APPUNTI SISTEMI OPERATIVI

Introduzione

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

Macchina Virtuale: ambiente virtuale (organizzato a pagine e/o segmenti) dove eseguire applicazioni, potente e semplice da usare.

Processo: è un programma in esecuzione formato da: -un insieme ordinato di **stati** assunti dal programma nel corso dell'esecuzione;
-un insieme ordinato delle **azioni** effettuate dal programma nel corso dell'esecuzione.

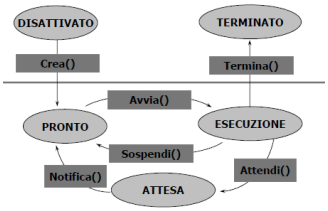
Lo spazio di indirizzamento logico è la memoria virtuale in cui il processo può leggere e scrivere il programma eseguibile, i dati del programma e le aree su cui può lavorare.

In un sistema coesistono processi utente e di SO che possono cooperare tra loro ma con privilegi diversi:

-Il SO assegna ai processi che avanzano concorrentemente le risorse necessarie all'esecuzione in base alle politiche di ordinamento (a divisione di tempo o a livello di prioritá);

-Il SO deve fornire i meccanismi e i servizi necessari ai processi che devono comunicare e sincronizzarsi tra loro;

Stati di avanzamento di processo



DISATTIVATO: il programma è in mem secondaria. Un supervisiore lo carica in memoria tramite una chiamata di sistema che crea una struttura di controllo di processo PCB (process control block);

PRONTO: il processo pronto per l'esecuzione rimane in attesa del suo turno;

ESECUZIONE: il processore esegue il processo selezionato.

ATTESA: il processo è sospeso in attesa di una risorsa non disponibile al momento;

TERMINATO: il processo ha concluso regolarmente le sue operazioni e abbandona la macchina virtuale.

Kernel: è il nucleo del SO. -Gestisce l'avanzamento dei processi scegliendo quale eseguire ad un dato istante tramite lo Scheduling(ordinamento),
-Gestisce le interruzioni esterne causate da eventi I/O o altre situazioni anomale;
-Consente ai processi di accedere a risorse di sistema e di attendere eventi.

La politica di ordinamento deve essere equa, i processi pronti ad eseguire devono avere l'opportunitá di farlo e processi in attesa di risorse devono avere l'opportunitá di accedervi. Inoltre i servizi di comunicazione e sincronizzazione devono essere efficaci, un dato inviato da un processo deve raggiungere il destinatario in un tempo breve e in modo sicuro.

Risorsa: è un qualsiasi elemento fisico(hardware) o logico(software) necessario alla creazione,esecuzione e avanzamento dei processi.
Possono essere ad accesso divisibile(se tollera l'alternanza con accessi di altri processi) o indivisibile(se non tollera alternanze durante l'uso), ad accesso individuale o molteplice(molteplicitá fisica o logica).

3 tipi: -**Risorsa CPU**: indispensabile per l'avanzamento dei processi. A livello harware corrisponde al processore, a livello software può essere vista come macchina virtuale offerta dal SO alle sue applicazioni.

-**Risorsa Memoria**: risorsa ad accesso multiplo per la lettura,ad accesso individuale per la scrittura. Viene virtualizzata dal software, utilizzandola assieme alla memoria secondaria attribuendone l'accesso secondo particolari politiche. E' consumabile e indivisibile,ma una volta virtualizzata diventa riutilizzabile e prerilasciabile.

-**Risorsa I/O**: riutilizzabile, non prerilasciabile e ad accesso individuale. Il software ne facilita l'impiego nascondendo le caratteristiche hardware e uniformando il trattamento. L'accesso fisico ha bisogno di utilizzare programmi specifici detti **BIOS**(Basic I/O System).

Caricamento del SO: -Il SO può risiedere permanentemente in ROM (in sistemi di controllo industriale e in sistemi dedicati);
-oppure nella memoria secondaria per essere caricato(tutto o in parte) in RAM tramite bootstrap all'attivazione del sistema(in ROM risiede solo il caricatore del sitema detto bootstrap loader).

SINCRONIZZAZIONE TRA PROCESSI:

Molti processi condividono risorse e informazioni funzionali, per gestire la loro condivisione servono meccanismi di sincronizzazione di accesso.

MUTUA ESCLUSIONE: la modalità di accesso indivisa ad una risorsa condivisa, avviene mediante una variabile logica "lock" che indica se la variabile condivisa è in uso da un altro processo, qui l'accesso consentito ad un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa.

```
/* i processi A e B devono accedere
** a X ma prima devono verificarne
*/ lo stato di libero
if (lock == 0) { // variabile "lucchetto"
   /* X è già in uso:
   */ occorre ritentare con while do
}
else {
// X è libera, allora va bloccata ...
   lock = 0;
   ... // uso della risorsa
// ... e nuovamente liberata dopo l'uso
   lock = 1;
}
```

⇒Una soluzione del genere al problema della sincronizzazione è inadeguata:

-Ciascuno dei due processi può essere prerilasciato dopo aver letto la variabile lock ma prima di essere riuscito a modificarla, ciò causa Race Condition e può generare inconsistenze;

-L'algoritmo causa inoltre una Busy Wait che causa spreco di tempo di CPU.

Una soluzione al problema della sincronizzazione tra processi è valida se:

-garantisce l'accesso esclusivo;
-gestisce un attesa finita;
-non fa assunzioni sull'ambiente di esecuzione;
-non subisce condizionamenti dai processi esterni alla sezione critica.

Tecniche Alternative:

-Disabilitazione interruzioni: previene prerilascio dovuto all'esaurimento del quanto di tempo e/o la promozione di processi a più elevata priorità. Può essere però innaccettabile per sistemi soggetti ad interruzioni frequenti ed è sconsigliabile lasciare controllo dell'interrupt agli utenti. E' inutile con due o più processori.

-Supporto Hardware diretto Test-and-Set-Lock: si può cambiare automaticamente il valore alla variabile di lock se questa segna "libero". Ciò evita situazioni di race contition ma comporta busy wait.

Situazione Critica: zona del programma che delimita l'accesso a l'uso du una variabile condivisa.

Problema Inversion Priority: dati due processi H (ad alta priorità) e L (a bassa priorità).

-Se H prerilascia il processore per esequire I/O e se H conclude le operazioni di I/O mentre L si trova nella sua situazione critica

⇒Allora H rimane bloccato in busy wait perché L non avrà più modo di concludere la sezione critica.

STRUTTURA SEMAFORO:

Consente forme più generali di sincronizzazione tra processi rispetto alla mutua esclusione. Qui si richiede un accesso indiviso alla variabile di controllo detta semaforo e per fare questo la struttura si appoggia sopra una macchina virtuale meno potente che fornisce una modalitá di accesso indiviso più primitiva.

```
void P (struct x)                    P
    if x.value = 1 //se libera
            x.value = 0
    sospendi(me, x.coda) //accodo
rilascia
```

```
void V (struct x)                    V
    x.value = 1 //rilascio
    if not_empty(x.coda)
            ready(get(x.codal)
rilascia
```

***ES**: Lavoro sulla stessa variabile*

```
PROC. A
<<esegui comando indipendente>>
P(x) //richiedo risorsa
<<eseguo comando in lock>>
V(x) //rilasico

PROC. B
P(x) //richiedo risorsa
<< se occupata aspetto >>
<< inizio >>
V(x) //rilascio dopo l'uso
```

-**Semaforo Binario**: struttura composta da un campo valore booleano(0="risorsa occupata" o 1="risorsa libera") e da un campo coda che accoda tutti i PCB(Process Control Block) dei processi in attesa sul semaforo. Due operazioni:

-Richiesta di accesso ad una risorsa P(): decrementa il contatore se questo non è già a 0, altrimenti accoda il chiamante;

-Avviso di rilascio di una risorsa V(): incrementa il contatore di 1 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 nel mezzo tra le chiamate P e V sul semaforo associato ad R.

```
void P (struct x)
    x.value -- //tolgo unità di risorsa
    if x.value < 0 //se unità insuff
    sospendi(me, x.coda) //accodo
rilascia
```

```
void V (struct x)
    x.value ++
    if x.value <=0  //se unità insuff
            ready(get(x.codal)
rilascia
```

***Es**: produttore e compratore*

```
#define N 100
Typedef int x
x mutex = 1
x n-v = 0
x n-p = N
PRODUTTORE
Int produci
    While (1)
        Produci:=prod()
        P(&n-p)          //fabbrica lock-on
        P(&mutex)        //lock-on
        Inserisci()
        V(&mutex)        //lock-off
        V(&n-v)          //magazzino lock-off

CONSUMATORE
Int prod
    While (1)
        P(&n-v)          //magazzino lock-on
        P(&mutex)        //lock-on
        Preleva()
        V(&mutex)        //lock-off
        V(&v-p)          //fabbrica lock-on
        Consuma()
```

-**Semaforo Contatore**: stessa struttura del binario ma usa una logica diversa per il campo valore. Consente l'accesso simultaneo di più processi ad una risorsa condivisa, fino al limite fissato dal valore di inizializzazione del contatore(che indica la capacità massima):

-Se il valore è >0 significa che la disponibilità non esaurita;

-Se <1 significa che ci sono richieste pendenti;

-Il valore iniziale indica la capacità massima della risorsa.

Rischio:

Usare semafori a livello di programma è rischioso, posizionamento improprio di P può causare situazioni di blocco infinito(deadlock) oppure esecuzioni errate difficili da verificare(race condition).

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

CONSUMATORE
procedure Prod
    begin
    while true do begin
        prod:=PC.preleva
        consuma(prod)
    end

MONITOR
Monitor
    condition: n-v, n-p
    int cont = 0

    procedure inserisci
        if cont = N
            wait(n-p)
        <<inserisci>>
        cont ++
        if cont = 1
            signal(n-v)

    procedure preleva
        if cont = 0
            wait(n-v)
        preleva:= <<preleva>>
        cont –
        if cont = N-1
            signal(n-p)
        end
```

-**Struttura Monitor**: strutture esplicite di controllo delle regioni critiche dette monitor, sono un aggregato di sottoprogrammi, variabili e strutture dati. Solo i sottoprogrammi del monitor possono accedere alle variabili interne e solo un processo per volta può essere attivo nel monitor(questo è garantito dai meccanismi di supporto a tempo di esecuzione, funzionamento simile al kernel).

Per consentire una sincronizzazione intelligente, si introducono 2 procedure operanti su variabili speciali (condition variables) che consentono di modellare condizioni logiche specifiche del problema:

-**Wait**(<condizione>):permette di bloccare il chiamante quando le condizioni logiche della risorsa non consentono l'esecuzione del servizio(ad es quando contenitore è pieno per il produttore e il contenitore è vuoto per il consumatore);

-**Signal**(<condizione>) risveglia il processo in attesa non appena la condizione diventa vera. Il processo risvegliato compete col Chiamante della Signal per il possesso della CPU. Il segnale di risveglio non ha memoria.

→Wait e Signal sono invocate in mutua esclusione, quindi non si può causare race condition.

In ambiente locale si hanno 3 possibilità per supportare la sincronizzazione tra processi:

-Con linguaggi concorrenti: 1)ad alto livello,con supporto esplicito a strutture monitor;

-Con linguaggi sequenziali senza supporto per monitor o semafori:

2)a basso livello,con uso di semafori tramite strutture primitive del SO e chiamate a sistema;

3)a bassissimo livello, con semafori primitivi in assembler, senza supporto del SO.

⇒Monitor e Semafori non sono utilizzabili per realizzare scambio di informazioni tra elaboratori.

Barriere: Per sincronizzare gruppi di processi si usano attività cooperative suddivise in fasi ordinate. Si utilizza una barriera che blocca tutti i processi che la raggiungono fino all'arrivo dell'ultimo. Questo non comporta uno scambio esplicito di messaggi, ma l'avvenuta sincronizzazione avverte implicitamente ai processi del gruppo che tutti hanno raggiunto un dato punto della loro esecuzione.

Tipiche situazioni di Stallo:

-**Deadlock**: stallo con blocco, cioè 2 o più processi si bloccano a vicenda aspettando che un altro esegua qualcosa;

-**Starvation**: stallo senza blocco, si ha un'attesa indefinita in quanto un processo è impossibilitato ad ottenere risorse di cui necessita per essere eseguito;

-**Race Condition**: esecuzioni non predicibili, cioè quando in un sistema basato su processi multipli, il risultato finale dell'esecuzione dipende dalla sequenza con cui i processi vengono eseguiti;

-**Busy Wait**: attesa attiva;

Problemi di Sincronizzazione:

-**Produttore-Consumatore**: due processi condividono un buffer comune e 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" finchè il produttore non inserisce qualcosa nel buffer e lo risveglia.

SOLUZIONE con Sleep and Wakeup

```
#define N100
int count =0;

void producer (void)
{
int item;
while (TRUE){
        item=produce_item();            produci l' oggetto
        if (count==N) sleep();          se buffer pieno,allora dormi
        insert_item(item);              metti l'oggetto nel buffer
        count=count+1;                  aumenta di +1 il contatore degli oggetti
        if (count==1) wakeup(consumer); se c'è almeno un oggetto nel buffer sveglia il consumatore
        }
}

void consumer (void)
{
int item
while(TRUE){
        if (count==0) sleep();          se il buffer è vuoto dormi
        item=remove_item();             altrimenti rimuovi un oggetto dal buffer
        count=count-1;                  decrementa il contatore degi oggetti del buffer
        if (count== N-1) wakeup(producer); se il buffer ha 99 elementi,sveglia il produttore
        consume_item(item);             stampa l' oggetto
        }
}
```

⇒Le Wakeup non vengono memorizzate

→quindi se c'è una sleep in attesa, le wakeup vengono perse.

⇒C'è rischio di race condition sulla variabile count, ad esempio:

-Se il buffer è vuoto e il consumer legge count=0

→allora prima che il consumatore istanzi la sleep, lo scheduler decide di fermare il consumer e di eseguire il producer.

→allora producer produce un elemento e imposta count=1.

-Se count=1 ⇒ il producer emette una wakeup che nessuno ascolta.

→allora ad un certo punto lo scheduler decide di eseguire di nuovo il consumer il quale istanzia la sleep, ma dato che count è gia ad 1 e la wakeup è già stata emessa, il consumer non verrà più risvegliato.

Soluzione con Semafori

```
#define N 100           #semafori nel buffer
typedef int semaphore;  inizializzo semaforo
semaphore mutex = 1;    variabile controlla accessi alla regione critica
semaphore empty = N;    count degli slop vuoti del buffer
semaphore full = 0;     count degli slot pieni del buffer

void producer (void)
{       int item;
        while(TRUE)
        {       item =produce_item(); produci nuovo oggetto da mettere nel buffer
                down(&empty);         decremento #slot vuoti
                down(&mutex);         entro nella regione critica
                insert_item(item);    metto il nuovo oggetto nel buffer
                up(&mutex);           lascio la situazione critica
                up(&full);            incremento #slot pieni
        }       }
void consumer(void)
{       int item;
        while(TRUE)                   loop infinito
        {       down(&full);          decremento #slot pieni
                down(&mutex);         entro nella regione critica
                item=remove(item);    tiro fuori l'oggetto dal buffer
                up(&mutex);           lascio regione critica
                up(&empty);           incremento #slop vuoti
                consume_item(item);   faccio qualcosa con l'oggetto selezionato
        }       }
```

⇒le variabili intere contano il #richieste pendenti, il valore è 0 se ci sono risorse disponibili a servire le richieste e >0 altrimenti.

2 operazioni: -**Down(x)**: -se x>0 allora x=x-1 ed il processo continua l'esecuzione;

-se x=0 allora il processo si blocca.

-**Up(x)**: -se ci sono processi in attesa di completare Down sul semaforo ed x=0 → uno di questi viene svegliato e x rimane a 0. Altrimento x=x+1;

-in caso contrario, cioè x>0 → x=x+1;

-**Filosofi a Cena**: Esso simula l'accesso esclusivo a risorse condivise da parte dei processi. N filosofi sono seduti in un tavolo circolare con un piatto di spaghetti per ciascuno e una posata alla propria sinistra. Ogni filosofo per mangiare ha bisogno di 2 posate, quindi intervalla momenti di riflessione e momenti in cui mangia.

Soluzione A

⇒Ogni forchetta è modellata come un semaforo Binario a mutua esclusione per dare accessi ad entrambe le forchette (funzionamento garantito)

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

→Causa stallo deadlock, in quanto l'accesso alla prima forchetta non garantisce accesso alla seconda.

Soluzione B

Ciascun processo attende un tempo casuale⇐incece che fisso (funzionamento non garantito)

→Causa 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]);                    }
```

Soluzione con Monitor

```
Monitor Tavolo{
        boolean fork_used[5] = false;    forchette numerate da 0 a 4
        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);
        }
}
```

Soluzione con 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]);
        }       }
```

⇐Per evitare deadlock del sistema utilizziamo l'algoritmo Del "filosofo mancino".

-**Lettori e Scrittori**: un insieme di processi deve leggere e scrivere in un database condiviso. Più lettori possono accedere contemporaneamente al database mentre gli scrittori devono averne l'accesso esclusivo. I lettori hanno precedenza sugli scrittori.

Soluzione con Semaforo

```
typedef int semaphore;
semaphore mutex = 1;        controlla accesso a rc
semaphore db = 1;           controlla accesso al database
int rc = 0;                 #processi che vogliono legger

void reader(void){
        while(TRUE){                    ripeti all'infinito
                down(&mutex);           ottieni accesso esclusivo ad rc
                rc = rc+1;              incremento #lettori
                if(rc == 1) down(&db);  se rc è il primo lettore db=0
                up(&mutex);             rilascio accesso esclusivo ad rc
                read_data_base();       accedo al database
                down(&mutex);           ottieni accesso esclusivo ad rc
                rc = rc-1;              decremento #lettori
                if(rc == 0) up(&db);    se rc è l'ultimo lettore db=1
                up(&mutex);             rilascio accesso esclusivo ad rc
                use_data_read();        esco dalla regine critica
        }       }
void writer(void){
        while(TRUE){                    ripeto all'infinito
                think_up_data();        entro nella regione non critica
                down(&db);              ottengo accesso esclusivo al db
                write_data_base();      scrivo nel db
                up(&db);                ottieni accesso esclusivo al db
        }       }
```

-**Barbiere che dorme**: in un negozio gestito da un solo barbiere è dotato di una poltrona da barbiere e di un certo numero di sedie per i clienti in Attesa. In assenza di clienti il barbiere si siede sulla sedia da lavoro e dorme. Se arrivano clienti il barbiere li serve in ordine di arrrivo. I clienti entrano nel negozio e se ci sono posti a sedere liberi si siedono, altrimenti lasciano il negozio.

Soluzione con Semafori

```
Define CHAIRS 5             #sedie per i clienti
typedef int semaphore;
semaphore customers=0;      #clienti che aspettano il servizio
semaphore barbers=0;        #barbieri che aspettano i clienti
semaphore mutex=1;          per la mutua esclusione
int waiting=0;              #clienti che stanno aspettando

void barber(void){
while(TRUE){
        down(&costumers);   barbiere va a dormire se #clienti=0
        down(&mutex);       ottiene accesso esclusivo a waiting
        waiting=waiting-1;  decremento il #clienti in attesa
        up(&barbers);       il barbiere è pronto a tagliare i capelli
        up(&mutex);         rilascio accesso esclusivo a waiting
        cut_hair();         taglia i capelli ed esco dalla regione critica
}       }

void customer(void){
down(&mutex);               entro nella regione critica
if(waiting<CHAIRS){         se non ci sono sedie libere il cliente esce
        waiting=waiting+1;  incremento #clienti in attesa
        up(&customers);     sveglio il barbiere se necessario
        up(&mutex);         rilascio accesso esclusivo a waiting
        down(&barbers);     il barbiere va a dormire se #barbieri liberi=0
        get_haircut();      taglia i capelli al cliente
} else up(&mutex);      }   Se il negozio è pieno, non si aspetta
```

Condizioni necessarie e sufficenti affinchè si verifichi stallo:

-Accesso esclusivo ad una 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 nella catena.

Strategie per affrontare stalli: -Prevenzione: impedendo almeno una delle condizioni procedenti attraverso:
-Accesso esclusivo ad una risorsa;
-Accumulo di risorse (però molti problemi richiedono uso simultaneo di + risorse);
-Inibizione del prerilascio (però alcune risorse non consentono di farlo);
-Attesa circolare (difficile da rilevare e complessa da evitare e sciogliere).

Si può fare prevenzione anche sulle richieste d'accesso:
-Durante l'esecuzione: ad ogni richiesta di accesso si verifica se questa può portare a stallo
In caso affermativo è difficile capire cosa conviene fare, inoltre la verifica di ogni richiesta è un lavoro molto pesante.
-Prima dell'esecuzione: all'avvio di ogni processo si verifica quali risorse dovranno utilizzare cosi da ordiarne l'attivita in maniera conveniente.

-Riconoscimento:si deve essere ammettere che lo stallo si possa verificare ed essere in grado di riconoscerlo. Farlo durante l'esecuzione è molto oneroso in quanto si deve bloccare periodicamente l'avanzamento del sistema per analizzare lo stato di tutti i processi.
-Recupero: si deve possedere una procedura di recupero detta sblocco che però porta alla terminazione forzata di uno dei processi in attesa.
-Indifferenza: non si prende nessuna precauzione e si considera trascurabile la probabilità di stallo.

POLITICHE DI ORDINAMENTO PROCESSI

Ad ogni processo è associato a un descrittore chiamato **Process Control Block** che relaziona il processo alla sua macchina virtuale, ne specifica le caratteristiche attraverso:
-Identificatore del processo;
-Contesto di esecuzione del processo, cioè le info necessarie a ripristinarne lo stato di esecuzione dopo sospensione o prerilascio;
-Stato di avanzamento del processo, cioè un puntatore alla lista del processo in quello stato;
-Priorità;
-Diritti di accesso alle risorse ed eventuali privilegi;
-Discendenza Familiare, con un puntatore al PCB del padre e dei processi figli;
-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 (importante per evitare inversion priority se un processo importante attende che uno meno importante che rilasci la sezione critica);
-All'occorrenza di un interrupt.

Classificazione Processi, possono essere:
-**CPU-bound**: dove l'attività sulla CPU è di durata molto lunga;
-**I/O-bound**: dove le varie attivita dei processi sono di breve durata sulla CPU in quanto sono intervallate da attività di I/O molto lunghe (penalizzata dalla FCFS).

Metodi Alternamento Processi:
-**Scambio Cooperativo**, un processo decide da solo quando cedere il controllo della CPU al processo successivo;
-**Scambio con Prerilascio**, il processo in esecuz viene rimpiazzato o da un processo appena arrivato con >priorità(priority-based pre-emptive, si usa nei sistemi a tempo reale) oppure all'esaurimento del quanto di tempo(time-sharing pre-emptive, si usa nei sistemi interattivi). Il prerilascio si realizza tramite un meccanismo esterno all' esecuzione dei processi che solleva un interruzione, un gestore software che la identifica e che se necessario la comunica allo Scheduler.

Scheduler: componente che usa meccanismi risiedenti nel nucleo che decide che ordinamento e gestione dei processi attuare. E' progettato prima dei processi che è chiamato a governare. Bisogna rendere il suo operato parallelo rispetto agli specifici attributi assegnati ai processi, in modo da non doverlo cambiare al variare delle diverse applicazioni.

Dispatcher: componente del nucleo che realizza l'ordinamento dei processi e che opera su mandato dello scheduler. Deve essere molto efficiente perché opera ad ogni scambio di contesto, esso infatti salva il contesto del processo in uscita, installa quello del processo in entrata e gli affida il controllo della CPU (context switch).

Le **politiche di ordinamento** sono determinate fuori dal nucleo, vengono decise dalle applicazioni, fissando quali valori assegnare ai parametri di configurazione dei processi considerati dai meccanismi di gestione del nucleo.

L'**efficienza** delle politiche scelte si misura in termini di:
-% impiego utile della CPU (tempo di esecuz di scheduler e dispatcher viene sottratto ai processi);
-#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).

L'esecuzione comprende sequenze di azioni eseguibili dalla CPU,intervallate da sequenze di azioni di I/O. I processi si possono classificare in:

-**CPU-bound**: se comprendono attività di lunga durata sulla CPU;

-**I/O-bound**: se comprendono attività di breve durata sulla CPU intervallate da attività I/O molto lunghe. (penalizzati dalla politica FCFS dato che quando si cede la CPU durante le attività di I/O sono ritardati al ritorno dai processi che li hanno restituiti).

Si parla di processo quando si opera con prerilascio, di lavoro se si opera senza prerilascio.

Politiche di Ordinamento:

-**Politica a rotazione con priorità**: ci sono code per ciascuna categoria di processo(CPU-bound,I/O-bound) che vengono ordinate per priorità, si stabilisce poi una politica di ordinamento tra code(Round Robin) ottenendo cosi una politica di ordinamento a livelli;

-**Politica a priorità con rotazione**: ci sono code per ogni livello di priorità. Si seleziona tra le code quella con prioritá più elevata e si applica la politica a rotazione(Round Robin) sul processo selezionato., ottenendo cosi una rotazione equa entro ciascuna coda.

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

-**Sistemi a Lotti (batch)**: hanno ordinamento predeterminato, lavori di lunga durata e di limitata urgenza. Qui il prerilascio non è necessario. Le politiche per tali sistemi devono garantire massimo prodotto per unità di tempo(throughput),massima rapidità di servizio per singolo lavoro(turn-around) e massimo utilizzo delle risorse di elaborazione.

Politiche di ordinamento:

-FCFS: tecnica di gestione più semplice, il primo accesso ad entrare in coda sarà anche il primo avviato all' esecuzione che però non è garantita dato che dipende però dall'esecuzione degli altri processi (se senza priorità, Tempo Attesa=Tempo Risposta);

-SJF(Short Job First): senza prerilascio, esegue prima il lavoro più breve. Non equo con i lavori non presenti inizialmente;

-SRTN(Shortest Remaining Time Next): variante di SJF con prerilascio, esegue il processo più veloce da completare e tiene conto dei nuovi processi quando arrivano (minimizza Turn-Around) .

-**Sistemi Interattivi**: hanno grande varietà di attività, qui il prerilascio essenziale. Le politiche per tali sistemi devono garantire rapidità di risposta per singolo lavoro e soddisfare le aspettative generali dell'utente.

Politiche di ordinamento:

-Ordinamento a quanti(Round-Robin): tecnica a rotazione che impone time sharing, con prerilascio o senza, come FCFS ma ogni processo esegue al massimo un tot di quanti per volta per poi prerilasciare il processo per farne spazio ad un altro.(Tempo Attesa≥Tempo Risposta);

-Ordinamento a quanti con priorità: ci sono quanti diversi per ogni livello di priorità;

-Con garanzia per processo: con prerilascio e con promessa di una data quantità di tempo di esecuzione(per es 1/n per n processi concorrenti). Esegueper primo il lavoro maggiormente penalizzato rispetto alla promessa.

-Senza garanzia: con prerilascio e priorità, opera sul principio della lotteria, ogni processo riceve dei numeri da giocare, più la priorità del processo è alta,più numeri ha da giocare.Ad ogni scelta di assegnazione della risorsa,essa va al processo possessore del numero estratto. Ogni estrazione avviene ogni tot di quanti.

-Con garanzia per utente: come garanzia per processo ma con garanzia riferita a ciascun utente.

-**Sistemi in Tempo Reale**: sistemi concorrenti nei quali i lavori hanno 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 ed oltre tale limite il valore ha utilità decrescente,nulla o negativa.

Politiche di ordinamento:

-Modello semplice(cycle executive): opera su un insieme fissato di processi periodici ed indipendenti. Ogni processo è suddiviso in una sequenza ordinata di procedure di durata massima nota. L'ordinamento viene costruito a tavolino come una sequenza di chiamate a procedure di processi fino al loro completamento. Il major cycle racchiude l'invocazione di tutte le sequenze di tutti i proessi, esso è suddiviso in N cicli minori di durata fissa che racchiudono l'invocazione di specifiche sottosequenze.

-Ordinamento a priorità fissa: si assegnano le priorità in base al periodo. Per scadenza uguale al periodo, si da priorità maggiore al periodo più breve.

-Ordinamento a priorità fissa con prerilascio: per scadenza inferiore al periodo, si da priorità in base alla scadenza. Qui c'è il rischio di Inversion Priority, in quanto processi a priorità maggiore possono essere bloccati dall'esecuzione di processi a priorità minore. Questo è causato dall'accesso esclusivo a risorse condivise e può causare deadlock.

Es: Data la politica RR con quanto q e N processi interattivi tutti con lo stesso comportamento. Ogni interazione da luogo ad un CPU burst che richiede alla CPU per un tempo c. Se:

- c<q: -un processo aspetta al max (N-1) · c nella coda dei pronti per ottenere la CPU; -un utente aspetta al max (N-1) · c + c = Nc per aspettare che la CPU finisca di elaborare l'interazione;

- c>q: -considerando che c può essere espresso come c= a·q+b (dove a=quanti di tempo; b=tempo nell'ultimo quanto(<c)), un utente aspetta N·q·a+(N-1)·b (oppure (N+1)·c+a·q) per iniziare l'ultimo quanto di interazione per l' ultimo processo rimasto ancora attivo.

RICAPITOLAZIONE CONCETTI BASE:

Gerarchia fisica di memoria:

-**Registri** interni alla CPU sono ampi 32 o 64 bit;
-**Cache:** è controllata dall'hardware e suddivisa in blocchi detti line con ampiezza tipica di 64 bit;
-**Dischi Magnetici:** hanno capienza 100 volte superiore e costo/bit 100 volte inferiore rispetto alla RAM ma con tempo di accesso 1000 inferiore;

-**CMOS**: Memoria volatile alimentata da una piccola batteria, memorizza l'ora e da quale disco fare il boot, le impostazioni BIOS diverse da quelle di default.
-**MMU**: Gestisce lo spazio di memoria virtuale dei processi, logicamente è posta tra CPU e RAM ed è sotto la responsabilità del SO.

Trattamenti degli Interrupt: segnale asincrono(cioè che non può avvenire contemporaneamente ad un altro interrupt) che indica il bisogno di effettuare un I/O oppure la richiesta da parte di processo o del SO di effettuare una particolare operazione. L'uso delle interruzioni evita il ricorso al polling (verifica ciclica del SO a tutte le unità/periferiche).
Viene gestito in modalità kernel ed 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(pin) di notifica verso la CPU;
4. Quando la CPU si dispone a ricevere la notifica, il controllore delle interruzioni comunica l'identità del dispositivo;
5. All'arrivo dell'interruzione i registri PC(program counter) e PSW(program status word) 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 è possibile individuare il gestore designato a servire l'interruzione;
8. La parte immediata del gestore esegue nel contesto del processo interrotto.

Plug&Play: utilizzata da Intel e Microsoft, agli inizi ogni scheda I/O aveva un livello di interrupt fisso e un indirizzo fisso per i registri, 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): salvato in un chip ROM, contiene un software a basso livello per la gestione di I/O. Viene caricato all'avvio del computer e 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 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: esso contiene un programma che esamina la tabella di partizione e legge il SO dalla partizione attiva e lo esegue. Il SO interroga il BIOS per ottenere informazioni sulla configurazione del sistema e successivamente attua varie inizializzazioni per poi eseguire il programma iniziale (login,GUI).

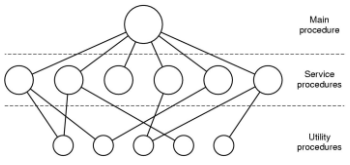
System Calls: la maggior parte dei servizi del SO sono eseguiti in risposta a invocazioni esplicite di processi dette chiamate di sistema, cioè un segnale sincrono(ovvero più chiamate a sistema possono essere fatte nello stesso istante).
Esse sono nascoste in procedure di libreria predefinite: l'applicazione non effettua direttamente chiamate di sistema, la procedura di libreria svolge il lavoro di preparazione necessario ad assicurare la corretta invocazione della system call. La prima istruzione di una system call (**trap**) 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 e invoca la procedura di libreria corrispondente alla chiamata;
2. Questa pone l'ID della chiamata in un luogo noto al SO;
3. Poi esegue l'istruzione trap per passare all'esecuzione in modo operativo privilegiato;
4. Il SO individua la chiamata da eseguire e la esegue;
5. Poi ritorna al chiamante oppure ad un nuovo processo;
6. Cancella dati nello stack facendo avanzare il puntatore.

Es: -Se un sistema è in blocco da 10ns ⇒ 10ns fa ha eseguito una system call.
-Un processo per creare/lanciare un nuovo processo deve effettuare una system call.
-Una system call bloccante causa sempre un context switch. ma solo se c'è qualche altro processo attivo.

STRUTTURE DEL KERNEL, cioè le architetture logiche di S/O**:**

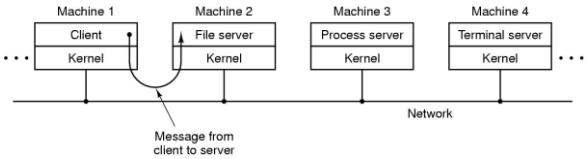
Struttura Monolitica: l'approccio monolitico non ha struttura ma definisce un'interfaccia virtuale di alto livello sull'hardware, con un set di primitive o chiamate di sistema per implementare servizi di sistema operativo come gestione dei processi, multitasking e gestione della memoria, in diversi moduli 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 stessso 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. C'è un programma principale che invoca le procedure di servizio richiesta;
2. Un insieme di procedure che eseguono le chiamate a sistema;
3. Un insieme di procedure di utilità che sono di ausilio per 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-K consentono gestione più flessibile;
-I Monolitici sono + semplici da realizzare e da mantenere;
-I Micro-K hanno problemi di sincronizzazione tra le varie componenti che ne rallentano sviluppo e mantenimento.

Tabella Unità Metriche:

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

GESTIONE DELLA MEMORIA

La memoria deve essere: Capiente, Veloce, Permanente. Solo l'**intera gerarchia** di memoria al completo possiede tutte queste caratteristiche.

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: eseguono un solo processo per volta. La memoria disponibile è ripartita tra solo quel processo e il SO. L'unica scelta progettuale rilevante è decidere dove allocare la memoria del SO, la parte di SO ospitata in RAM è solo quella che contiene l'ultimo comando invocato dall'utente.

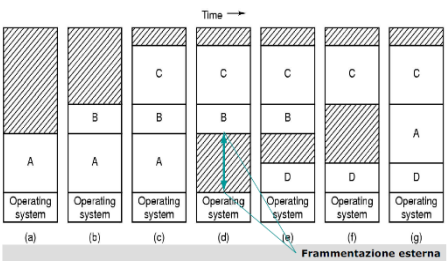
Sistemi multiprogrammati: la forma più rudimentale crea all'avvio del sistema una partizione di dimensione diversa per ogni processo. Il problema è assegnare dinamicamente processi a partizioni minimizzando la frammentazione interna, due modi:

- Ogni partizione contiene una coda di processi e ad ogni nuovo processo(o lavoro) viene assegnata la partizione di dimensione più appropriata → 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 viene assegnata al miglior processo scandendo l'intera coda → Vengono discriminati i processi più piccoli.

Utilzzo stimato CPU nei sistemi multiprogrammati: si fa la **valutazione probabilistica** di quanti processi devono eseguire in parallelo per massimizzare l'utilizzazione della CPU. Dati:-P= percentuale di tempo dedicato all' I/O di un processo;
-N= numero di processi simultaneamente in memoria
⇒**Utilizzo CPU= $1-P^N$**

Esempi Progettazione Memoria: si consideri un computer con 32 MB di memoria e 80%(P=0,8) di attesa I/O media per ogni processo.
-Se ci sono: -16 MB riservati per il sistema operativo
-4 MB riservati per ciascun processo
In totale si hanno 4 processi simultaneamente in memoria ⇒ Utilizzo CPU= $1-0{,}8^4=60\%$
-Se ci sono: -32 MB riservati per il sistema operativo
-4 MB riservati per ciascun processo
In totale si hanno 8 processi simultaneamente in memoria ⇒ Utilizzo CPU= $1-0{,}8^8=83\%$
-Se ci sono: -48 MB riservati per il sistema operativo
-4 MB riservati per ciascun processo
In totale si hanno 12 processi simultaneamente in memoria ⇒ Utilizzo CPU= $1-0{,}8^{12}=93\%$

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 ed è difficile aumentare dinamicamente l'ampiezza della partizione assegnate, perciò vengono assegnate con margine.

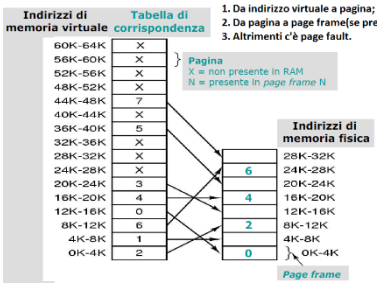
Con la memoria allocata dinamicamente è essenziale tenere traccia del suo stato d'uso, due strategie:
-**Mappe di bit**: la memoria è vista come insieme di unità di allocazione (1 bit per unità) che fanno parte di una grande struttura di gestione.
-**Liste collegate**: la memoria è vista a segmenti in cui ognuno di essi è visto come un processo o uno spazio libero tra un processo e l'altro.
Ogni elemento di lista rappr un segmento e ne specifica punto di inizio,ampiezza e successore.Le liste sono ordinate per indirizzo base.
Politiche allocazione per liste collegate: -**First Fit**: verrà usato il primo segmento libero ambio abbastanza per contenere il processo;
-**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;
-Quick Fit: liste diverse di ricerca per ampiezze "tipiche".

MEMORIA VIRTUALE:
L'intera RAM è presto divenuta insufficente per ospitare un intero processo. Nasce cosi 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 della 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 alternative **tecniche di gestione** sono:

PAGINAZIONE: l'obbiettivo è quello di avere a disposizione spazi di indirizzamento più grandi della RAM.

Premesse: -**Pagine**: unità a dimensione fissa in cui è divisa la memoria virtuale, ogni pagina può contenere al massimo un processo. (pagine virtuali);
-**Page Frame**: unità "cornici" ampie come pagine in cui è suddivisa la RAM. (pagine fisiche);
-**Page Fault**: avviene quando una pagina è assente in RAM. I trasferimenti da e verso disco avvengono sempre per pagine, e per ogni pagina referenziata si controlla se è già presente in RAM o meno attraverso un bit di presenza. Se una pagina non è presente, si genera l'evento di page fault che chiama il SO per gestire il rimpiazzo tramite trap. (molti page fault su un processo causano un diminuimento delle prestazioni degli altri processi)



→La **traduzione da indirizzo virtuale a fisico** avviene tramite una **Page Table**(tabella delle pagine) indicizzata per numero di pagina in cui → Indirizzo fisico=f(indirizzo virtuale). Ogni traduzione deve essere molto veloce, dato che ogni istruzione potrebbe far riferimento più volte alla tabella.
-Ogni indirizzo può essere tradotto tramite:
-Vettore di registri (uno per ogni pagina virtuale) caricato ad ogni cambio di contesto;
-Struttura sempre residente in RAM, con un singolo registro che punta all'inizio della page table.

-Ciascun processo ha la sua tabella delle pagine che può essere molto grande. Ciò è necessario poiché ogni processo ha il suo spazio di indirizzamento virtuale.
(se ho indirizzi virtuali da 32 bit e pagine da 4Kb⇒ ho una memoria virtuale da 4Gb con 1M di pagine)

L'indirizzo di disco dove la pagina si trova quando non è in RAM non si trova nella tabella. La page table serve all'MMU, e nel caso di page fault sta al SO caricarla sulla RAM.

→Se la page table è molto grande, non può risiedere su registri (avrebbe impatto devastante sulle prestazioni) ⇒ quindi deve risiedere in RAM.
Per implementarla serve una struttura hardware supplementare, più agile e che velocizzi la traduzione da indirizzi virtuali in indirizzi fisici, detta **TLB**(translation lookaside buffer) che è una piccola memoria associativa (tipo la cache) interna alla MMU e che consente scansione parallela su tutte le righe simultaneamente. (pensata sul fatto che un processo usa più frequentemente poche pagine)

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 l'equivalente di un cache miss, quindi le info richieste vengono prelevate dalla page table e caricate in TLB, rimpiazzandone una cella e riflettendo il valore nella page table.

→Oggi le TLB sono di solito realizzate attraverso software, in modo da lasciare + spazio alla MMU per usi ritenuti più vantaggiosi(tipo per la cache).

→PROBLEMA: Con le architetture a 64 bit le page table hanno dimensioni proibitive dato che ho:
indirizzi da 64 bit e pagine da 4kb ⇒ allora ho memoria virtuale da 16Eb(1E=1Gb*1Gb) con 4P pagine (1P=1Mb*1Gb)
SOLUZIONE: si impiega una **TABELLA INVERTITA** in cui non si ha più una riga per pagina ma si ha un page frame in RAM(perciò la sua dimensione dipende dall'ampiezza delle pagine in RAM).
La traduzione però diventa ancora più onerosa, dato che la pagina potrebbe risiedere in qualunque page frame e bisognerebbe scandire l'intera tabella per trovarla, anche se la ricerca è velocizzata dall'uso della TLB.
-Si potrebbe realizzare la tabella come una tabella hash in cui i dati relativi alle pagine i cui indirizzi virtuali indicizzano una stessa riga di tabella vengono collegati in lista.

POLITICHE RIMPIAZZIO PAGINE: Quando si produce un page fault il SO deve rimpiazzare una pagina, salvando su disco la pagina rimossa(solo se modificata nell'uso). Vi sono molteplici politiche di rimpiazzo:

-**Optimal Replacement**: rimpiazza la pagina in memoria che non sarà utilizzata dal processo per maggior tempo. La perfetta scelta della pagina è irrealizzabile, in quanto il SO non sa a quali pagine il processo vorrà accedere in futuro;

-**NRU**(Not Recently Used): per ogni page frame vengono aggiornati: bit M(modified, inizializzato a 0 dal SO), bit R(referenced, posto a 0 periodicamente dal SO per stimare la frequenza d'uso).

I page frame vengono classificati in:
1. Classe 0: non riferita, non modificata;
2. Classe 1: non riferica,modificata;
3. Classe 2: riferita, non modificata;
4. Classe 3: riferica, modificata.

→NRU sceglie una pagina a caso nella classe non vuota a indice più basso.

-**FIFO**: si ha una lista ordinata di page frame, ogni riferimento è messo in coda e rimuove la pagina a ingresso più antico in RAM (quella in testa);

-**Second Chance**: corregge FIFO rimpiazzando solo pagine con bit R=0, altrimenti il page frame viene considerato come appena caricato e posto in fondo alla coda con R=0;

-**Orologio**: come Second Chance 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 e necessita un'hardware dedicato;

-**NFU**(Not Frequentily 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 viene sostituita;

-**Aging**: NRU modificato, pero ogni page frame aggiorna un contatore C che cresce di più se R=1 ma non incremente C con R ma gli inserisce R a sinistra, ma 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** (evita page fault). w(k,t) è l'inzieme di pagine che soddisfano i k riferimenti emessi al tempo t.
Se si conoscesse il WS dei processi, le pagine da rimpiazzare sarebbero quelle che non vi sono comprese. Conoscerlo è però troppo costoso.

Anomalia di Belady: la frequenza di un page fault non sempre decresce al crescere dell'ampiezza della RAM(LRU sarebbe immune da questo, ma la suo forma ottimale è irrealizzabile).
Gli **stack algorithms**(ad es LRU e Optimal Replacement) sono immuni all'anomalia perché soddisfano tale proprietà:
M(m,r)C M(m+1,r) dove m rappresenta il numero di page frame, mentre r sono i riferimenti.
La proprietà dice che assumendo gli stessi riferimenti, le pagine caricate con m page frame sono un sottoinsieme di quelle caricate con m+1 page frame.

Per rimpiazzare una pagina occorre scegliere tra:

-**Politiche locali**: si rimpiazza il WS del processo che ha causato page fault, in tal caso ogni processo conserva una quota fissa in RAM;

-**Politiche globali**: la scelta avviene tra page frame senza distinzione di processi e la RAM a disposizione di ogni processo varia diamicamente nel tempo. Le politiche globali sono più efficenti con prestazioni superiori.

Grandezza Pagine: con **Pagine ampie**, si rischia di mantenere in memoria una parte di processo che non verrà mai usato, inoltre c'è maggiore rischio di frammentazione interna, dato che in media ogni processo lascia inutilizzata metà del suo ultimo page frame. D'altra parte i trasferimenti da e verso disco avvengono una pagina per volta, perciò **Pagine piccole** richiederebbero più tempo di trasferimento che con una pagina più ampia dato che la page table deve essere caricata nei registri ogni volta cha la CPU cambia processo., inoltre si ha maggiore ampiezza della tabella delle pagine;

Siano
-s = dimensione media di un processo in byte;
-p = dimensione media di una pagina in byte;
-ε = byte per ciascuna riga in tabella delle pagine;

L'overhead totale (spazio non necessario) può essere definito tramite la **funzione f(p)= (s/p) ·ε+p/2** dove:

-#pagine richieste per ognuno dei processi= s/p ⇒ quindi ogni pagina occupa se/p byte nella tabella delle pagine, questo valore è grande tanto quanto la dimensione delle pagine è piccola;

-La memoria sprecata nell'ultima pagina per via della framm interna=p/2 byte, questo valore è grande tanto quanto la dimensione delle pagine è grande;

Quindi si calcola il valore ottimo ⇒ si fa la derivata prima rispetto a p e la poniamo a 0 ⇒ otteniamo $-se/p^2+1/2=0$ ⇒
⇒da qui deriviamo la dim ottima della pagina ⇒ $p=\sqrt{2 \cdot s \cdot \varepsilon}$.
Es: Se s=1MB ed ε=8B ⇒ dim ottima=4KB. Ad oggi le pagine più comuni sono da 4 o 8 KB.

Trattamento di page fault: si deve capire quale riferimento è fallito in modo da poter completare l'istruzione interrotta. Il program counter dice a quale indirizzo si è verificato il problema, ma non sa distinguere tra istruzioni ed operando perciò tocca al SO capirlo. Si realizza:

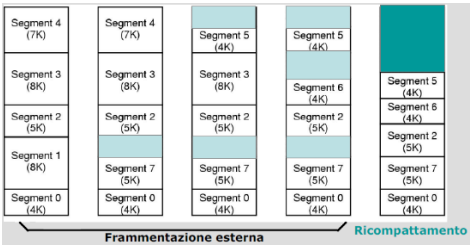
1.L'hadware fa un trap al kernel e salva il PC sullo stack;
2. Un programma assembler salva i dati nei registri e poi chiama il sitema operativo;
3. Il SO scopre il page fault e cerca di capire quale pagine lo abbia causato;
4. Ottenuto l'indirizzo virtuale causa del page fault, il SO verifica che si tratti di indirizzo valido e cerca page frame rimpiazzabile;
5. Se il page frame è sporco, 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 continauare 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, anche se tanto è più grande la lunghezza media dei programmi, tanto meno sarà rilevante il fenomeno della frammentazione interna, in quanto le pagine saranno più frequentemente rimpiazzate.

Es: dato un sistema di memoria con indirizzi virtuali suddivisi nei 4 campi a,b,c,d. I primi 3 dei quali utilizzati per indirizzare tre livelli gerarchici di tabelle delle pagine e il quarto campo rappresenti l'offset entro la pagina selezionata. ⇒ Allora l'ampiezza di a,b,c dipende dal numero di pagine indirizzate nel sistema.

SEGMENTAZIONE:
I segmenti hanno dimensione variabile (al contrario delle pagine che hanno dimensione fissa) ed inoltre questa tecnica può ridurre il consumo di memoria, in quanto consente a più processi di condividere blocchi di codice e dati, dato che ogni segmento può contenere più processi.
Si hanno spazi di indirizzamento completamente indipendenti gli uni dagli altri per dimensione e posizione in RAM e che possono variare dinamicamente. Essi rappresentano **entità logiche** note al programmatore e destinate a contenere informazioni condivise (codice di procedure, dati di inizializzazione di un processo, stack di processo…).

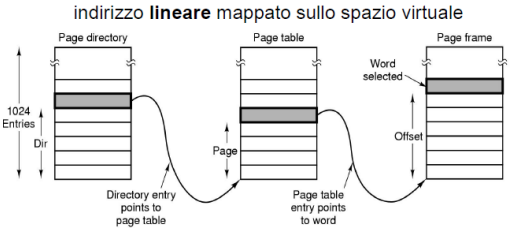


⇒Inizialmente si ha una memoria fisica che contiene 5 segmenti. Se si esegue la sequenza di operaz:
1. Il segmento 1 viene scaricato e al suo posto viene messo il segmento 7, che è più piccolo. Si crea così un buco, cioè un'area non usata;
2. Il segmento 4 è rimpiazzato dal segmento 5;
3. Il segmento 3 è rimpiazzato dal segmento 6;

⇒Alla fine si avrà un certo numero di porzioni contenenti segmenti e buchi. Perciò il SO vi pone rimedio ricompattando la memoria ed evitando così la frammentazione esterna.

La tecnica si presta a schemi di protezione specifica dato che il tipo del suo contenuto può essere stabilito in precedenza, ma ciò 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.

-Data la grande ampiezza potenziale dei segmenti, possono essere **paginati**, come nel Pentium di Intel:
Si hanno: -Fino a 16k segmenti indipendenti di ampiezza massima 4Gb (la base del segmento in RAM è da 32bit) ;
-Una **LTD(Local descriptor Table)**: descrive i segmenti del processo;
-Una **GDT(Global Descriptor Table)**: descrive i segmenti del SO.
Per accedere ad un segmento, un programma Pentium prima carica il selettore di quel segmento in uno dei 6 registri di segmento, di cui uno denota il segmento corrente.
L'**indirizzo lineare** è ottenuto da base di segmento+offset e può essere interpretato come:
-Indirizzo Fisico se il segmento considerato non è paginato;
-Indirizzo Logico se il segmento è paginato. In questo caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa.



FILE SYSTEM:

E' il servizio del 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**(residenti in memoria secondaria).

All'utente deve offrire:
-Modalità di accesso ai file;
-Struttura logica e fisica dei file;
-Operazione ammissibili su file.

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

FILE: sono un insieme di dati correlati, residenti in memoria secondaria e trattati unitariamente. Sono un concetto logico realizzato tramite meccanismi di astrazione che permettono di salvare informazioni sulla memoria secondaria per poi potendole trovare in seguito senza conoscerne ne la struttura logica e fisica ne il funzionamento.

All'utente interessa poter designare le proprie informazioni mediante nomi logici unici e distinti, le caratteristiche distintive di un file sono:

-**Attributi**: nome, dimensione corrente, data creazione, data ultima modifica, creatore e possessore, permessi d'accesso;

-**Struttura dei dati**: la struttura dei dati all'interno di un file può essere considerata da 3 punti di vista:

1. **Livello Utente**: il programma applicativo associa autonomamente significato al contenuto grezzo del file;
2. **Livello Struttura Logica**: il SO organizza dati grezzi in strutture logiche per facilitarne il trattamento. Le possibili strutture logiche sono:
   - -Sequenza di byte: l'accesso ai dati utilizza un puntatore relativo all'inizio del file. Lettura e Scrittura operano a blocchi di byte.
   - -Record con 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 (scelta obsoleta);
   - -Record con 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 separata, ordinata per chiave, contenente 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. 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 va chiuso.

Allocazione FS: essi sono memorizzati su disco. Ogni disco può essere partizionato 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. Il settore 0 del disco contiene info di inizializzazione del sistema(detta **Master Boot Record**) che viene eseguita dal BIOS, mentre il primo blocco di ogni partizione contiene le sue specifiche info di inizializzazione(**boot block**).

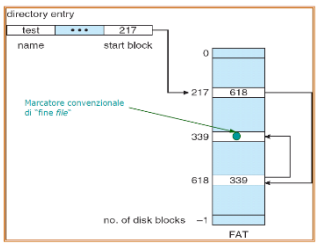
A livello fisico un file è un insieme di blocchi su disco, serve quindi decidere quali blocchi assegnare a quali file e come tenerne traccia.

Strategie di allocazioni di blocchi a file:

-**Allocazione Contigua**: memorizza i file su blocchi consecutivi. Ogni file è descritto dall'inizio del suo primo blocco e dal #blocchi utilizzati. Consente sia accesso sequenziale che diretto. Ogni file può essere letto/scritto con un solo accesso(CD/DVD), però ogni modifica comporta il rischio di frammentazione esterna.

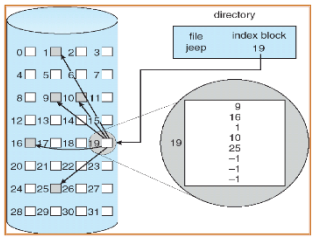
-**Allocazione a lista concatenata**: il file è visto come lista concatenata di blocchi. Si ha un puntatore al suo primo blocco(su alcuni file anche dell'ultimo) e ciascun blocco deve contenere il puntatore al blocco successivo(o punto di fine lista), perciò un solo guasto corrompere l'intero file. Consente sia accesso sequenziale che diretto(lento).

-**Allocazione a lista indicizzata**: i puntatori ai blocchi risiedono in strutture apposite(qui ciascun blocco contiene solo dati). Il file è quindi descritto dall'insieme dei suoi puntatori. Consente accesso sequenziale e diretto e non causa frammentazione esterna.
Non richiede di conoscere preventivamente la dimensione massima di ogni nuovo file.
Due strategie di organizzazione:

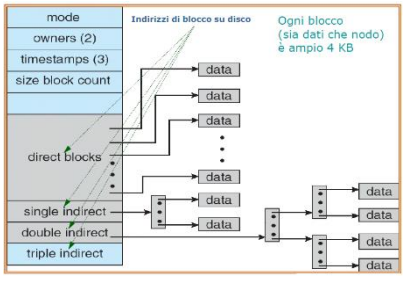


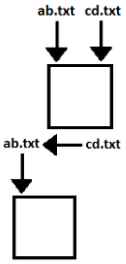
⇒**FAT**(file allocation table): un file è visto come una catena di indici).
Si ha una tabella ordinata di puntatori, con un puntatore ∀blocco del disco. La porzione di FAT relativa ai file in uso deve sempre risiedere interamente in RAM: ciò consente accesso diretto ai dati seguendo sequenzialmente i collegamenti.

Dato un disco ampio 4Gb con blocchi ampi 4Kb e contenente 128K file
⇒Allora l'ampiezza in blocchi della FAT dipende dall'ampiezza del disco in blocchi e dell'ampiezza degli indirizzi di blocco.



⇒**Nodi Indice**(forma indicizata): ∀ file si ha una struttura indice detta **i-node** con gli attributi del file e i puntatori ai suoi blocchi. L'i-node è contenuto in un blocco dedicato, mentre una tabella di i-node per i soli file in uso è allocata in RAM.
→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, invece 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 cosi via.





⇐ **Hard Link**: è un puntatore diretto al descrittore(i-node) di un file regolare che 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 che n non può essere distrutto fin quando i suoi descrittori remoti non vengono eliminati. Più efficace a livello prestazionale.

⇐**Symbolic Link**: viene creato un file speciale il cui contenuto è il cammino del file originario (si ha un puntatore al puntatore all'i-node del file originale). Il file originario può avere qualunque tipo e risiedere anche in un FS remoto. Mantiene una sola via d'accesso al file originario e l'accesso condiviso avviene tramite cammino sul FS.

Gestione Blocchi liberi: -Con Bitmap dove ogni bit indica lo stato del blocco corrispondente(0=libero, 1=occupato);
-Con una Lista Concatenata di blocchi che sfrutta i campi al puntatore successivo(come nella FAT). E' l'equivalente di un file composto da soli blocchi liberi.

Gestione blocchi danneggiati: -Via hardware: si crea e mantiene 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.

Directory: ogni FS usa directory o cartelle 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 e ciascuno con un proprio unico nome(semplice ma gestione onerosa all'aumentare del file);
- -A due livelli: una root directory contiene una UFD(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: c'è un numero arbitrario di livelli. Il livello superiore è detto root. 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 e 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").

La directory fornisce info su: nome,collocazione,attributi di file appartenenti a quel particolare catalogo.

File e Directory risiedono in aree logiche distinte, perciò conviene minimizzare la complessità gestionale della struttura interna alla directory assegnando una lunghezza fissa anche se il suo contenuto è di ampiezza variabile.

Ricerca di un File: correla il suo nome alle informazioni necessarie all'accesso: nome e directory di appartenenza del file sono determinati dal percorso indicato dalla richiesta. -Ricerca lineare in directory è facile realizzazione ma di esecuzione onerosa.
-Ricerca mediante tabelle hash è + complessa ma + veloce: F(nome)=posizione tabella->puntatore a file
Si può anche creare in RAM una cache di supporto alla ricerca.

Modalità Accesso file: -Accesso Sequenziale: viene trattato un gruppo di byte(o un record) per volta. Un puntatore indirizza il gruppo di byte(o record) corrente e avanza a ogni lettura o scrittura(la lettura può avvenire in qualunque posizione del file che 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, la posizione è determinata rispetto alla base(che ha offset=0)
- -Accesso Indicizzato: per ogni file si ha una tabella di chiavi ordinate contenenti gli offset dei rispettivi record nel file (perciò l'info di navigazione non è più nei record ma in una struttura a parte ad accesso veloce). Si fa una ricerca binaria della chiave e poi si utilizza l'accesso diretto per accedere al file.

Il File System tratta **diversi tipi di file**: -File regolari: sui quali l'utente può operare normalmente(su windows e unix);
-File catalogo: tramite i quali il FS permette di descrivere l'organizzazione di gruppi di file(su windows e unix);
-File speciali: con i quali il FS rappresenta logicamente dispositivi orientati a carattere o a blocco(solo unix).

Offset: numero intero che indica la distanza tra due elementi all'interno di un gruppo di elementi dello stesso tipo.

File Mappati: Il SO può mappare un file in memoria virtuale, all'indirizzo di ogni suo dato(risiedente in memoria secondaria) corrisponde un indirizzo in memoria virtuale(base+offset). Le operazioni su file avvengono in memoria principale:
⇒Chiamata di indirizzo→Page Fault→Caricamento→Operazione→Rimpiazzo di pagina→Salvataggio in memoria secondaria.
Ciò riduce gli accessi a disco ma crea problemi nella condivisione e con file di enorme dimensione.

Gestione dei file condivisi:

- -∀ file condiviso si pone nella directory remota un symbolic link verso il file originale. Esiste così un solo descrittore(i-node) del file originale. L'accesso condiviso avviene tramite cammino sul FS.
- -In alternativa si può porre nella directory remota il puntatore diretto(hard link) al descrittore (i-node) del file originale. Si hanno più possessori di descrittori dello stesso file condiviso ma c'è un solo proprietario effettivo del file condiviso. Il file condiviso non può essere distrutto fin quando esistono suoi descrittori remoti anche se il suo proprietario avesse intenzione di cancellarlo.

NTFS: file system usato da Win NT XP e Vista,supporta l'intera gamma di FS Windows e anche ext2fs di GNU/Linux. Utilizza indici espressi su 64bit.

Caratteristiche:
- -Nome di file fino a 255 caratteri in codifica Unicode (2 o 4B per carattere);
- -Un file è visto come aggregato di attributi rappresentati come sequenza di caratteri(byte stream);
- -E' ad architettura gerarchica(come ext2fs), supporta entrambe le forme di link;
- -E' una collezione di volumi logici (un volume logico può mappare su + partizioni e anche su + dischi);

Volume: si intende una sequenza lineare di blocchi (cluster) di ampiezza fissa (tra 512B e 4KB).
- -Blocco piccolo → ridotta frammentazione interna ma piu accessi a disco;
- -Blocco grande → meno accessi a disco ma più frammentazione;

MFT (Master File Table) è la principale struttura dati del NTFS:
- -Una per volume;
- -Fisicamente è realizzata come un file (quindi può essere salvata ovunque);
- -Logicamente è strutturata come una sequenza lineare di $\leq 2^{48}$ record di ampiezza da 1 a 4KB (ciascun record descrive 1 file identificato da un indice ampio 48bit). Gli altri 16 bit servono come numero di sequenza per il contatore di riuso.
- - I primi 16 record sono sempre riservati per i **metadata** (record che descrivono l'organizzazione del volume);

Ciascun Record contiene un numero variabile di coppie del tipo **<descrittore di attributo,valore>** dove:
- -Il primo campo specifica la struttura dell'attributo (ha ampiezza 24B per attributi residenti,piu ampio per quelli non residenti).
- -Il secondo specifica il valore dell'attributo(un attributo è residente se il valore è rappresentato interamente nel record, non residente se rappresentato da un puntatore al suo record.

⇒La strategia NTFS consente di rappresentare **file di ampiezza virtualmente illimitata**:

Il numero di record necessari per i dati di un singolo file dipende dalla contiguità con cui sono scritti i suoi blocchi su disco piuttosto che dalla sua ampiezza.

Es° -file da 20Gb costituito da 20 seq di 1M blocchi da 1Kb ⇒ Ciascun record richiede 20+1 coppie di valori espressi su 64bit ovvero $21 \cdot 2 \cdot 8B = 336B$;

-file da 64Kb costituito da 64 seq di 1 blocco ciascuna ⇒ Ciascun record richiede 64+1 coppie di valori espressi ovvero $65 \cdot 2 \cdot 8B = 1040B$.

→Quindi se rappresentare un file richiede piu di un record, NTFS usa una tecnica a continuazioni analoga a quella usata dagli i-node di UNIX e GNU/Linux. Infatti il record base di MFT contiene un puntatore a N>=1 record secondari in MFT che descrivono la sequenza di blocchi 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 sarebbe posta in un altro file denotato da un record posto in MFT.

Se un file è di piccola dimensione (<1Kb), l'NTFS potrebbe decidere di scriverne il contenuto direttamente nel record dell'MFT.

Es: la contiguità dei blocchi con cui viene scritto un file su disco non influenza l'overhead generato dai FS NTFS e EXT2FS

Es: La dim max del file dipende dalla contiguità con cui sono scritti i blocchi su disco?
- - EXT2: No, perché sono un gruppo concatenato di puntatori quindi non importa nulla della contiguità.
- - FAT: No, perché è una tabella indicizzata ed indicizza ogni blocco, quindi anche se fosse a contiguità nulla l indicizzazione sarebbe comunque ad ogni blocco;
- - NTFS: Si, la contiguità influenza moltissimo sul peso della struttura perche la memorizzazione è indicizzata da un puntatore inizio e un puntatore fine per ogni blocco contiguo, infatti il record base di MFT contiene un puntatore a N>=1 record secondari in MFT che descrivono la sequenza di blocchi del file. Per esempio un file con contiguità massima avrebbe solo una coppia <inizio,fine> invece con contiguità nulla N coppie con N uguale al numero di blocco non contigui. 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 sarebbe posta in un altro file denotato da un record posto in MFT.

NB: Potrebbero mancare argomenti e altri potrebbero essere errati, perciò non prendete questi appunti come oro colato, ci sono buone probabilità che io abbia scritto una gran compilation di minchiate.

JACOPO Z.