampante e varie periferiche, diventa difficile sfruttare le capacità della macchina. Per quest motivo i PC hanno un sistema operativo che in base al punto di vista si occupa di due questioni: **aumenta l'astrazione della macchina**, quindi, per quanto riguarda l'utente, sono necessarie meno azioni per compiere lo stesso lavoro, in aggiunta è più semplice ragionare ed immedesimarsi nella macchina. Fondamentalmente diventa molto più semplice usare l'elaboratore e programmare. D'altro canto, il sistema operativo si occupa di **amministrare le risorse del pc**, per cui spartisce le risorse tra i processi e ne gestisce le comunicazioni.

1.1 Astrazione

L'OS aumenta l'astrazine degli elaboratori, per cui l'utente interagisce con lo shell, che è text based. Oppure grazie al sistema operativo viene implementata un'interfaccia grafica, GUI, graphical user interface, grazie alla quale non c'è bisogno di studiare a fondo il funzionamento di un elaboratore. Il livello software è diviso in kernel mode e in user mode. Il sistema operativo si trova nel kernel, mentre la GUI e lo shell si trovano nella user mode. Aumentando l'astrazione, sopra la GUI si trovano le icone che aprono i programmi dell'utente. Ad un livello di astrazione più basso sono disponibili un maggior numero di istruzioni, tuttavia è più complesso utilizzare l'elaboratore. E.g. a livello GUI utente è sufficiente un doppio click per aprire un file; invece, usando il terminale bisogna scrivere open seguito dal nome del file da aprire.
L'astrazione maggiore permette ai programmatori che scrivono programmi (ma non sistemi operativi) di utilizzare istruzioni più generali e astratte, senza aver bisogno di conoscere i dispositivi di input/output e i dettagli dell'architettura sopra cui vogliono eseguire il programma. Questo vuol dire che il lavoro del programmatore viene semplificato molto a discapito delle prestazioni.
Fondamentalmente si comporta da interprete tra programmi e la macchina e tra la macchina e l'utente.

1.2 Gestione delle risorse

L'OS gestisce le risorse presenti in un elaboratore. Una risorsa è tutto ciò che è necessario ad un processo per essere portato a compimento (e.g. la CPU, la memoria e le periferiche). Ancora una volta il lavoro dei programmatori viene semplificato a discapito delle prestazioni. Non solo, ma in questo modo è permessa una maggior compatibilità di un programma su piattaforme differenti.

Un sistema operativo odierno conta almeno cinque milioni di righe di codice. Scrivere un sistema operativo è un processo molto lungo, e può richiedere anni. Per questo motivo, una volta scritto il primo sistema operativo si parte da quello e viene aggiornato oppure sono proposte altre versione dello stesso, con qualche variante. Il sistema operativo è formato da processi nascosti all'utente.

1.3 Processi

Un processo è un'astrazione del sistema operativo. Un processo è un programma in esecuzione ed è composto da tutti i dati che utilizza, che siano istruzioni, dati di input o variabili locali. Anche la memoria posta a disposizione del processo è una porzione del processo stesso. I processi svolti negli elabolatori odierni si scambiano l'accesso alle varie risorse costantemente, questo riguarda anche la CPU. Il sistema operativo gestisce i processi e costruisce una tabella che racchiude tutte le informazioni di ciascun processo per riconoscerli. Il PCB, **process control block** è come una carta d'identità per i processi. Il PCB contiene informazioni come l'address space, la prossima istruzione da eseguire, gli eventuali processi figli e il processo padre, ...
Molto spesso un processo attiva altri processi per portare a compimento il proprio lavoro. Il sistema operativo si occupa anche di gestire le comunicazioni tra i processi.

Address space

Quando un processo viene creato, viene creata anche la macchina virtuale sopra cui viene eseguito il processo. La macchina virtuale possiede memoria infinita. Il sistema operativo si occupa di mappare la memoria della macchina virtuale su una porzione della RAM (memoria principale). Se un processo ha bisogno di più memoria di quella che il sistema operativo prevedeva o viene assegnata una maggior quantità di memoria al processo oppure alcuni dati sono trasferiti dalla RAM alla memoria secondaria e il sistema operativo si occupa del trasferimento dei dati in base al bisogno e all'architettura del PC.

1.4 Directories

Per gestire i dati l'OS li visualizza all'utente come se fossero file. I file sono raggrupati in directory, folder o cartelle. L'organizzazione dei file e delle directory è ad albero. Ogni file è dotato di un proprio path, univoco. I processi hanno una working directory, ovvero la directory che ha il ruolo di root directory per quel processo. Quando si connette una memoria esterna al PC le directory della memoria esterna sono inserite dentro predeterminate directory della memoria secondaria (solo su UNIX, solo l'OS accede alla memoria principale).

UNIX tratta le periferiche come fossero file: sono utilizzate le medesime chiamate di sistema e l'architettura viene semplificata. Esistono due tipologie di special files:

- **block special file** le periferiche che hanno accesso casuale rientrano in questa categoria (per esempio i CD, HDD e le USB), per cui la loro memoria è divisa in blocchi. Un'altra volta, questa divisione è virtuale e operata dal sistema operativo;
- **character special file** le periferiche che comunicano attraverso una character stream rientrano in questa categoria (per esempio le stampanti o il modem).

In genere le periferiche sono contenute nella directory /dev. UNIX cerca di trattare qualunque cosa come se fosse un file. Per questo motivo anche le comunicazioni tra i processi sono dei file, in particolare si chiamano pipe. Un processo scrive una pipe come se fosse un file di output. Un processo legge una pipe come se fosse un file di input. Sono necessari dei provvedimenti per permettere a due processi di comunicare.

2 Processo

Virtualmente ogni processo ha una propria CPU dedicata. In realtà, il dispatcher effettua il cambio di contesto molto velocemente e prerilascia i processi. In un intervallo di tempo più o meno lungo più processi avanzano. In un dato istante viene eseguito solo un processo per volta.

2.1 Il ciclo di un processo

Un processo può essere creato per 4 motivi:

- Inizializzazione del sistema: l'elaboratore crea il primo processo dopo il boot del sistema;
- Un processo padre effettua una chiamata di sistema per un processo figlio;
- l'utente richiede di creare un nuovo processo;
- Initiation of a batch job.

Tutti i processi che rimangono in background aspettando un segnale dall'esterno si chiamano daemons e.g. tutti i processi che controllano l'arrivo di messaggi, mail, ...

In UNIX è possibile creare un processo solo attraverso il comando fork, che affettua una chiamata di sistema e crea un clone del processo che l'ha invocata. Il processo figlio cambia i propri descrittori (don't know what that means) e esegue un execve (o simili), in questo modo cambia i dati all'interno del proprio PCB e dunque le proprie caratteristiche e diventa un processo figlio.

Ci sono 4 cause che portano al termine di un processo:

- Normal exit: il processo esegue una chiamata di sistema;
- Error exit: il processo stesso si accorge di un errore ed effettua una chiamata di sistema;
- Fatal occur: il processo prova a fare qualcosa di non permesso, e.g. divisione per 0, accedere all'address space di un altro processo;
- viene eseguito il comando kill da parte di un altro processo.

In UNIX tutti i processi appartengono ad una gerarchia. Effettiamo il boot del sistema, init viene caricato nella RAM e crea tanti processi figli quanti sono i terminali che trova. Come abbiamo visto, per creare un processo si utilizza il comando fork e si tratta dell'unico modo per creare un processo (boot del sistema a parte), per cui ogni processo deve avere un processo padre.

2.2 Stati

Un processo si può trovare in 3 stati:

- **Pronto**: il processo è nella lista dei pronti e aspetta di accedere alla CPU;
- **Esecuzione**: il processo ha accesso alla CPU;
- **Attesa**: il processo ha bisogno di accedere a qualche risorsa per proseguire, CPU a parte.

Il sistema operativo è composto dallo **scheduler** e da altri processi nascosti all'utente. Lo scheduler decide l'ordine di esecuzione dei processi che si trovano tra i pronti e ne regola i tempi di esecuzione. Lo scheduler crea i processi del sistema operativo che spostano i processi utente nei vari stati. E.g. un processo ha esaurito il proprio tempo nella CPU, allora lo scheduler esegue una chiamata di sistema e lo prerilascia. Tutti i processi hanno bisogno del PCB, process control unit. Nel PCB sono contenute tutte le informazioni per eseguire il cambio del contesto più altre informazioni utili allo scheduler per ordinare l'esecuzione dei processi.

2.3 Comunicazione tra processi

Ci sono 4 relazioni che i processi possono avere tra loro:

- $P_1 \rightarrow \text{pipe} \rightarrow P_2$;
- $P_1 \rightarrow R \leftarrow P_2$;
- $P_1 \leftrightarrow P_2$;
- caso banale: nessuna comunicazione tra i processi.

Due processi potrebbero accedere alle medesime celle di memoria. In alcuni OS oppure in alcune architetture esiste una porzione di memoria condivisa tra due o più processi. La **race condition** si verifica quando due o più processi leggono o scrivono la medesima porzione di memoria e il risultato finare è arbitrario, ovvero dipende da quale dei due processi è uscito dalla regione critica per primo. La **regione critica** è la porzione di programma in cui si accede alla memoria condivisa. La soluzione alla race condition è la mutual exclusion, se un processo utilizza una porzione di memoria condivisa gli altri devono aspettare. Bisogna mantenere 4 condizioni per evitare la race condition:

- un solo processo per volta può trovarsi nella regione critica;
- non bisogna supporre caratteristiche architetturali (n. core, velocità);
- nessun processo fuori dalla regione critica ne deve bloccare altri;
- nessun processo deve mantenere lo stato di attesa per sempre (perchè non riesce ad entrare nella regione critica);

Disabilitare gli interrupt

Una soluzione possibile consiste nel disabilitare gli interrupt: quando un processo entra nella propria regione critica disabilita gli interrupt, ovvero non può essere prerilasciato. La race condition viene risolta solo per quelle architetture che sono dotate di un singolo core. In aggiunta, è pericoloso lasciare uno strumento del genere in mano all'utente, un processo utente potrebbe non riabilitare gli interrupt.

Lock variables

$0 \rightarrow$ nessun processo è nella regione critica;
$1 \rightarrow$ un processo è nella regione critica;
La race condition potrebbe verificarsi sulla variabile lock, per cui due processi entrano nella regione critica e quindi potrebbe verificarsi la race condition sulla variabile "importante".

Strict Alternation

La variabile *turn* viene impostata su un valore (0). Solo il processo 0 può accedere alla regione critica se *turn* == 0. Il processo 0 entra nella regione critica e cambia *turn* = 1. Solo il processo 1 può entrare nella regione critica se il valore di *turn* = 1. Il processo 1 entra nella regione critica e cambia *turn* = 0 ...

Se il processo 0 è l'ultimo processo ad aver effettuato l'accesso alla regione critica e ha bisogno di effetture nuovamente l'accesso alla regione critica si verifica la **busy wait**, per cui fino a che il processo 1 non entra nella regione critica il processo 0 continua ad verificare *turn*. Si perde in prestazioni. In più se il processo 0 ha una priorità più alta del processo 1, allora la CPU risulta bloccata: si continua ad eseguire cicli while.

Peterson's solution

Ci sono due processi che sono associati a 0 e 1.
Il processo 0 vuole entrare nella regione critica;
Chiama la funzione enter_region();
Imposta interested[0] = TRUE, ovvero il processo 0 dichiara di voler entrare;
Imposta turn = 0, ovvero pone la bandiera;
Controlla se il processo 1 vuole entrare;
Se il processo 1 non vuole entrare, allora il processo 0 entra;
Il processo 0 non necessita più l'accesso alla regione critica;
Imposta interested[0] = FALSE;

Se il processo 1 vuole entrare e ha già impostato interested[1] = TRUE, allora il processo 0 esegue cicli a vuoto ripartendo da capo fino a che non viene prerilasciato;
Processo 1 entra nella regione critica, se è stato prerilasciato dopo turn, altrimenti sarà il processo 0 ad entrare;
Che sia processo 1 o processo 0, uscendo impostano interested[process] = FALSE, e l'altro processo può entrare;
L'altro processo imposta interested[altro_processo] = FALSE;
O processo 0 o processo 1 possono entrare nella regione critical, si ricomincia da capo.

Qui sotto riporto la soluzione di Peterson in C (è il linguaggio più utilizzato per scrivere i sistemi operativi).

```
#define FALSE 0
#define TRUE  1
#define N     2                       // numero di processi

int turn;
int interested[N];                    // tutti i valori sono 0

void enter_region(int process) {
  int other;                          // numero dell'altro processo

  other = 1 - process;                // l'opposto del processo
  interested[process] = TRUE;         // dichiara di essere interessato
  turn = process;                     // pone la bandiera
  while (turn == process && interested[other] == TRUE) // ciclo infinito
}

void leave_region(int process) {
  interested[process] = FALSE;        // indica l'uscita dalla regione critica
}
```

Nella strict alternation è permesso solo $A \to B \to A \to \ldots$, non è possibile $A \to A \to \ldots$); con la soluzione proposta da Peterson è possibile anche il secondo caso. La busy wait viene evitata: lo stesso processo può entrare due volte.

2.4 Problema del produttore e del consumatore

Ora andiamo ad approfondire soluzioni che richiedono l'implementazione hardware. L'implementazione hardware consiste nell'aggiungere un'istruzione TLS oppure EXCHG. Questa istruzione preleva il contenuto di una cella di memoria chiamata *lock* ed è spostato in una cella di memoria *RX*. Nello stesso momento nel registro è posto un valore non nullo. Nel caso di EXCHG il contenuto di *lock* è invertito con il contenuto di *RX*. Questa istruzione è indivisibile, è atomica. Non solo le interruzioni sono disabilitate, anche il bus viene bloccato e solo la CPU che sta eseguendo questa operazione può accedere alla memoria. Questa istruzione evita la race condition, ma se non è ben gestita incappa nella busy wait. Per esempio poniamo che un procello L (low priority) entri nella regione critica. Un processo H (high priority) entra nella lista dei pronti e il processo L viene prerilasciato. Il processo H esegue un ciclo while infinito per accedere alla regione critica, ma è bloccato dal processo L che si trova nella lista dei pronti. Questo si chiama priority inversion problem.

Sono introdotte anche le system call: **sleep** e **wakeup**. Quando viene invocata sleep un processo bersaglio passa dallo stato di pronto allo stato attesa. Quando wakeup è invocato un processo passa dallo stato attesa allo stato pronto. Con queste due chiamate di sistema lo stato di un processo viene cambiato in modo forzoso e si evita la busy wait.

Analizziamo il problema del produttore e del consumatore. Ci sono due processi: produttore e consumatore. I due processi condividono un buffer di grandezza fissa. In produttore inserisce un dato nel primo posto libero. Il consumatore toglie un dato dal primo posto occupato. Se tutti i posti sono occupati il produttore invoca la chiamata di sistema sleep e viene posto in attesa. Se tutti i posti sono liberi il consumatore invoca la chiamata di sistema sleep ed è posto in attesa. In questo caso si sfruttano le chiamate di sistema sleep e wakeup.

N è una costante e indica il numero di posti disponibili. Il pointer count indica quanti posti sono occupati. Per cui se count è uguale a N allora il produttore va in attesa. On the other hand, se count è uguale a 0 il consumatore va in attesa. Quando il produttore, inserisce un oggetto aggiorna count (count += 1); quando il consumatore preleva un oggetto, aggiorna count (count -= 1). I due processi stanno attenti e effettuano la chiamata di sistema wakeup nei momenti opportuni.

Il problema della race condition può avvenire sul pointer count:

1. produttore controlla count (count = N);
2. produttore è prerilasciato;
3. consumatore controlla count (count != 0), procede;
4. consumatore toglie un elemento;
5. consumatore controlla count (count = N - 1);
6. chiamata di sistema: wakeup(produttore), produttore è nella lista dei pronti, per cui non succede nulla.
7. consumatore viene prerilasciato;
8. produttore si ricorda che (count = N) e invoca sleep(produttore), per cui va in attesa;
9. consumatore va in esecuzione e svuota tutto il buffer;
10. consumatore invoca sleep(consumatore) e va in attesa.

In questa caso entrambi i programmi sono in attesa. Il problema sussiste perché il segnale wake up viene perso. In realtà, non è sufficiente memorizzare il bit wakeup. Il problema non si risolverebbe se ci fossero più produttori e più consumatori. In questo caso si dice che wakeup non ha memoria.

Semafori

Il semaforo è una cella di memoria che memorizza un intero. In particolare se il valore del semaforo è 0, allora nessun processo può entrare nella regione critica, se il valore del semaforo è maggiore di 0 (> 0) allora uno o più processi possono entrare nella regione critica.
Sono introdotti due comandi: down e up. **down** controlla il valore del semaforo: se il semaforo è positivo (> 0) allora lo decrementa di 1 e il processo accede alla regione critica; altrimenti il semaforo viene decrementato di 1 e viene eseguita la chiamata di sistema sleep(processo). **Up** il semaforo è incrementato di 1. Se il valore del semaforo è negativo (< 0), allora viene effettuata la chiamata di sistema wakeup(processo) e un processo che deve accedere alla regione critica lo fa. Le chiamate di sistema sono atomiche, ovvero indivisibili, per cui un processo non può essere prerilasciato durante di esse: le interruzioni e il bus sono disattivati. Comunque si tratta di opearazioni molto veloci, per cui non si perdono prestazioni.

Sono introdotti i semafori *full*, *empty* e *mutex*. Il semaforo full indica il numero di posti occupati. Il semaforo empty indica il numero di posti vuoti. Il semaforo mutex si assicura che produttore e consumatore non entrino nello stesso momento. I semafori sono chiamati binari se sono inizializzati a 1. Il codice implementato è il seguente:

```
#define N 100                   // n. di posti nel buffer
typedef int semaphore;          // viene definito semaforo
semaphore mutex = 1;            // controlla gli accessi alla regione critica
semaphore empty = N;            // conta i posti liberi
semaphore full = 0;             // conta i posti occupati

void producer(void) {
        int item;

        while (TRUE) {
                item = produce();       // viene generato l'oggetto
                down(&empty);           // empty viene diminuito di 1
                down(&mutex);           // entra nella regione critica
                insert();               // viene inserito l'oggetto nel buffer
                up(mutex);              // lascia la regione critica
                up(&full);              // full viene incrementato di 1
        }
}

void consumer(void) {

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove();
                up(mutex);
                up(&empty);
                consume(item);
        }
}
```

Mettiamo di invertire down(&full) e down(&mutex). Il consumatore esegue down(&mutex), non c'è nessuno nella regione critica, per cui entra. Il consumatore esegue down(&full), non ci sono posti occupati, per cui il consumatore entra nello stato di attesa. Il produttore in questo caso non può accedere alla regione critica perchè mutex è stato cambiato, e quindi anche produttore è posto in attesa. Questa situazione si chiama deadlock.

Monitors

Un monitor è un insieme di procedure, variabili e strutture dati contenute in un modulo o pacchetto. I processi possono invocare le procedure dei monitor. I monitor non possono essere modificati dall'esterno del monitor stesso. Solo un processo per volta può essere attivo all'interno di un monitor.
Un processo che vuole entrare dentro un monitor chiama la procedura, la quale controlla se è libero. Se il monitor è libero allora il processo entra; se il monitor è occupato allora il processo è posto nello stato di attesa.
Il compilatore si occupa della mutual exclusion, non il programmatore, in questo modo si evitano errori e diventa più semplice programmare.

Quando un monitor si accorge che un processo non può più proseguire, allora effettua una chiamata di sistema: wait(). La wait() fa sì che il processo nel monitor esca dal monitor, sia posto in attesa ed effettui una chiamata di sistema: signal(), che pone il partner nella lista dei PRONTI (il consumatore se il processo posto in attesa è il produttore o viceversa). Il processo nella lista dei pronti ora può accedere al monitor.

Mettiamo che avvenga il contrario: il processo non riesce più a proseguire, allora sveglia il suo partner, qui viene prerilasciato. Il suo partner prova ad accedere al monitor, ma non ci risce, allora viene posto in attesa. Il processo iniziale torna in esecuzione e viene posto in attesa. Ora il monitor è vuoto e i processi che devono entrarci sono in ATTESA, abbiamo un deadlock.

2.5 Ordinamento dei processi

Precedentemente abbiamo incontrato la coda dei pronti: un processo ha bisogno di accedere solo alla CPU. Ora introduciamo due componenti hardware: il **dispatcher** effettua il cambio di contesto, mentre lo **scheduler** ordina la coda dei pronti. In breve, il dispatcher smantella la macchina virtuale del processo in esecuzione e costruisce quella del processo successivo. Dal momento che la coda dei pronti contiene un discreto numero di processi, tendenzialmente maggiore di 1 è necessario pensare ad un algoritmo che decida quale sia il prossimo processo ad andare in esecuzione, lo scheduler si occupa di applicare l'algoritmo.

Ci sono diversi metodi per misurare l'efficienza di un algoritmo:

- efficienza di utilizzo: tempo in cui un processo viene eseguito / tempo impiegato dal dispatcher e dallo scheduler per gestire i processi;
- throughtput: processi completati / unità di tempo;
- termpo di turn-around: per noi sarà costate tra i vari algoritmi di ordinamento, perchè non calcoliamo il tempo di gestione. Si tratta della somma del tempo di esecuzione e del tempo di attesa nella coda dei pronti dei processi;
- tempo di attesa: permanenza di un processo nella coda dei pronti;
- tempo di risposta: permanenza di un processo nella coda dei pronti fino alla prima esecuzione.

FCFS, first come first served

I processi hanno accesso al processore in base al tempo di arrivo nella coda dei pronti. Non è contemplato il prerilascio, per cui fino a che un processo non è completato continua ad essere eseguito.

RR, round robin

Ci sono diverse varianti di questo metodo. Partiamo dalla versione di base: in base all'arrivo nella coda dei pronti i processi entrano in esecuzione; ogni processo ha a disposizione un quanto di tempo che dipende dall'architettura. Esaurito il quanto di tempo il processo arrivato subito dopo quello attuale entra in esecuzione. Questo metodo riduce il tempo di risposta in modo significativo. Inoltre dà all'utente un'illusione di simultaneità, come se tutti i processi fossero eseguiti contemporaneamente.

Una variante prende in considerazione la priorità, per cui i processi sono divisi in varie code dei pronti in base alla priorità; fino a che la coda dei pronti dei processi con priorità maggiore non è vuota tutti gli

altri processi sono nella coda dei pronti e tutti i processi a priorità maggiore alternano l'accesso alla CPU in quanti di tempo.

Un'altra variante prevede quanti di tempo maggiori per processi a priorità maggiore, ma tutti i processi alternano l'accesso alla CPU. Questo metodo permette di diminuire il tempo di risposta.

Shortest Job First (shortest next-cpu-burst first)

In questo caso il processo che si stima verrà terminato per primo viene eseguito prima. Non è contemplato il prerilascio, per cui se un processo verrebbe completato più velocemente di quello che ha accesso alla CPU, resterà nella coda dei pronti. In questo caso il tempo di attesa viene ridotto molto.

Shortest Remaining Time next

Si tratta di un algoritmo di ordinamento molto simile al precedente, ma è possibile prerilasciare un processo se un nuovo processo entra nella coda dei pronti e si stima che venga completato in un tempo minore rispetto al tempo rimanente al processo in esecuzione. A parità di tempo di completamento il cambio di contesto non viene eseguito. Questo metodo minimizza il tempo di attesa.

3 Memoria Virtuale

Gli indirizzi calcolati nella CPU da un processo devono essere interpretati: questa azione prende il nome di *rilocazione*. Una volta che un processo calcola un indirizzo bisogna controllare che questo sia relativo o assoluto. Se è relativo allora ad esso bisogna aggiungere un offset, in modo tale che rientri nella porzione di memoria dedicata al processo.
Ottenuto l'indirizzo assoluto bisogna controllare che il processo abbia l'autorizzazione ad accedere alla cella di memoria indicata. Questa azione prende il nome di *protezione*.

3.1 Partizione ad hoc

La partizione ad hoc è un sistema di assegnamento della memoria a cura del sistema operativo che divide la RAM in base alla previsione di utilizzo della stessa. Questo sistema permette di diminuire la frammentazione interna. Per contro ha una frammentazione della memoria esterna marcata. Inoltre le dimensioni di un processo possono variare durante l'esecuzione dello stesso. Allora viene assegnato uno spazio di memoria con un margine.

3.2 RAM allocata dinamicamente

Dal momento che la RAM viene allocata dinamicamente, quindi mentre altri processi la usano nuovi processi la richiedono e vecchi processi terminano e la liberano. Il sistema operativo presta molta attenzione allo stato di utilizzo della memoria principale. Ci sono due tecniche per mappare le zone di memoria libere e quelle occupate:

- **Mappe dei bit**: un bit indica lo stato (0: libero; 1: occupato) di un'unità di allocazione; se le unità di allocazione sono molto grandi, sono sufficienti pochi bit per mappare la memoria, ma l'assegnamento della memoria è poco preciso e aumenta la frammentazione interna. D'altro canto se l'unità di allocazione è piccola, diminuisce la frammentazione interna, ma sono necessari molti bit per riuscire a mappare l'intera memoria principale;
- **Lista collegata** o mappa di allocazione: ogni volta che lo stato di una porzione di memoria cambia viene aggiornata la lista di una tabella che è composta da tre colonne: bit occupato/ libero, un puntatore alla cella di memoria che indica l'inizio della porzione di memoria e un unsigned che indica il numero di celle che compongono la regione in questione. Con questa tecnica la ricerca di una porzione libera delle dimensioni adatte è piuttosto veloce, ma in caso di frammentazione della memoria, la tabella diventa molto grande.

Molti programmi occupano più memoria di quella che mette a disposizione la RAM, per questo motivo sono necessarie delle tecniche per aggirare questo problema. All'inizio si utilizzava l'overlay, per cui un

programma era diviso in porzioni dal programmatore e veniva caricata solo una porzione per volta all'interno della RAM. Questo sistema risulta poco comodo e molto complesso. Per questo motivo si è introdotto un hardware ad hoc per risolvere il problema alla radice: la **memoria virtuale**.
Con l'introduzione della memoria virtuale i processi hanno a disposizione memoria infinita ed i dati sono portati in RAM solo quando sono utilizzati dal processo. La memoria virtuale può essere divisa in pagine e in segmenti. La paginazione divide la memoria virtuale in pagine da 4KB, in questo modo il sistema operativo si occupa di caricare in RAM solo le pagine che sono utilizzate. Alla paginazione si contrappone la segmentazione, per cui la memoria virtuale è divisa in base alle dimensioni del contenuto, si tratta di una divisione più elastica rispetto alla paginazione.
La **memory management unit** è la componente hardware che si occupa di mappare gli indirizzi: converte gli indirizzi relativi in indirizzi assoluti. Non solo, la MMU controlla se l'informazione a cui punta l'indirizzo calcolato è contenuta in RAM e se il processo ha l'autorizzazione ad accedere all'indirizzo.
La RAM è divisa in **page frame** ampie quanto le pagine. I trasferimenti tra RAM e memoria secondaria avvengono sempre nell'ordine delle pagine. Se una pagina è assente quando riferita si genera un evento del tipo **page fault** gestito dall'OS tramite comando trap. Se la RAM è piena viene effettuato lo **swapping**: porzioni di processo o interi processi sono portati dalla RAM alla memoria secondaria e viceversa per permettere un più efficace utilizzo della CPU. Il sistema operativo si occupa di gestire lo swapping.

3.3 La tabella delle pagine

La tabella delle pagine è l'astrazione di un array che serve all'MMU per mappare la memoria virtuale sulla RAM. Ad ogni riga della tabella delle pagine sono contenute le seguenti informazioni:

indice della protezione	cache abilitata/ disabilitata	referenziato	modificato	protezione	assente/ presente	numero frame

Table 1: suddivisione dei campi nella tabella delle pagine

L'indice della pagina è contenuto nell'indirizzo emesso dal processo e serve per accedere alla tabella delle pagine. Cache abilitata/ disabilitata è un bit che disattiva o abilita l'uso della cache: come abbiamo visto in ADE nel caso dell'I/O memory mapped le periferiche non supportano la cache. Il campo referenziato tiene traccia di quanto una pagina sia utilizzata. Il campo modificato è il corrispettivo del dirty bit dei blocchi nella politica di sostituizione write back. Il campo protezione è composto da 2 o 3 bit e rispettivamente indicano lettura, scrittura ed esecuzione. Il bit assente/presente indica una hit o una miss. Infine, il numero della page frame è l'indirizzo che viene sommato al offset per ottenere la cella di memoria.

Ogni volta che un processo emette un indirizzo, questo è mandato all'MMU. L'MMU isola il campo pagina dal campo offset, accede alla tabella delle pagine e vede se la pagina in questione è presente o meno in RAM. Dal momento che questi passaggi devono avvenire per ogni indirizzo emesso dai processi, l'accesso alla tabella delle pagine è molto veloce. Ogni processo ha la propria memoria virtuale, mappata sulla propria tabella delle pagine. Una tabella delle pagine è un array molto grande. Per questo motivo, non è possibile che ad ogni context switch la tabella delle pagine venga sostituita: la tabella delle pagine si trova in RAM, nella CPU si trova un registro che contiene un puntatore alla tabella delle pagine. In questo modo il context switch diventa veloce, però ogni volta che viene emesso un indirizzo da un processo bisogna accedere 2 volte alla RAM: la prima per accedere alla tabella delle pagine, la seconda per accedere al dato (sempre che sia contenuto nella memoria principale).
Per velocizzare la procedura è stato effettuato uno studio dei dati e si nota che si accede principalmente alle medesime pagine e raramente si accede a tutte quante le pagine. Per questo motivo è stato introdotto il **TLB**, **translation lookaside buffer**. Se la tabella delle pagine è la RAM, allora il TLB è la cache: il TLB è un piccolo buffer presente nella CPU che ha la medesima funzione della tabella delle pagine, ma memorizza solo le ultime pagine richieste. Quando è emesso un indirizzo da un processo si cerca contemporaneamente tra tutti gli elementi del TLB. Se l'indirizzo è presente nel TLB allora si verifica una hit, altrimenti si verifica un TLB miss. Se si verifica un TLB miss, nelle architetture RISC, il problema viene risolto a livello software e si accede alla tabella delle pagine. Se la pagina in questione è trovata nella tabella delle pagine, allora si dice che si è verificato una soft miss. Altrimenti si verifica una miss della tabella delle pagine; in questo caso si effettua una table walk, per cui si cerca la pagina tra le tabelle delle pagine degli altri processi. Altrimenti

la pagina in questione deve essere caricata dalla memoria. Se la pagina in quesitone non si trova in RAM si dice che si è verificato un hard miss. Solo a questo punto si controlla che l'indirizzo emesso sia valido, altrimenti si verifica un segmentation fault e il sistema operativo termina il processo. Notiamo che l'hard miss è milioni di volte più lento rispetto ad un soft miss.

3.4 La tabella delle pagine invertite

Dal momento che la memoria virtuale di ciascun processo è molto ampia è sconveniente mantenere la tabella delle pagine: occupa troppo spazio, anche se è veloce. La tabella delle pagine è sostituita dalla tabella delle pagine invertite che mappa i frame sulle pagine virtuali. In questo modo per verificare se una pagina si trova in RAM bisogna scorrere tutta la tabella delle pagine invertite. Questo procedimento è richiede molto tempo. Per cui la tabella delle pagine invertite è organizzata come una hash map: viene effettuato una sorta di direct mapping, per cui viene dato un valore che si può trovare solamente dentro una porzione della RAM. In questo modo la ricerca diventa circoscritta in una porzione decisamente minore.

3.5 Algoritmi di inserimento delle pagine

Nel momento in cui la RAM ha dello spazio libero, perchè il computer è appena stato acceso oppure perchè diversi processi sono terminati, ci sono vari algoritmi per scegliere dove inserire un nuovo processo.

First Fit

A seconda del metodo utilizzato per mappare gli spazi liberi della RAM, si comincia a scorrere la tabella e appena viene trovato uno spazio che è in grado di contenere il processo in questione, questo è inserito. Per ogni processo si ricomincia a scorrere la tabella dall'inizio. Si tratta dei un metodo piuttosto veloce, ma crea una forte frammentazione interna.

Next Fit

Per il primo assegnamento si comporta come il First Fit, però memorizza l'indice da cui ha cominciato a cercare e da lì riprende per il processo successivo. Quando arriva alla fine della tabella che riporta gli spazi liberi ricomincia da capo. Ovviamente se non c'è spazio libero il processo non parte oppure viene liberata della memoria. Si tratta di un sistema piuttosto veloce, ma crea una forte frammentazione interna.

Best Fit

Sono scorsi tutti gli spazi liberi e viene preso lo spazio più piccolo, ma che sia comunque sufficiente al processo ($\min(\text{segmenti\_liberi}) \neq \text{spazio\_necessario}$). Si tratta del sistema più lento, perchè in qualsiasi caso sono scorsi tutti gli spazi liberi.

Worst Fit

Il nuovo processo viene sempre inserito nello spazio libero più grande ($\max(\text{segmenti\_liberi})$). Si tratta del caso peggiore, perchè sono scorsi tutti gli spazi liberi, in più viene scelto lo spazio che crea più frammentazione interna.

Quick Fit

Gli spazi liberi sono raggruppati a seconda della loro dimensione e il nuovo processo è inserito nella prima regione di memoria libera della propria categoria. Si tratta di un sistema molto veloce, ma crea frammentazione interna.

3.6 Sostituzione delle pagine

Proprio come nel caso dei processi anche eseguibili anche le pagine non possono trovarsi tutte in RAM nello stesso momento. La soluzione ideale sarebbe conoscere il frame che viene richiesto più tardi in assoluto. Questo non è possibile, ma sono stati pensati diversi algoritmi per ovviare al problema.

Optimal

Non realizzabile: viene tolta la pagina che verrà richiesta più tardi.

Not Recently Used

Ad ogni frame sono associati due bit: M (modified) e R (referenced). Il bit M si comporta come un dirty bit, vd. ADE, per cui indica se un frame deve essere riscritto nella memoria secondaria nel momento in cui viene sostituito. Il bit R indica se entro un lasso di tempo un frame è stato utilizzato; il sistema operativo lo reimposta a 0 ogni unità di tempo. Avendo 2 bit i frame sono divisi in 4 classi:

- 0 non riferita, non modificata (R = 0, M = 0);
- 1 non riferita, modificata (R = 0, M = 1);
- 2 riferita, non modificara (R = 1, M = 0);
- 3 riferita, modificata (R = 1, M = 1);

Come ci si può immaginare, i frame sono tolti prima dalla classe 0, poi dalla classe 1 e così via.

FIFO

Come in molti altri casi, il primo frame che viene tolto dalla RAM è il primo frame che è stato inserito in RAM. Si tratta di un algoritmo molto semplice, ma scomodo: per esempio i frame del sistema operativo che indicano come interpretare gli input del mouse o della tastiera sono richiesti molto di frequente, per cui è scomodo applicare un algoritmo di questo tipo.

Second Chance

Ad ogni pagina è associato un bit R (referenced), se una pagina viene richiamata R viene impostato su 1. Se tutti i frame hanno come bit associato 1, allora viene impostato su 0; viene sostituito il frame inserito per primo nella RAM che abbia per R uno 0.

Orologio

Si comporta come Second Chance, i frame sono contenuti in una lista circolare.

Least Recently Used

Guarda la pagina chiamata prima per l'ultima volta e va a sostituire quella pagina. Adotta un approccio speculare rispetto alla optimal, che sarebbe la migliore. Necessita di un hardware ad hoc, per cui non è facilmente adottabile: aumenta la complessità architetturale.

Not Frequently Used

Ad ogni frame viene associato un contatore R (referenced), il quale conta viene incrementato per il numero di volte che un frame si trova in RAM. Sono sostituiti prima i frame con un R minore. Contro: se un frame viene chiamato molto all'inizio e poi non viene sostituito con grande difficoltà.

Aging

Ad ogni processo è associata una stringa di bit. Ogni unità di tempo il bit più a sinistra viene perso e viene aggiunto un bit a destra: 0 se nell'unità di tempo il frame non è stato riferito, 1 altrimenti. Si tratta di una buona approsimazione delle LFU (least frequently used).

Working Set

Un working set è l'insieme dei frame che sono utilizzati da un processo in un dato istante. Se la memoria non è sufficiente sono portati frame dalla RAM alla memoria secondaria e viceversa mentre il processo è in esecuzione, in questo caso il processo prende il nome di threshing. Il processo per cui il working set viene caricato in RAM prima che il processo vada in esecuzione è chiamato prepaging e previene eventuali page fault. $w(k,t)$ è l'insieme delle pagine che soddisfano i k riferimenti al tempo t. La funzione è monotonica crescente asintotica. Per cui è possibile arrivare ad un dato momento per cui non ci sono più page fault. Il working set apprssimato è molto simile all'aging.

3.7 L'anomalia di Belamy

Generalmente si pensa che aumentando le dimensioni della RAM si diminuiscano i page fault e quindi il computer migliori le prestazioni. In realtà, è stato dimostrato che questo non è vero. Per questo motivo si sono cercati algoritmi che non subiscono questa anomalia: il least recently used.

Le politiche per liberare memoria possono essere globali o locali:

- politiche locali: ogni processo conserva la porzione di memoria iniziale. Nel momento in cui si verifica una page fault viene tolto un frame tra i frame dello stesso processo. Si tratta di una politica poco conveniente;
- politiche globali: le porzioni della RAM sono dinamiche. Lo swap avviene tra page frame senza distinzioni a seconda del processo. Si tratta della politica più efficiente. Risulta comodo imporre il numero di frame minimo e massimo da dare ad un processo.

3.8 Dimensione ideale delle pagine

Lo spreco medio a causa della frammentazione interna è di circa mezzo frame. Ma allora qual'è lo spreco medio per ogni processo?

$$Spreco(\pi) = \frac{\sigma}{\pi} \cdot \epsilon + \frac{1}{2}\pi \tag{1}$$

La formula sopra riportata indica lo spreco medio di memoria di ciascun processo: σ indica le dimensioni medie di un processo, π indica le dimensioni di una pagina, ed in fatti si tratta dello spreco in funzione della dimensione delle pagine. per cui σ/π indica il numero di pagine di cui necessita un processo. ϵ insica le dimensioni di una riga nella tabella delle pagine: ad ogni pagina corrisponde una riga nella tabella delle pagine. Data la funzione semplicemente si effettua la derivata prima. Il risultato da sempre un numero negativo, ma agli informatici questo non interessa.

Lo swap nei linux è gestito in una partizione della memoria secondaria, per evitare la frammentazione, il che rende lo swap limitato. Al contrario su Windows lo swap è gestito da due file: hiberfil.sys e pagefile.sys. Il primo consiste in una copia della RAM in caso di ibernazione; il secondo riguarda la mancata capienza della RAM. In questo caso la frammentazione rallenta molto il computer.

3.9 Segmentazione

In questo caso gli spazi di indirizzmento sono completamente indimendenti gli uni dagli altri. Inoltre ogni segmento è dotato di dimensione propria. Con la segmentazione c'è un rischio notevole di frammentazione esterna. LDT è una tabella che descrive i segmenti propri di un unico processo. Il GDT è una tabella che descrive i segmenti della RAM. In tutto ci sono $2^{13} = 8k$ descrittori di un segmento. Questi 13 bit fungono da indice per leggere le informazioni del segmento. Nella tabella sono presenti due colonne, oltre all'indice: 1 bit per distinguere LDT da GDT e 2 bit per gestire il privilegio del descrittore.
Per accedere ad un dato si ha bisogno di un selettore e di un offset. Il selettore permette di trovare il descrittore del segmento. Un descrittore è composto da alcuni campi: è composto da una base, da un limite, dai bit di protezione e da un bit che indica se il descrittore è locale o globale. Il limite è confrontato con l'offset, se l'offset è maggiore allora è segnalato un segmentation fault. Altrimenti, la base è sommata all'offset e così viene ottenuto l'indirizzo assoluto. Si noti che la base è la base di inizio del segmento. Il limite è il limite

superiore del segmento. La segmentazione permette un semplice e prestante uso della memoria dinamica. Per riuscire ad utilizzare la segmentazione in modo efficiente è necessario implementare dell'hardware ad hoc.

4 File System

I dati prodotti dai processi e dai programmi generalmente hanno una durata, un ambito e una dimensione maggiore dei processi stessi, per cui devono persistere, essere condivisibili tra applicazioni diverse e non devono avere limite di dimensione. Un file è una raccolta di dati correlati trattati unitariamente; risiedono nella memoria secondaria. Ogni file è caratterizzato da alcuni attributi: nome, dimensione, data creazione, ultima modifica, creatore e possessore, permessi di accesso, ...
Un file può essere formato da:

- sequenza di byte: si tratta di una striga di bit. È una struttura semplice e flessibile. L'applicazione sa come usare i byte, il sistema operativo li può gestire come meglio crede. L'accesso è sequenziale tramite puntatore relativo ed il file è modificato per blocchi;
- record di lunghezza e struttura fissa: l'OS ha bisogno di conoscere la struttura del file, si tratta di una struttura inutilmente complessa: non viene più adoperata;
- record di lunghezza e di struttura variabile: l'OS ha bisogno di conoscere la struttura del file, comunque in alcuni sistemi questo comporta un'ottimizzazione della memoria. Comunque il sistema è poco flessibile. Viene utilizzato principalmente per i main frame.

Come abbiamo già visto in ADE, ci sono varie modalità di accesso ai file:

- sequenziale: i dati sono letti sempre partendo dall'inizio. Nella scrittura i dati sono aggiunti alla coda (vd. Append);
- diretto: ha dei check point a partire dai quali si presenta come l'accesso sequenziale. I checkpoint sono piuttosto vicini tra loro, per cui diventa semplice rimpiazzare un singolo blocco;
- indicizzato: ricerca binaria mista ad accesso diretto.

Su Unix i file possono appartenere a tre categorie: regolari, i file che un utente può decidere di aprire e modificare; catalogo: i folder, le cartelle; infine speciali: si tratta di file di input e output per interagire con altri dispositivi. Non è possibile toccare direttamente questi file: si tratta di tutto ciò che ci permette di interagire con dispositivi esterni al PC medesimo: stampante, fax, modem, ...
Windows è caratterizzato solo da file regolari e file catalogo.
Quando un file è aperto è portato in RAM e viene generata una handle per poter accedere ed eventualmente modificare il file. Dopodichè il file è chiuso e le eventuali modifiche sono riportate nella memoria secondaria.

Il settore 0 del disco contiene le informazioni di inizializzazione del sistema:

- master boot record;
- MBR, descrizione delle partizioni;
- il primo blocco contiene le sue specifiche informazioni di inizializzazione.

4.1 File

Un file è composto da un insieme di blocchi e dagli attributi del file stesso. Occorre decidere come mappare i blocchi:

- allocazione contigua: un file è memorizzato su blocchi contigui ed è descritto dall'indirizzo del primo blocco e dal numero di blocchi che lo compongono. Permette l'accesso sequenziale e diretto. Tuttavia come contro ha una forte frammentazione esterna;
- allocazione a lista concatenata: un file è memorizzato in una lista concatenata di blocchi. Per accedere al file è sufficiente il puntatore al primo blocco. Ogni blocco alla propria fine contiene un puntatore al blocco successivo. Se un solo blocco viene perso l'intero file è perso;

- allocazione a lista indicizzata: i puntatori ai blocchi si trovano in liste apposite. Ciascun blocco contiene solo dati. Evita la frammentazione esterna. Permette l'accesso sequenziale e diretto. La metodologia FAT (File allocation table) è un esempio dell'applicazione di questa tecnologia: 1 puntatore per blocco, all'incremento della partizione aumenta il numero di puntatori necessari per mappare l'intera memoria. Alla tecnologia FAT si contrappongono i nodi indice: una tabella formata dagli attributi del file e dai puntatori ai blocchi: c'è un numero fisso di puntatori a blocchi di dati. Oltre il quale ci sono puntatori a blocchi di puntatori e puntatori a blocchi di puntatori a blocchi di puntatori e così via. In questo caso ogni blocco è formato da 4kB, sia per quando riguarda i blocchi di dati che per gli i-node.
  Tra gli attributi è presente una variabile count che tiene conto del numero di hard-link: il numero di puntatori al descrittore. Quando il count dei hard-link va a 0 allora i blocchi del file sono liberati.

Proprio come i blocchi occupati dai file, anche i blocchi liberi hanno bisogno di essere mappati:

- bitmap: funziona come il mappaggio dei frame nella RAM;
- lista concatenata di blocchi sfruttando i campi puntatore al blocco successivo. Questa tecnica è utilizzata dalla tecnologia FAT.

4.2 Directory

Le directory contengono informazioni sul proprio nome e sui file che contengono: in particolare mantengono le informazioni sugli attribuiti dei file che contengono oppure al loro interno sono memorizzati il nome del file e un hard-link al descrittore del file.

Si parla di consistenza se ciascun blocco appartiene ad una sola lista dei blocchi: ci sono la lista dei blocchi occupati e la lista dei blocchi liberi. Se un blocco non appartiene a nessuna delle due liste allora si parla di perdita. Se un blocco è presente più volte nella stessa lista allora si parla di duplicazione.