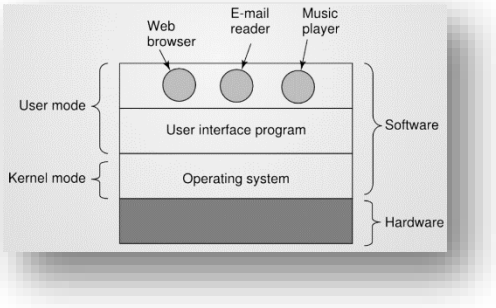
SISTEMI OPERATIVI



Un SO è un astrazione della macchina sottostante, è una parte del software che lavora tra l'hardware e il resto del software che comunica più ad alto livello (interfaccia utente, programmi).
Le moderne CPU hanno 2 modalità:

- Modalità Kernel (supervisor) → pieni poteri.
- Modalità Utente (user mode) → poteri limitati, i processi in questa modalità hanno limitazioni e gli è negato l'uso di operazioni riservate.

Il SO è ovviamente eseguito in modalità Kernel, mentre i vari programmi sono eseguiti in user mode per questioni di sicurezza la UI (User Interface) è eseguita in modalità utente: questo perché è bene creare e programmare un software con il minor numero di permessi possibili.

APPROCCI DESCRITTIVI

1) Il SO come macchina estesa:

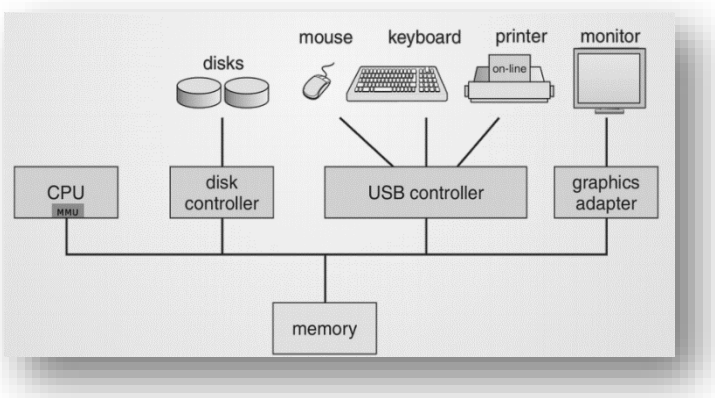
L'hardware (HW) ha una propria interfaccia che è però a basso livello, è quindi di difficile interazione con i livelli soprastanti.
Tramite il SO possiamo interagire più facilmente con HW, infatti questo trasforma "l'ugly interface" dell'HW in un interfaccia più "user friendly" e quindi più fruibile dalle applicazioni attraverso le chiamate di sistema (System Call).
Agevola l'operato dell'utente rendendo le operazioni più facili ed intuitive.

2) Il SO come gestore di risorse:

Il SO mette a disposizione dell'utente una comoda interfaccia, gestisce tutti i pezzi che compongono un sistema più complesso, è colui che ha il pieno controllo. Ha il compito di distribuire in maniera efficace ed efficiente le risorse disponibili ai vari programmi che competono tra loro per usarle.
Deve gestire e proteggere le componenti che sono sfruttate in concorrenza da vari utenti/programmi (memoria, dispositivi I/O, CPU, ecc…). I sistemi odierni sono Multi-core, quindi riescono a gestire più richieste contemporaneamente, e Multi-utente. Il SO deve saper gestire bene la CPU, deve cederne l'uso ciclicamente a processi diversi così da soddisfare le varie richieste.
Il suo compito principale è tenere traccia di chi sta usando quale risorsa e soddisfare le richieste degli utenti/programmi mediandole se risultano conflittuali.

HARDWARE

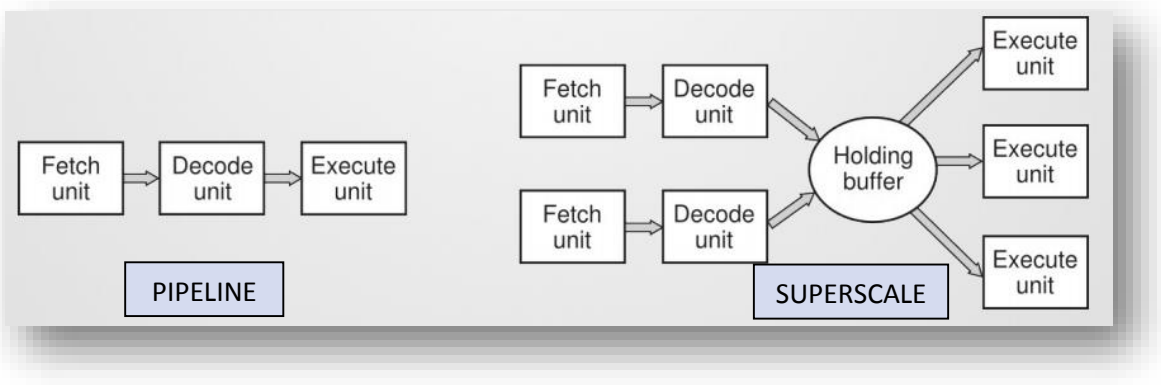
Un SO è profondamente legato alle risorse HW che ha a disposizione dalla macchina.
Concettualmente, un PC è strutturato come nell'immagine: CPU, dischi, memoria, dispositivi di I/O comunicano tra loro attraverso dei BUS.



PROCESSORE (CPU): è il cervello del calcolatore.
Ogni processore può eseguire un set di comandi predefinito (un processore INTEL non può eseguire un programma o un comando creato per processori SPARC e viceversa).
Il ciclo di base di un processore prevede il prelievo delle informazioni (fetch), la loro decodifica (decode) e la loro esecuzione (execution).
Tutte le CPU contengono al loro interno alcuni registri per mantenere temporaneamente risultati temporanei, variabili e chiavi.

Altri particolari registri della CPU sono:

- Program Counter (PC): Contiene l'indirizzo di memoria della prossima istruzione, una volta che l'istruzione viene recuperata il PC viene aggiornato e punterà all'istruzione successiva.
- Program Status Word (Parola di stato del programma, PSW): È un registro che contiene vari flag e valori che rappresentano lo stato del processo in un determinato momento. Inoltre contiene uno o più bit legati allo stato della CPU (modalità kernel/user) e vari bit di controllo. Non è consentito di agire su tutti i campi del PSW, ma solo su determinati bit.
- Stack Pointer (Puntatore a pila, SP): È un puntatore che punta alla cima dello stack corrente in memoria. Ogni stack contiene una struttura (frame) per ogni procedura ancora non terminata, contiene le variabili di ingresso, le variabili locali e temporanee che non vengono contenute in altri registri.



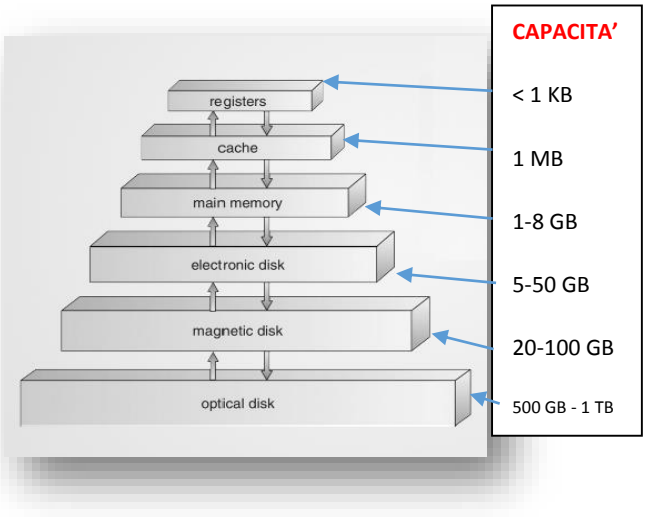
Esistono diversi **progetti per la CPU**:

- Pipeline: È un organizzazione che consente alla CPU di eseguire più azioni diverse in maniera contemporanea: al contrario del ciclo base (dove fetch, decode e execute sono eseguiti uno dietro l'altro un processo alla volta), la CPU potrebbe avere unità separate per il fetch, il decode e l'execute.
  Mentre il processo N è in *execute*, la CPU fa il *decode* del processo N+1 e contemporaneamente la *fetch* del processo N+2.
- Superscale: Consiste in unità multiple di esecuzione, vengono recuperate due o più istruzioni per volta, vengono decodificate e riservate in un buffer in attesa di essere eseguite. Non appena un unità di esecuzione si libera, pesca dal buffer un istruzione che è in grado di gestire e la esegue. In questo modo le istruzioni non saranno eseguite secondo l'ordine pensato dal programmatore.

Modalità di esecuzione:

Abbiamo visto che esistono due modalità di esecuzione: kernel (SO, e altre cose) e user mode (UI, processi in generale). Per ottenere servizi da parte del sistema operativo, i programmi effettuano delle chiamate di sistema (system call), che esegue una TRAP al kernel ed invoca il SO: l'istruzione TRAP cambia la modalità da utente a kernel. Una vota finito il lavoro, il controllo viene restituito. Le TRAP sono usate soprattutto per errori nei programmi (divisione per 0, underflow, ecc…) e vengono gestiti dal SO.

MEMORIA: Idealmente, la memoria dovrebbe essere veloce (più della CPU in modo tale da non rallentare l'elaborazione), abbondante e a bassissimo costo… NON ESISTE UNA MEMORIA DEL GENERE.



Per questo motivo si è adottata una **gerarchia a strati**.

- Registri della CPU: costruiti con la stessa tecnologia della CPU, quindi sono veloci quanto questa e non c'è nessun ritardo di accesso. La capacità di memorizzazione è di 32x32 bit (CPU a 32 bit) o di 64x64 bit (CPU 64 bit).
- Cache: è una memoria controllata dall'HW, è divisa in linee di cache di 64 byte (con indirizzi 0-63, nella prima linea, 64-127 seconda, ecc…). Serve a mantenere indirizzi di dati frequentemente utilizzati, pronti per essere recuperati in fretta. Possono esistere diversi livelli di cache, più piccoli sono, più veloce è l'accesso, ma tutti sono molto costosi. Se l'esito della ricerca sulla cache è nullo, si passa il dato alla memoria centrale.
- Memoria Centrale (Random Access Memory, RAM): memoria rappresentativa delle capacità di lettura-scrittura della macchina.
- Altri dischi (Hard disk, SSD): memorie di dimensione massiccia per il salvataggio dei dati.

MULTITHREADING (O HYPERTHREADING)

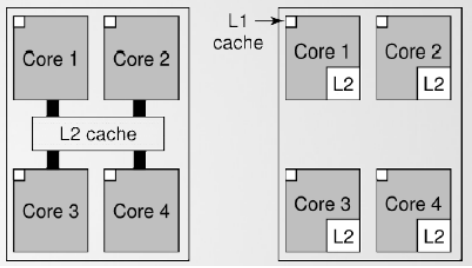
Permette di mantenere in contemporanea due stati di diversi processi nella stessa CPU, questo avviene quando un processo effettua una chiamata TRAP ed è in attesa di risposta dal SO, mentre ciò avviene il SO commuta il processore per servire un altro processo. Il Multithreading migliora l'efficienza, non si tratta di esecuzione parallela, ma abbatte i tempi del non uso della CPU.
Nei sistemi Multi-processori (sia fisici che virtuali) ogni processore ha il suo set completo, è indipendente dagli altri, può utilizzare tutto il set messo a disposizione dal nostro sistema.
I sistemi Multi-processori sono stati soppiantati da sistemi Multi-core. Uno dei problemi è quando due processi in esecuzione su core diversi, lavorano su dati comuni, potrebbero sorgere problemi gravi.

Nella **GPU** possiamo trovare anche centinaia di core, questo perché deve essere una componente efficace ma che comunque non ha la necessità di avere un eccessivo numero di task disponibili per funzionare al meglio, lavora su un insieme limitato di dati, ma ha una capacità di elaborazione elevatissima.

Esistono due tipi di **strutture della memoria del processore**:

- UMA (Uniform Memory Access): qualunque CPU/core accede alla memoria con gli stessi costi delle altre \ CPU/core.
- NUMA (Not Uniform Memory Access): i processi hanno aree di memoria predefinite dove vengono serviti in maniera più efficiente, così le CPU sanno dove servire determinati processi così da ottimizzare i tempi. Il problema è che così la memoria è suddivisa e se è necessario servire un numero grande di processi che richiedono la stessa area di memoria dovranno o aspettare il proprio turno oppure possono essere serviti in aree non predisposte con un dispendio maggiore di tempo.

Ogni processore può avere diversi **livelli di Cache**:
1° Livello→ Cache intrinseca al core.
2° Livello→ Può essere o condivisa dai core (Tecnologia INTEL) oppure suddivisa nei vari core (tecnologia AMD).
Nel caso la cache sia suddivisa (AMD) e un core modifica un dato su cui sta lavorando un altro core, sorge un problema di sincronizzazione interna che deve essere risolto dalla progettazione HW.



Dispositivi di I/O (Input/Output)

Anche i dispositivi di I/O interagiscono pesantemente con il SO e da questo devono essere gestiti. Sono provvisti di un *controllore* che fa comunicare il SO con il dispositivo: il SO comunica con le unità, il controllore abbassa il livello della comunicazione agevolando le funzioni dei dispositivi. Il SO trova valido aiuto nei Drivers, componenti software molto semplici creati per consentire l'interazione dispositivo-SO.

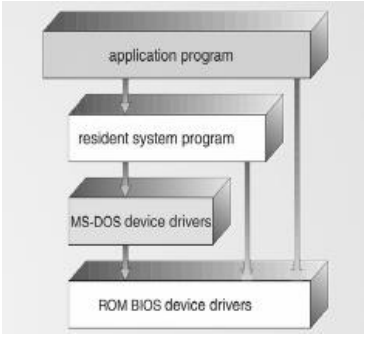
Le operazioni di I/O devono essere eseguite in modalità Kernel. I controller comunicano con il SO attraverso il PCI EXPRESS, che usa dei BUS seriali (e non condivisi).

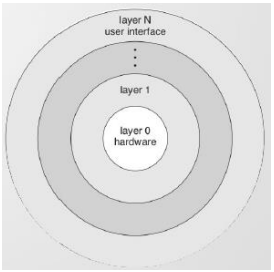
- Busy Waiting: Un programma dà il via ad una chiamata di sistema che il kernel traduce in una chiamata di procedura al driver appropriato. Dopodiché il driver avvia l'operazione di I/O, entrano in un ciclo finché il dispositivo di I/O non finirà il lavoro restituendo qualcosa. Fatto ciò il driver darà i risultati al kernel (se ce ne sono) e restituisce il controllo al chiamante. Questo metodo ha lo svantaggio di tenere occupata la CPU anche quando non è necessario farlo, privandola agli altri processi.

STRUTTURA DI UN SISTEMA OPERATIVO

Esistono diversi tipi di strutture di SO, ognuna con i suoi pregi e i suoi difetti.

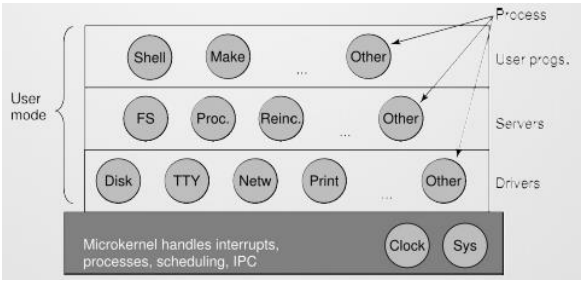
1) Monolitico: è la struttura più difficile, codice poco strutturato e progetto molto grosso.
   Struttura interna quasi inesistente e difficilmente modificabile: il SO consiste in un insieme di procedure che possono richiamarsi a vicenda. Per costruire il vero programma oggetto del SO prima tutte le procedure vengono compilate e in seguito sono legate tra loro tramite un linker di sistema. Non né esiste codifica né mascheramento: tutte le procedure sono visibili le une alle altre e possono liberamente interagire tra loro. Non ha supporto HW ed è poco gestibile.

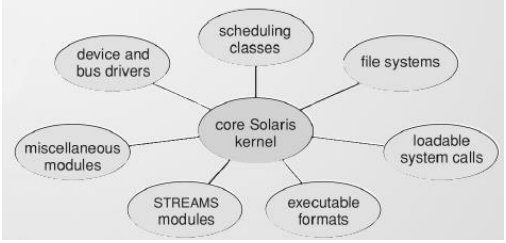




2) Sistemi a livelli: è possibile creare un sistema a livelli che utilizza funzioni degli strati inferiori per offrire servizi a strati superiori che li richiedono tramite chiamate TRAP (ogni volta effettuata la chiamata TRAP se ne controlla la validità). È un modello più semplice da controllare e da sviluppare (usa un incapsulamento tipo OOP). Principalmente ha problemi di prestazioni, dovuti alle numerose chiamate nidificate. I livelli possono essere rappresentati da cerchi concentrici.

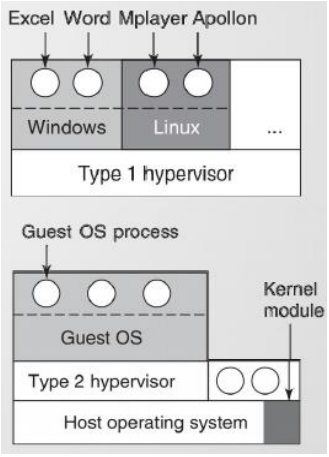
3) Microkernel: si occupa di microscheuling, memoria a IPC. Tutti gli altri servizi sono gestiti da altri moduli eseguiti in modalità utente. Il tutto è gestito tramite scambio di messaggi interni, ha un miglior design interno e stabilità.





4) Struttura modulare: il kernel principale ha funzionalità ridotte, posso caricare i moduli dinamicamente, in base alle mie esigenze, possono comunicare tra loro ed ogni modulo ha il suo utilizzo specifico. I moduli sono eseguiti in modalità utente. È una programmazione orientata agli oggetti (OOP).

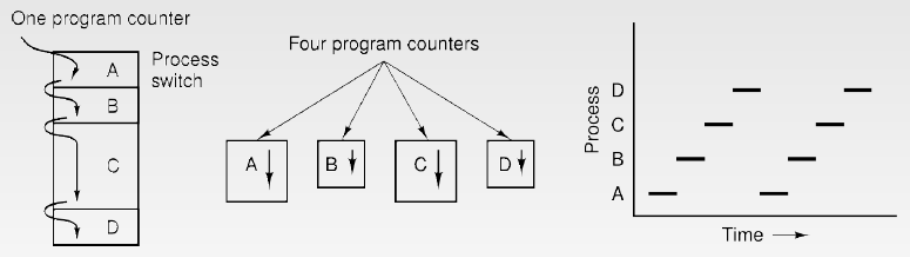
MACCHINE VIRTUALI



Il monitor della macchina virtuale (virtual machine monitor) gira direttamente sull'HW, si occupa della multiprogrammazione e fornisce al livello superiore l'opportunità di avviare più macchine virtuali contemporaneamente, queste virtual Machines (VM) sono delle copie esatte dell'HW della macchina, con tutti i suoi interrupt, le modalità user/kernel, ecc… Dato che ogni VM possiede il proprio HW, è possibile far girare un qualunque SO che sia compatibile con l'HW.
Una macchina virtuale, diversa da quelle viste fin ora, è la Java Virtual Machine(JVM). Questa è stata sviluppata da Sun Microsystem per poter interpretare il codice Java in tutte le macchine provviste della JVM, così da non avere problemi di esecuzione anche nel caso di invio dati.

PROCESSI

Tutto il SW, compreso il SO, è eseguito su un unico calcolatore, è organizzato in un certo numero di processi sequenziali o, in generale, processi.
Si può pensare che ogni processo possegga la propria CPU e venga eseguito in parallelo agli altri processi, ma sappiamo che in realtà la CPU switcha continuamente processo e li fa avanzare in "falso-parallelismo". Questo rapido cambio di contesto è chiamata multiprogrammazione.
Generalmente un processo è un attività che possiede un programma, dei dati in ingresso, dei dati in uscita ed uno stato. Un unico processore può essere condiviso tra vari processi, questo usa un algoritmo di schedulazione per determinare quando smettere di lavorare su un processo per servirne un altro.



CREAZIONE E TERMINAZIONE DEI PROCESSI

Un processo viene creato:

1) All'inizializzazione del sistema: all' inizializzazione del SO si avvieranno in automatico dei processi di background che offrono servizi al sistema (per esempio un processo per la gestione delle e-mail, sincronizzazione Drive, antivirus, processi per la gestione delle periferiche, ecc…), questi sono detti Demoni(Daemons). Questi processi possono essere analizzati usando il Task Manager di Windows.
2) Da parte di un processo Padre o da parte dell'utente: spesso un processo effettua delle chiamate di sistema per avviare processi secondari (figli) che lo aiutino nel suo lavoro (questo è utile quando il lavoro può essere facilmente distribuito ed effettuato in "parallelo"). Un esempio è il download-streaming presente sul Torrent, un processo ci consente di vedere un film mentre un altro processo lo sta scaricando.
   Quando un utente apre un programma o una cartella, si crea un nuovo processo che gestisce il tutto.

Un processo viene creato attraverso dei comandi particolari:

- In UNIX si usano *fork* e *execeve*: fork crea una copia del processo chiamante, il processo figlio ha la stessa immagine di memoria, gli stessi dati, le stesse stringhe di ambiente del processo genitore. Normalmente il processo figlio esegue una execeve per cambiare la sua immagine di memoria ed eseguire un nuovo programma.
- In Windows si usa *CreateProcess*: questa chiamata ha 10 parametri e gestisce la creazione di un processo da 0, gli assegna la memoria, il programma da eseguire, informazioni di priorità, attributi di sicurezza, bit di controllo, puntatori a strutture, ecc…

In entrambi i casi esiste il problema della condivisione delle informazioni in memoria: cosa succede se un processo A modifica delle informazioni su cui sta lavorando un processo B?

Una volta eseguito il suo compito, un processo deve essere terminato, questo succede quando:

- Uscita normale (volontario): *exit* (UNIX), *ExitProcess* (Win). Una volta che il processo ha terminato il proprio compito, effettua una system call per far notare al SO che ha finito.
- Uscita su errore (volontario): il processo riscontra un errore fatale, qualcosa che non riesce a gestire, un componente che manca per proseguire l'esecuzione. Quindi, in certi casi, avverte l'utente con un pop-up e si auto-termina.
- Errore critico (involontario): errori di programmazione, bug, divisioni per 0, accesso a memoria inesistenti, ecc… Sono errori che il processo non può gestire da solo, interviene quindi il SO: se è un errore che riesce a gestire e risolvere non è un grande problema, ma esistono errori non contemplati dal SO e che comportano un invio di segnale di chiusura al processo che si auto-termina.
- Terminato da un altro processo (involontario): Un processo con autorità necessaria invia sua system call richiedendo la terminazione di un altro processo attraverso determinati comandi (*kill* (UNIX), *TerminateProcess* (Win)).

STATO DI UN PROCESSO

Nella sua vita, ogni processo può trovarsi in 5 stati differenti (il primo e l'ultimo sono opzionali):

1. Creazione: Il processo viene creato, aspetta l'ammissione nel ciclo della CPU.
2. Pronto: il processo è pronto per essere scelto dallo schedulatore ed essere eseguito
3. In esecuzione: il processo è in esecuzione, è stato scelto dallo schedulatore, rimane in questo stato o finché termina il suo lavoro o finché viene interruptato.
4. Bloccato: il processo ha subito un interrupt, è in attesa di un qualche evento esterno per continuare il suo lavoro
5. Terminato: il processo ha finito il suo lavoro, non ha bisogno di compiere altre azioni e si termina.

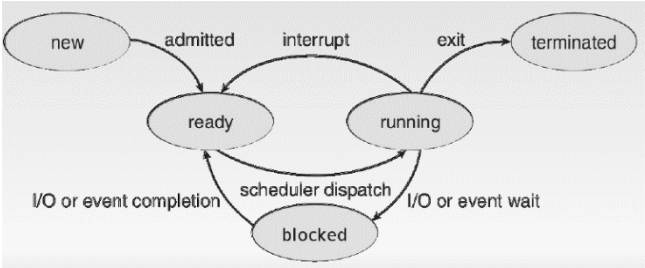
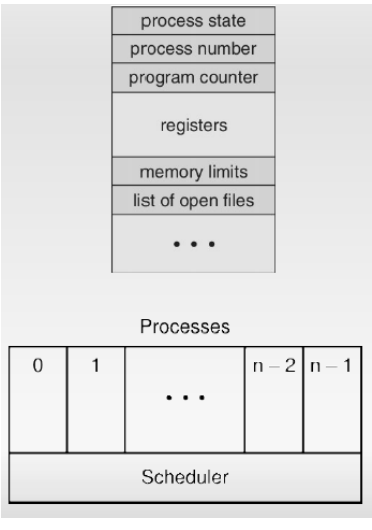


TABELLA DEI PROCESSI

Per implementare il modello a processi il SO mantiene una tabella (vettore di strutture) chiamata tabella dei processi (Process Table) dove ogni elemento di questa struttura rappresenta un processo ed è detto Process Control Block (PCB).
Ogni PCB contiene le informazioni del processo a cui si riferisce, il suo stato, il suo PC, il puntatore, l'allocazione di memoria, lo stato dei file su cui si appoggia, e molte altre cose utili al processo e alla sua esecuzione in modo tale che, quando ne sarà ripresa l'esecuzione, sembrerà che non sia mai stato fermato.

Com' è possibile dare l'illusione che una sola macchina dotata di una sola CPU faccia lavorare una molteplicità di processi contemporaneamente e sequenzialmente?
Ogni classe di dispositivi I/O ha associata una locazione detta vettore delle interruzioni (Interrupt Vector) che contiene l'indirizzo della procedura per la gestione delle interruzioni.

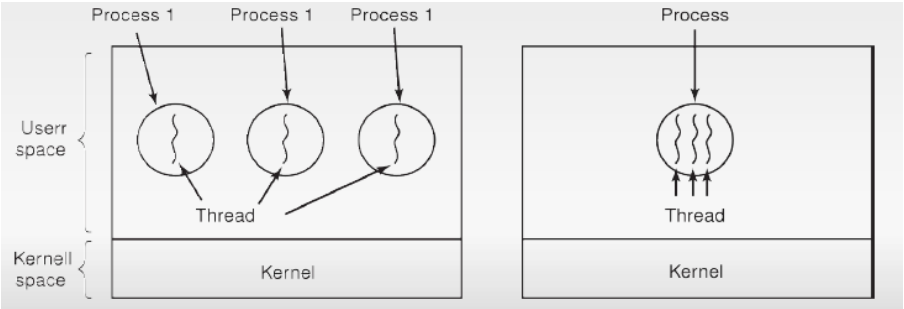


Se un dispositivo chiama un interruzione, il PC, lo stato e alcuni registri del processo corrente sono salvati nello stack corrente dell'HW dedicato alle interruzioni e la CPU salta all'indirizzo specificato dall'interrupt vector.

Schematizzando (esistono comunque variazioni da SO a SO):

1) Salvataggio dei registri;
2) L'HW carica un nuovo PC dal vettore delle interruzioni;
3) Salvataggio registri e impostazione di un nuovo stack;
4) Esecuzione procedura di servizio per l'interrupt (scritta in C);
5) Interrogazione dello scheduler per sapere con quale processo proseguire;
6) Ripristino dal PCB dello stato di tale processo (registri, mappa memoria);
7) Ripresa nel processo corrente.

THREAD



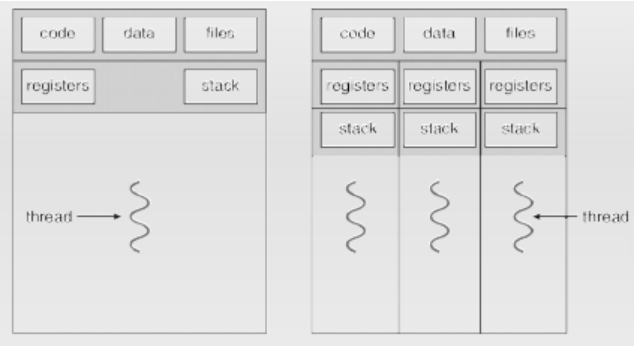
I processi dei SO odierni sono organizzati a thread, dei "processi leggeri" che cooperando portano avanti il lavoro del processo (flussi di esecuzione). Esistono sia processi con un solo thread sia processi che ne contengono molteplici. Il termine multithreading è utilizzato per descrivere la situazione in cui ad un solo processo sono associati più thread.

Ogni thread ha:

- Il suo PC, che tiene traccia della prossima istruzione da eseguire;
- Suoi registri, che mantengono le variabili utili;
- Uno stack, che contiene la storia dell'esecuzione e un elemento per ogni procedura ancora in esecuzione.

In definitiva i processi radunano le risorse e i thread sono le entità schedulate per l'esecuzione nella CPU. Thread dello stesso processo condividono risorse, spazio di indirizzamento e file aperti.

Multiprocessing o multithreading?



Quando un processo con thread multipli viene eseguito su un sistema con una sola CPU, i thread vengono eseguiti a turno; la CPU passa rapidamente avanti e indietro per i thread, è un lavoro più complesso il cambiare il contesto tra processi rispetto al cambiarlo tra thread dello stesso processo. Thread diversi in un processo non sono indipendenti come processi diversi, perché tutti i thread hanno esattamente lo stesso spazio di indirizzamento, cioè condividono anche le stesse variabili globali. Dal momento che ogni thread può accedere ad ogni indirizzo di memoria dello spazio di indirizzamento del processo, un thread può leggere, scrivere o persino cancellare completamente lo stack di un altro thread: non c'è protezione tra i thread perché è impossibile realizzarla e non dovrebbe essere necessaria. Oltre alla condivisione dello spazio di indirizzamento, tutti i thread condividono lo stesso insieme di file aperti, processi figli, allarmi eccetera.
Quindi quando si hanno dei lavori da eseguire correlati fra loro, è meglio utilizzare un unico processo con diversi thread, invece se i lavori non sono correlati è preferibile utilizzare diversi processi.

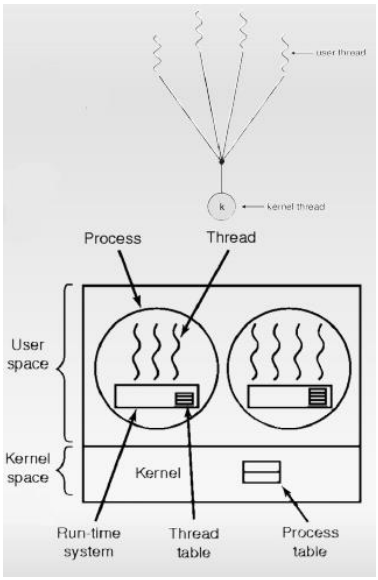
Operazioni tipiche sui thread (ne esistono altre):

- *thread_create*: un thread ne crea un altro;
- *thread_exit:* il thread chiamante termina;
- *thread_join*: un thread si sincronizza con la fine di un altro thread;
- *thread_yield*: il thread chiamante rilascia volontariamente la CPU.

Quando è presente il multithreading, normalmente i processi partono con solo un thread presente, che ha la capacità di creare nuovi thread richiamando una procedura dalla libreria, come ad esempio *thread_create* e un parametro che tipicamente specifica il nome della procedura che il nuovo thread deve eseguire. Non è necessario né possibile specificare qualcosa di nuovo sullo spazio di indirizzamento del thread appena creato, dal momento che questo, automaticamente, è in esecuzione nello spazio di indirizzamento del thread che l'ha creato (lo spazio di indirizzamento è condiviso). Con o senza relazione gerarchica (poche volte c'è una gerarchia), al primo thread viene restituito l'identificatore del thread creato, che permette di riferirsi per nome al nuovo thread. Mentre i thread spesso si rivelano utili, introducono un certo numero di complicazioni nel modello di programmazione.

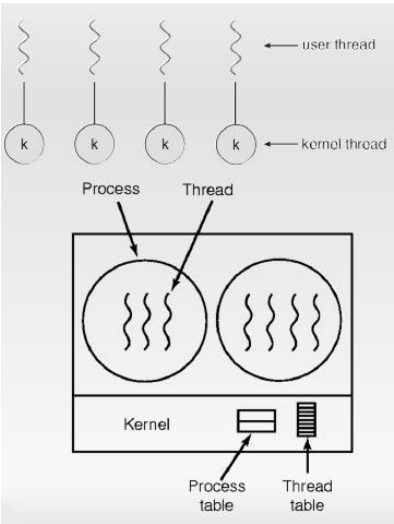
'Ma non ci bastavano i processi che erano in "falso-parallelismo"? dobbiamo metterci i thread che sono un "falso-parallelismo del falso-parallelismo" ??'
Ebbene stolto, come abbiamo visto i processi paralleli semplificano non poco il lavoro. Ma adesso abbiamo entità parallele che condividono gli stessi dati e lo stesso spazio di indirizzamento, rendendo possibile un elaborazione molto più veloce efficiente di prima… e non solo! Creazione/distruzione di un thread è molto più facile e veloce di quella di un processo (in certi sistemi anche 100 volte più veloce), il lavoro parallelo di thread permette una più efficace sovrapposizione di attività, infine nei processori multi-core i thread possono usare il "parallelismo puro". Convinto ora?

THREAD A LIVELLO UTENTE



È detto "modello uno a molti", utile nel caso il kernel non supporti i thread (molti SO del passato), infatti eseguendoli in questo modo il kernel non sarà a conoscenza della loro esistenza.
I thread sono eseguiti sopra un Sistema a tempo di esecuzione (Run-time System) che altro non è che una collezione di procedure che gestiscono i kernel (*thread_create, thread_exit*, *ecc..*).
Ogni processo ha la propria tabella dei suoi thread (thread table) che tiene traccia dei thread (stato corrente del thread, PC, registri, ecc…).
Quando un thread deve essere bloccato, perché in attesa di dati o di un altro thread, invia una richiesta di blocco al Run-time system. Se questa viene accolta lo stato (PC, registri, ecc…) del thread da bloccare viene salvati nella tabella dei thread, il Run-time system cerca un altro thread pronto all'avvio e ricarica i registri macchina (che si occupano dell'esecuzione delle istruzioni) con quelli del nuovo thread. Fatto ciò scambia lo stack pointer con il program counter e il thread appena pescato entrerà in esecuzione.

THREAD A LIVELLO KERNEL



È detto "modello uno a uno", in questo caso non serve un Run-time System, e la thread table è una sola ed è contenuta nel kernel. Quando un thread ha bisogno di creare, distruggere un altro thread esegue una richiesta al kernel che tramite TRAP la soddisfa. Tutte le tabelle contenute prima nei Run-time System sono ora nel kernel.
Se un thread si blocca il kernel (a sua discrezione) può eseguire un thread dello stesso processo oppure di un altro processo.

Differenze, pro e contro dei due modelli:
PRO:

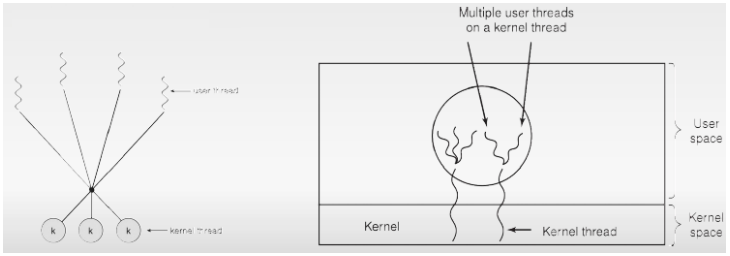
- Il livello utente non effettua TRAP e lo scheduling è gestito tra thread dello stesso processo.
- Nel livello kernel un thread bloccante non rallenta gli altri.

CONTRO:

- Nel livello utente le chiamate bloccanti sono un grosso problema, è possibile renderle non-bloccanti, ma è molto complicato; inoltre, essendo in modalità utente, se un thread non rilascia spontaneamente la CPU è impossibile bloccarlo, così facendo un unico thread blocca tutti gli altri thread del processo.
- Nel livello kernel le chiamate TRAP sono molto più costose e dispendiose, stesso discorso per creazione, distruzione dei thread.

THREAD MULTILIVELLO (SOLUZIONE IBRIDA)

Un modo per risolvere i problemi dei due modelli è creare un modello ibrido che ne prenda il meglio: esistono thread a livello kernel che gestiscono vari thread al livello utente. Il kernel gestisce solo una parte dei thread, gli altri sono al livello utente e son gestiti in gruppo dai thread al livello kernel.



La maggior parte dei SO moderni supporta i thread a livello kernel, e supportano quelli a livello utente attraverso determinate librerie.

COMUNICAZIONE TRA PROCESSI

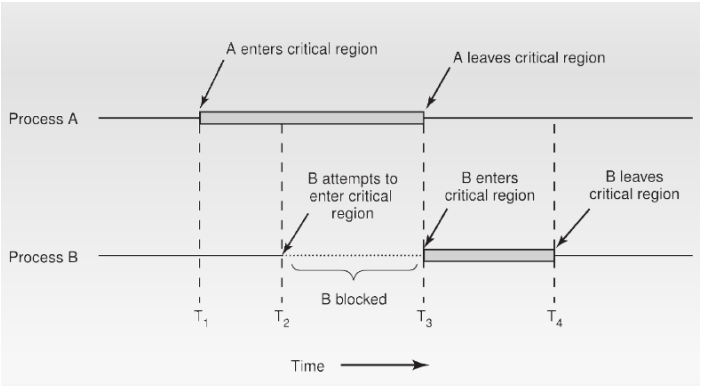
C'è la necessità che i processi comunichino tra loro in modo ben strutturato e senza utilizzare le interruzioni, cioè che in definitiva ci sia Interprocess Communication (IPC).
Ci sono tre questioni da considerare:

- Come un processo passa informazioni all'altro.
- Essere sicuri che due o più processi non si mettano l'uno sulla strada dell'altro.
- Mettere in sequenza i processi in modo adeguato.

È importante notare che le ultime due questioni si applicano anche ai thread, la prima è semplice da realizzarsi poiché i thread condividono uno spazio di indirizzamento.

CORSE CRITICHE E SEZIONI CRITICHE

In certi casi, processi diversi potrebbero condividere una parte di memoria dove possono leggere-scrivere contemporaneamente. Se due o più processi stanno leggendo o scrivendo su un qualche dato condiviso ed il risultato finale dipende dall'ordine in cui vengono eseguiti i processi, si presenta una Corsa critica (Race Condition).
L'unico modo per evitare le corse critiche è impedire a più di un processo alla volta di lavorare su risorse condivise.
Abbiamo bisogno della mutua esclusione (Mutual Exclution) per fare in modo che una risorsa non sia più accessibile se già in uso da qualcuno. Per semplificarci la vita basterà isolare le parti in cui i programmi accedono alla memoria comune e fare in modo che non collidano tra loro. Queste sezioni sono dette Regioni critiche (Critical Region) o Sezioni Critiche (Critical Section).



Per avere processi che cooperino tra loro e che usino dati condivisi in maniera efficiente, si devono soddisfare quattro condizioni (condizioni di Dijkstra):

1) Due processi non devono mai trovarsi contemporaneamente all'interno della sezione critica.
2) Non si deve fare alcuna ipotesi sulle velocità e sul numero delle CPU.
3) Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi.
4) Nessun processo deve aspettare indefinitivamente ("non si sa quanto") per poter entrare nella sua sezione critica.

MUTUA ESCLUSIONE CON ATTESA ATTIVA

Vediamo come effettuare la mutua esclusione:

- Disabilitare le interruzioni (interrupt):

  La soluzione più semplice è di permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua regione critica, e di riabilitarle non appena ne esce. Questo approccio ha però dei difetti: e se un processo disabilitasse le interruzioni e non le riabilitasse più? Potrebbe essere la fine del sistema. In più, in un elaboratore multiprocessore, con due o più CPU, la disabilitazione delle interruzioni avrebbe effetto solo sul processore che esegue l'istruzione *disable*, mentre le altre continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa. D'altra parte, però, è spesso conveniente che lo stesso kernel disabiliti le interruzioni per poche istruzioni, mentre sta aggiornando variabili o liste importanti, quindi il disable è uno strumento utile ma pericoloso che non è adatto al nostro scopo.

- Variabili di lock:

  Come secondo tentativo, vediamo una soluzione di tipo SW. Supponiamo di avere una sola variabile condivisa (di lock), con valore iniziale a 0. Quando un processo vuole entrare nella sua regione critica, controlla prima la sua variabile di lock e se vale 0, la mette a 1 ed entra, altrimenti aspetta che sia 0. Supponiamo che un processo legga la variabile di lock e veda che contiene 0, ma prima che possa metterla a 1, viene schedulato un altro processo che va in esecuzione e setta la variabile di lock a 1. Quando il primo processo torna in esecuzione, imposterà, a sua volta, la variabile a 1 e i due processi saranno contemporaneamente nella loro regione critica non garantendo la mutua esclusione. Il ricontrollo della variabile prima di eseguire le operazioni nella zona critica non funziona.

- Alternanza stretta:

  Si utilizza una variabile turn, inizialmente posta a 0 che tiene traccia del processo al quale tocca entrare nella sezione critica. Inizialmente, il processo A legge turno (=0) ed entra nella propria regione critica; anche il processo B trova turno a 0 ed entra in un piccolo ciclo, testando in continuazione turno per vedere quando sarà uguale a 1. Il testare continuamente una variabile si dice busy waiting (in questo caso si parla di Spin Lock). Un lock che usa l'attesa attiva si dice spin lock. Quando il processo A lascia la sezione critica, pone turno a 1 per permettere al processo B di entrare. Il processo 1 entra e termina rimettendo il turno a 0. A però non aveva bisogno di entrare nella sua area critica. B ha di nuovo bisogno di entrare in fase critica ma il turno è di A che non lo rilascia perché esegue codice non critico. Questa proposta non è ottima se un processo è molto più lento dell'altro. Questa situazione viola la terza condizione, ossia, un processo (B)è bloccato da un altro che non è nella propria sezione critica(A).

```
while (true) do
while (turn != 0) do
nothing
critical_region()
turn = 1
noncritical_region()
```

```
while (true) do
while (turn != 1) do
nothing
critical_region()
turn = 0
noncritical_region()
```

- Istruzioni TSL (Test and Set Lock) e XCHG (eXCHaGe):

  Questa è una proposta che richiede un piccolo aiuto anche dall'hardware.
  Molti calcolatori hanno un istruzione TSL, RX e LOCK.
  TSL mette il contenuto di una parola di memoria lock nel registro RX e poi memorizza un valore diverso da 0 all'indirizzo di memoria di lock. Le operazioni di lettura e memorizzazione della parola sono garantite indivisibili: nessun altro processore può accedere alla parola finché l'istruzione non è finita (è un istruzione atomica). La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire che altre CPU accedano alla memoria.

```
Enter_region:
TSL REGISTER,LOCK → Copia Lock in un registro e lo imposta a 1.
CMP REGISTER,#0   → Lock era a 0?
JZE Enter_region  → Se non è 0, Lock è stata impostata, quindi cicla.
RET               → Ritorna al chiamante, entra nella regione critica
```

```
Leave_region:
MOVE LOCK,#0 → Memorizza 0 in Lock.
RET          → Ritorna al chiamante.
```

  Per usare l'istruzione TSL, useremo una variabile condivisa, lock, per coordinare l'accesso alla memoria condivisa. Quando lock è a 0, qualunque processo può metterla a 1 usando l'istruzione TSL e poi leggere o scrivere nella memoria condivisa. Quando ha finito, il processo mette lock di nuovo a 0 utilizzando una normale istruzione move. Come può essere usata questa istruzione per garantire la mutua esclusione? La

prima istruzione copia il vecchio valore di lock su un registro e mette il valore di lock a 1, dopodiché il vecchio valore di lock viene confrontato con 0. Se non è uguale a 0, il blocco era già impostato, quindi il programma torna all'inizio e continua a controllare il valore. Per risolvere il problema della sezione critica, un processo chiama Enter_region che fa attesa attiva finché il blocco non è libero, poi acquisisce il controllo e termina; dopo la sezione critica, il processo chiama Leave_region che memorizza uno 0 in lock. L'istruzione TSL è una istruzione non privilegiata e quindi eseguibile anche in modalità utente e non solo kernel.

In sintesi:
TSL → Evoca un restrizione ad una locazione di memoria
Leggo le informazioni del REGISTER le salvo in una locazione, e setto il LOCK a 1.
Nel caso sfortunato dove 2 TSL siano eseguite contemporaneamente entrambe setteranno il lock a 1, ma solo una delle due ha letto 0 nel registro, e questa andrà avanti.

- Soluzione di Peterson:

dati N processi, ognuno eseguirà una routine (enter_region) prima di entrare nella sezione critica e una routine (leave_reagion) quando ne uscirà, il porcesso passerà un proprio identificativo alla funzione.
Other → identificativo dell'altro processo (1 se 0, 0 se 1).
Interested[process]→ manifestazione dell'interesse di entrare.
Turn → variabile di turno che indica l'identificativo del processo che entra (la imposto col mio ID).
Il controllo indica che se il processo A vuole entrare nella sezione critica e se B è già nella sua sezuone critica, A cicla senza fare nulla. In realtà la variabile "Interested" non è quella fondamentale, quella che lavora è il "Turn":
I processi A e B vorrebbero entare nella sezione critica, impostano "interested" a true.

```
int N=2
int turn
int interested[N]
function enter_region(int process)
other = 1 – process
interested[process] = true
turn = process

while (interested[other] = true and turn = process)
do nothing

function leave_region(int process)
interested[process] = false
```

Se il processo A imposta il proprio turno, e immediatanemte il processo B imposta il suo, nel while A vedrà che B è interessato e che il turno non è suo e quindi entrerà nella sezione critica mentre B cilerà finchè A non si imposterà il suo inrteresse su false, a quel punto entrerà.
C'è ancora busy waiting, può avere problemi nei moderni multiprocessori a causa del riordino degli accessi alla memoria centrale. La soluzione è facilmente applicabile a 2 processi, aumentandone il numero i controlli diventano molto più complessi (aumentano esponenzialmente).

Sia la soluzione di Peterson che quella che usa TSL, sono corrette, ma entrambe hanno il difetto di richiedere busy waiting, il che può provocare dei comportamenti inaspettati:
Consideriamo un calcolatore con due processi, H con priorità alta e L con priorità bassa; le regole di schedulazione sono tali che H viene mandato in esecuzione non appena si trova in ready. Ad un certo istante, mentre L si trova nella propria regione critica, H passa nello stato di ready. H comincia quindi l'attesa attiva, ma poiché L non viene mai scelto quando H è in esecuzione, L non ha mai la possibilità di lasciare la sezione critica, così H cicla all'infinito.
Questa situazione si chiama problema dell'inversione di priorità.

SOSPENSIONE E RISVEGLIO (SLEEP AND WAKEUP)

Il SO usa l'algoritmo di scheduling per decidere quale sia il processo successivo ad usare la CPU (bassa-alta priorità).
Le due primitive Sleep e Wakeup sono importantissime:
Sleep → Un processo chiede al SO di sospendere la propria esecuzione, è una pausa passiva che chiede di eliminare il suo nome dalla coda dei processi pronti e farsi inserire in un'altra coda in riferimento dell'avvenimento che si aspetta.
Wakeup→ Risveglio un altro processo

Problema Produttore-Consumatore

Come possiamo utilizzare Sleep e Wakeup per i nostri scopi? Consideriamo il seguente problema:
Ho N processi o thread che si scambiano informazioni. I processi produttori producono e inseriscono dati, i processi consumatori estraggono i dati dal buffer, tutto ciò in modo concorrente. L'unica variabile condivisa è il count, inizialmente impostato a 0. Quando il buffer è pieno, i produttori dovranno sospendere il proprio lavoro, e aspettare che i consumatori svuotino il buffer per risvegliarsi, analogamente se il buffer è vuoto i consumatori si addormenteranno in attesa dei dati dei produttori.
Il problema anche in questo caso sta' nelle Corse Critiche: se in un certo momento il consumatore legge count=0 si sospenderà, ma tra la lettura e la sospensione il controllo viene dato al produttore che produce ed aggiorna il count a count=1 e manderà un wakeup al consumatore. In questo modo il wakeup verrà perso dato che il consuamtore non è ancora sospeso, ma quando il controllo sarà restituito al consumatore, questo, avendo letto 0 e non essendosi accorto dell'aggiornamento, si sospenderà. Una volta che il buffer verrà riempito, il produttore andrà in sleep. Entrambi saranno sospesi per sempre. Per risolvere il problema si potrebbe aggiungere un bit di riserva dove, se viene mandato un wakeup mentre il processo è sveglio, il wakeup viene salvato e usato in seguito. Questo funziona con 2 processi, ma diventa veramente complicato con più thread e ha un maggior costo di bit di riserva.

PRODUCER:
produce_item→ crea un elemento
N → elementi massimi del buffer
Count →numero elementi
Insert_item →inserimento
Count=1→ prima era 0, il buffer era vuoto e quindi il consumer era in sleep

CONSUMER:
count=0→ buffer vuoto, entra in sleep.
Remove_item→ prelievo l'item.
Count = N-1→ prima il buffer era N cioè pieno, quindi sveglio il producer che era in sleep.

```
Function producer()
        while (true) do
                item = produce_item()
                if (count = N) sleep()
                        insert_item(item)
                count = count + 1
                if (count = 1)
                        wakeup(consumer)
```

```
function consumer()
        while (true) do
                if (count = 0) sleep()
                        item = remove_item()
                count = count - 1
                if (count = N - 1)
                        wakeup(producer)
                consume_item(item)
```

I SEMAFORI

È un miglioramento del sistema Sleep e Wakeup è una variabile intera positiva condivisa tra i processi che può essere modificata tramite due operazioni:

- Up: incrementa la variabile.
- Down: decrementa la variabile. Controlla che il valore sia maggiore di 0, se è così decrementa il valore (cioè consuma una delle sveglie che erano state salvate) e semplicemente continua, mentre se il valore è 0, il processo viene sospeso, per il momento senza completare la down.

Queste operazioni sono implementate in modo atomico, viene garantito che, se il semaforo è occupato, nessun' altro processo possa prendere il controllo e accedervi, finché la prima operazione non è terminata. L'atomicità è assoluta ed inevitabile per evitare le corse critiche.
I semafori non sono previsti dall'architettura, quindi devono appoggiarsi a funzioni e strutture dati, la garanzia di atomicità è data da operazioni effettuate dal kernel in modalità kernel, si disabilitano gli interrupt ad opera del SO per il tempo necessario dell'esecuzione dell'operazione dell'up-down. Nel caso si utilizzino CPU multiple, ciascun semaforo deve essere protetto da una variabile di lock e si deve utilizzare l'istruzione TSL. La disabilitazione degli interrupt non crea alcun problema dato che le operazioni di Up-Down sono veloci da eseguire e occupano veramente poco tempo alla CPU.

Problema Produttore-Consumatore con i Semafori

Questa soluzione utilizza tre semafori:

- Pieno (Full): per controllare il numero di elementi occupati (inizializzato a 0), vuoti per controllare il numero di elementi
- Vuoto (Empty): per controllare il numero di elementi vuoti (inizializzato a N)
- Mutex: per garantire la mutua esclusione (inizializzato a 1).

I semafori inizializzati a 1 e che vengono usati da due o più processi per assicurarsi che di volta in volta solo uno di essi possa entrare nella propria sezione critica, sono chiamati semafori binari o semafori Mutex. Se ogni processo chiama una down subito prima di entrare nella regione critica ed un up subito dopo esserne uscito, è garantita la mutua esclusione. Il semaforo mutex è utilizzato per la mutua esclusione, mentre l'altro uso dei semafori è per garantire la sincronizzazione. I semafori vuoti e pieni sono necessari per garantire che certe sequenze di eventi si verifichino o no in un certo ordine. Questa soluzione reintroduce il busy waiting, ma limitato agli accessi al semaforo e riguarda frammenti di codice di granularità inferiore rispetto alle regioni critiche generali e di conseguenza ha durata breve e prevedibile.

```
function producer()
        while (true) do
                item = produce_item()
                down(empty)
                down(mutex)
                insert_item(item)
                up(mutex)
                up(full)

function consumer()
        while (true) do
                down(full)
                down(mutex)
                item = remove_item()
                up(mutex)
                up(empty)
                consume_item(item)
```

TSO o XCNG hanno il solo problema dello spin lock.

I MUTEX

I mutex sono ottimi per gestire la mutua esclusione di una risorsa o di codice condiviso, sono facili e semplici da implementare. Un mutex è una variabile che può assumere i valori di "bloccato" o "non bloccato". Di fatto basterebbe un bit per rappresentarlo, ma in pratica lo si rappresenta con un intero, si usa intendere lo 0 come non bloccato e tutti gli altri valori come bloccato. Ci sono due procedure:

```
Mutex_lock:
TSL REGISTER,MUTEX  →  Copia Mutex in un registro e lo imposta a 1
CMP REGISTER,#0     →  Mutex era a 0?
JZE ok              →  Se era 0 Mutex era non bloccato, quindi termina
CALL thread_yield   →  Mutex è occupato, schedula un altro thread
JMP mutex_lock      →  Riprova più tardi
ok:RET              →  Ritorna al chiamante, entra nella regione critica
```

- Mutex_lock: quando un thread deve accedere ad una regione critica e il mutex non è già bloccato da altro, richiama questa procedura e si assicura il pieno controllo della risorsa.
- Mutex_unlock: quando il thread che è nella regione critica ha terminato, richiama la procedura rendendo il comando agli altri thread.

```
Mutex_unlock:
MOVE MUTEX,#0  →  Memorizza 0 in mutex
RET            →  Ritorna al chiamante
```

Dato che i Mutex sono tanto sempici, sono facilmente implementabili nello spazio utente, ma cosa li rende mgliori delle soluzioni viste fin ora?

Nei Mutex, se il processo non può entrare perché il Mutex è occupato, non entra in busy waiting, ma lascia il posto e viene schedulato un altro thread.

I FUTEX

Una variante sono i futex, uno strumento ben preciso usato nei sistemi linux. Ciò nasce da un'osservazione: l'idea è che l'operazione di sincronizzazione è sempre più frequente e avere implementazione più snella possibile migliora le prestazioni. Abbiamo già visto un modello più snello, il TSL e XCHG che non si appoggia sul SO, ma che ha il problema dello spin lock, problema risolto col semaforo. È possibile usare le TRAP, ma ha il suoi costi. Nel blocco del Mutex troviamo lo spin lock. Nel caso dei semafori quando non c'è contesa in una struttura dati, usandoli si effettua di fatto una chiamata di sistema che cambia la modalità della CPU sprecando tempo. I futex sono un connubio di entrambi gli scenari. Sono implementati con una doppia componente, una di kernel e l'altra di libreria. Nei futex, in modalità utente posso verificare se c'è contesa tramite una variabile LOCK gestita come visto prima, ma se mi accorgo che c'è contesa e non riesco a prendere il controllo del LOCK, chiede aiuto alla componente kernel del futex che blocca il thread che ha fatto la chiamata al futex. Quindi se ho poca contesa, non ho particolare bisogno del kernel: basta controllare la variabile LOCK, ma se per caso sono in una contesa, allora chiedo sostanzialmente al SO di bloccarmi e di rimuovere il mio PCB dalla coda degli in esecuzione.

I MONITOR

Concetto più evoluto dei semafori, ma con limitazioni. Usare i semafori è un'operazione delicata, i monitor sono una versione più semplificata.
I monitor rappresentano un tipo astratto di dato, un monitor ha un suo stato interno, con variabili che lo rappresentano, e una serie di metodi che operano su questo. Si avvicina molto al concetto di oggetto, ma il monitor è particolare. Il primo difetto è il vincolo di accesso ai dati, per cui le variabili interne al monitor possono essere usate solo dalle procedure interne del monitor, che di fatto è una pratica normale nella programmazione, ciò vale anche per le variabili esterne. Quindi i dati, le variabili e li metodi sono tutti dentro il monitor e i metodi non possono accedere ad eventuali dati esterni.
La mutua esclusione nei monitor si ottiene perché ci può essere solo un flusso di esecuzione che accede al monitor.
Può essere implementato con gli strumenti che già conosciamo. Questi strumenti sono offerti come astrazione, ma sotto sotto è il SO stesso che li offre, non il C# o il JAVA. Per evitare l'accesso ed avere più flussi che accedono al monitor, basta usare un LOCK sul monitor. Posso avere funzioni down(L) e up(L) per escludere l'accesso al monitor. In JAVA viene fatto tutto automaticamente dal compilatore, basta concentrale le sezioni critiche dentro lo stesso monitor.
Nota: i monitor sono strettamente legati alla programmazione multithread, ma nei monitor abbiamo un vincolo dello strumento stretto. I monitor si possono usare all'interno di un singolo processo. Infatti, nel modello a più processi che condividono uno spazio di memoria, abbiamo parlato di questi processi in modo generico, non chi li scrivesse o come fossero imparentati, ma di fatto un P1 può essere scritto da A scritto in C#, mentre P2 è scritto da B ma scritto in Java, eppure possono comunicare. Ciò non è vero nel caso dei monitor perché è strettamente connesso al linguaggio e chi lo può usare è solo il codice del mio programma stesso, quindi ha un modello più restrittivo.

SCAMBIO DI MESSAGGI

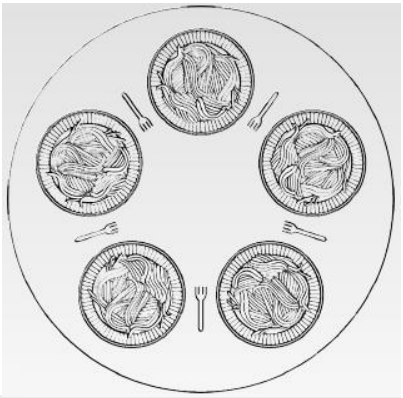
E' uno strumento analogo ai semafori offerto dal SO, non ho vincoli e se ho due processi, P1 e P2, possono avere due linguaggi diversi. Questo metodo di comunicazione tra processi usa due primitive, send e receive, che, come i semafori e a differenza dei monitor, sono chiamate di sistema invece che costrutti del linguaggio. Come tali, possono essere facilmente messe in procedure di libreria, come send(destinazione, messaggio) o receive(sorgente, messaggio).

```
function producer()
    while (true) do
        item = produce_item()
        receive(consumer, msg)
        build_msg(m,item)
        send(consumer, msg)
```

```
function consumer()
    for N times do
        send(producer, msg)
    while (true) do
        receive(producer, msg)
        item=extract_msg(msg)
        send(producer, msg)
        consuma(item)
```

SEND spedisce un messaggio ad una determinata destinazione e RECIVE riceve un messaggio da una determinata sorgente (o da *ANY,* qualunque sorgente, se al ricevente non interessa una particolare sorgente). Se non è disponibile nessun messaggio, il ricevente potrebbe bloccarsi finché non arriva un messaggio; in alternativa, può terminare immediatamente con un codice d'errore.
I sistemi a scambio di messaggi però hanno molti problemi e problematiche di progetto che non si presentano con i semafori o i monitor, in particolare se i processi che comunicano sono su macchine diverse connesse attraverso una rete; ad esempio, i messaggi possono essere persi dalla rete. Per prevenire la perdita di messaggi, il mittente e il destinatario possono concordare che non appena un messaggio viene ricevuto, il destinatario spedisce indietro uno speciale messaggio di acknowledgement (conferma dell'avvenuta ricezione); se il mittente non ha ricevuto la conferma entro un certo intervallo di tempo, ritrasmette il messaggio.
Consideriamo ora cosa succede se il messaggio stesso è stato ricevuto correttamente, ma la conferma viene persa. Il mittente ritrasmetterà il messaggio, così che il destinatario lo riceverà due volte: è essenziale che il destinatario possa distinguere un nuovo messaggio dalla ritrasmissione di uno vecchio. Di solito questo problema viene risolto mettendo numeri di sequenza consecutivi in ogni messaggio originale; se il destinatario ottiene un messaggio che porta lo stesso numero di sequenza del messaggio precedente, saprà che è un duplicato che può essere ignorato. I sistemi a scambio di messaggi devono anche trattare il problema dell'assegnazione dei nomi ai processi, in modo che il processo specificato in una chiamata send o receive non sia ambiguo. Anche l'autenticazione è un problema nei sistemi a scambio di messaggi: come fa il cliente a dire se sta comunicando con il file server reale, e non con un impostore?
All'altro estremo della gamma, ci sono anche problematiche di progetto che sono importanti quando il mittente e il destinatario sono sulla stessa macchina, tra le quali ci sono le prestazioni. La copia di un messaggio da un processo all' altro è sempre più lenta di un'operazione su un semaforo o di un accesso al monitor.
Cheriton (1984), ad esempio, ha suggerito di ridurre la dimensione dei messaggi in modo da contenerli nei registri della macchina, e di usare poi tali registri per lo scambio di messaggi.

PROBLEMA DEI 5 FILOSOFI



Il problema può essere formulato abbastanza semplicemente come segue: cinque filosofi sono seduti attorno ad un tavolo tondo e ciascun filosofo ha un piatto di spaghetti. Gli spaghetti sono così scivolosi che per mangiarli ogni filosofo deve avere due forchette e fra ogni coppia di piatti vi è una forchetta. La vita dei filosofi alterna periodi in cui essi pensano ad altri in cui mangiano. Quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta che gli sta a sinistra e di quella che gli sta a destra, una alla volta ed in un ordine arbitrario. Qualora riesca a prendere entrambe le forchette, mangia per un po' e, successivamente, depone le forchette e continua a pensare.

Soluzione ovvia: la procedura prendi_forchetta aspetta fino a che la forchetta specificata è disponibile e ne prende possesso.

Sfortunatamente, la soluzione ovvia è sbagliata. Supponete che tutti e cinque i filosofi prendano la loro forchetta sinistra nello stesso istante: nessuno di loro sarà più in grado di prendere la forchetta di destra e, di conseguenza, ci sarà uno stallo.

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2

monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            wait(self[i])

    function put_forks(int i)
        state[i] = THINKING;
        test(left(i));
        test(right(i));

    function test(int i)
        if ( state[left(i)] != EATING and state[i] = HUNGRY
         and state[right(i)] != EATING )
            state[i] = EATING
            signal(self[i])

function philosopher(int i)
   while (true) do
      think()
      dp_monitor.take_forks(i)
      eat()
      dp_monitor.put_forks(i)
```

Potremmo modificare il programma in modo tale che dopo aver preso la forchetta di sinistra, il programma esegua un controllo per vedere se la forchetta di destra è disponibile. Se non lo è, il filosofo rimette a posto quella di sinistra, aspetta per un po' e poi ripete l'intero processo. Anche questa proposta non funziona, sebbene per un motivo diverso. Con un po' di sfortuna, tutti i filosofi potrebbero cominciare l'algoritmo contemporaneamente, prendere le loro forchette di sinistra, accorgersi che la forchetta di destra non è disponibile, rimettere giù la forchetta di sinistra, aspettare, riprendere la forchetta di sinistra simultaneamente e continuare così per sempre. Una situazione come questa, nella quale tutti i programmi continuano ad essere in esecuzione per un tempo indefinito, ma nessuno di essi compie dei veri progressi, è nota come starvation (letteralmente, morte per fame).

Adesso si potrebbe pensare: "E se i filosofi aspettassero un tempo casuale anziché un tempo fisso fra un tentativo e l'altro di accedere alle forchette, la probabilità che tutto possa andare avanti a bloccarsi anche per una sola ora sarebbe molto bassa." Questa osservazione è vera e in quasi tutte le applicazioni riprovare più tardi non è un problema. Comunque, in alcune applicazioni si vuole una soluzione che funzioni sempre e che non possa fallire a causa di una serie sfortunata di numeri casuali.

La soluzione ultima appare la seguente: prima di prendere possesso delle forchette un filosofo dovrebbe fare una down su mutex (semaforo). Dopo aver deposto le forchette dovrebbe fare una up su mutex. Da un punto di vista pratico, questa soluzione presenta un problema: solo un filosofo alla volta può mangiare.

La soluzione seguente presenta, invece, il massimo grado di parallelismo per un numero arbitrario di filosofi. Usa un vettore, affamato, per tenere traccia se un filosofo sta mangiando, pensando o sia affamato. Un filosofo può passare nello stato in cui mangia se nessuno dei suoi vicini sta mangiando. Il programma usa un vettore di semafori (uno per ogni filosofo) cosicché i filosofi affamati possano bloccarsi se le loro forchette risultano bloccate.

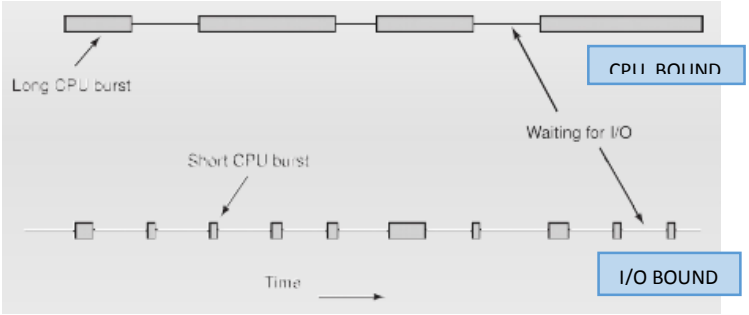
PROBLEMA DEI LETTORI E SCRITTORI

Oltre il problema dei 5 filosofi, un altro problema è quello della gestione di lettura e scrittura su un Database, conosciuto comunemente come problema dei lettori e scrittore. Come possiamo gestire chi scrive e chi legge su una stessa struttura dati?

La soluzione è fare in modo che quando un lettore ottiene l'accesso, esegue una down sul semaforo DB, i lettori successivi incrementano un contatore RC. Uscendo i lettori decrementano il contatore, l'ultimo eseguirà una UP sul semaforo, consentendo l'accesso ad uno scrittore, se esiste. In questo modo più lettori possono operare in parallelo poiché il loro operato non collide.

Per evitare che lo scrittore attenda tempo indefinito ad aspettare che indefiniti lettori leggano, basterà mettere in coda i lettori dietro lo scrittore.

SCHEDULING



Il compito dello Scheduling è scegliere il prossimo processo ad essere eseguito, a dedicargli quali e quante risorse (CPU, memoria, dispositivi I/O, ecc…)

CPU Burst → La CPU viene assegnata ad un determinato processo finché questo non decide di cederla oppure finché il SO non la riprende. È il tempo di utilizzo della CPU.

CPU-bounded → Processi che usano molta CPU rispetto all'uso di dispositivi I/O.

I/O-bounded → Processi che usano molto i dispositivi di I/O rispetto all'uso della CPU.

In un sistema reale, abbiamo processi di entrambi i tipi, quindi conviene avere in esecuzione un mix di processi in modo tale che non si abbia uso intensivo di CPU, o uso intensivo dell'I/O, ma un uso moderato di entrambi contemporaneamente.

Quando è necessario schedulare?

- Quando nasce un nuovo processo: in questo caso è necessario sapere quando eseguirlo, subito o dopo il processo padre.
- Quando muore un processo: quale deve essere il prossimo ad essere eseguito?
- Quando il processo in esecuzione si blocca: quando questo è in attesa del I/O, quando è bloccato ad un semaforo, quando riscontra un errore.

Se il clock della CPU fornisce interruzioni periodiche (ogni 50, 60, X Hz) si deve prendere una decisione di schedulazione ogni interruzione di clock o ogni K interruzioni.

Esistono due tipi di algoritmi di schedulazione:

- NONPREEMPTIVE (senza Prerilascio) → L'esecuzione del processo può essere interrotta o dalla CPU, o dal processo stesso che cede il controllo. In questo caso ad ogni interruzione di clock viene ripreso il processo precedente.
- PREEMPTIVE (con Prerilascio) → Il processo è lasciato in esecuzione per un certo periodo di tempo (l'unità di tempo che usa lo schedulatore per regolare l'esecuzione dei processi è il clock della CPU), dopo questo lo schedulatore deciderà cosa eseguire

Il Dispatcher è la componente del SO che si occupa del cambio di contesto (CPU – I/O).

OBBIETIVI DEGLI ALGORITMI DI SCHEDULING

Ambienti differenti: batch, interattivi e real-time.

- Obiettivi comuni:
  - Equità nell'assegnazione della CPU;
  - Controllo sull'applicazione della politica di scheduling;
  - Bilanciamento nell'uso delle risorse;
- Obiettivi tipici dei sistemi Batch:
  - Massimizzare il throughput (o produttività);
  - Minimizzare il tempo di turnaround (o tempo di completamento, tra la richiesta e la conclusione);
  - Minimizzare il tempo di attesa (tenere la CPU in costante utilizzo);
- Obiettivi tipici dei sistemi interattivi:
  - Minimizzare il tempo di risposta;
- Obiettivi tipici dei sistemi real-time:
  - Rispetto delle scadenze (evitare la perdita di dati);
  - Prevedibilità.

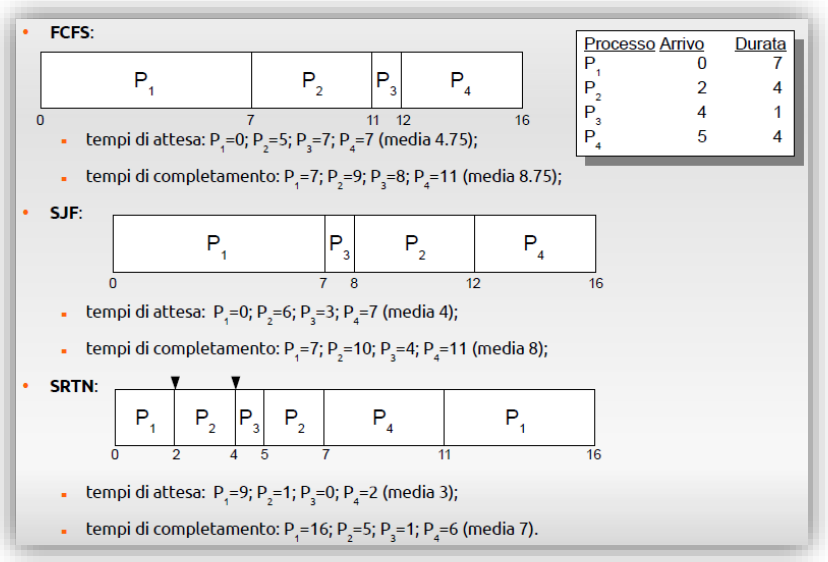
È normale che ambienti diversi richiedano diversi algoritmi di schedulazione, poiché ciò che lo scheduler deve ottimizzare dipende molto dal tipo di sistema. Tre ambienti ben distinti sono:

1) Batch
2) Interattivo
3) Real Time

SCHEDULING NEI SISTEMI BATCH

Fondamentalmente nei sistemi Batch non è necessaria una pronta risposta dato che non ci sono utenti in attesa al terminale: è possibile usare sia algoritmi preemptive che non-preemptive e si preferirà un approccio che riduce gli scambi tra processi migliorando le prestazioni.

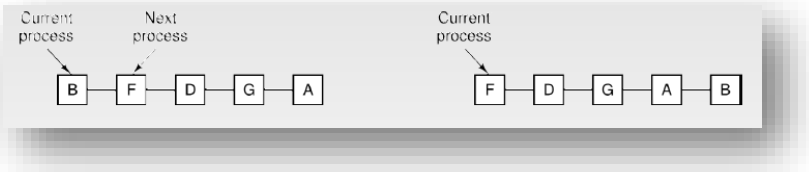
- First-Come First-Served (FCFS):
  Sicuramente il più semplice tra tutti gli algoritmi, è senza prerilascio, in cui ai processi viene assegnata la CPU nell'ordine in cui l'hanno chiesta fino a quando non la rilasciano. Il problema è che senza prerilascio un processo potrebbe monopolizzare la CPU rallentando l'esecuzione di processi potenzialmente veloci.
- Shortest Job First (SJF):
  È un algoritmo senza prerilascio che presuppone la conoscenza dei tempi di esecuzione dei singoli processi. Se in un certo momento un nuovo processo (o più processi) deve essere schedulato, questo non sarà schedulato in relazione ai processi già presenti, ma solo in relazione a quelli che devono essere schedulati nello stesso momento.
- Shortest Remaining Time Next (SRTN):
  È una versione con prerilascio dell'algoritmo precedente (SJF). Quando il ciclo di clock termina il processo con meno tempo di esecuzione rimanente viene eseguito, questo permette a nuovi processi di entrare in gioco se hanno tempi di esecuzioni minori rispetto a quello del processo in esecuzione.



SCHEDULING NEI SISTEMI INTERATTIVI

In un ambiente Interattivo il prerilascio è essenziale per evitare che un processo si impossessi della CPU impedendo l'accesso agli altri.

- Scheduling RoundRobin (RR):
  È uno degli algoritmi più vecchi, semplici ed imparziali e facili di implementare, è una versione con prelazione del FCFS. Ad ogni processo è assegnato un tempo di esecuzione detto quanto di tempo. Se al termine del suo quanto il processo è ancora in esecuzione la CPU viene rilasciata e riassegnata, se il processo entra in blocco prima della fine del suo quanto accade lo stesso. Il RR è semplice da implementare: c'è una lista di processi in coda, appena un processo termina il suo quanto viene sospeso e spostato alla fine della coda.
  L'unica variabile da settare è quindi il quanto: se lo setto troppo breve (1 ms) i tempi di esecuzione saranno soverchiati dai tempi di cambio del contesto (cambio registri, aggiornamento liste e tabelle, ricarica della cache, ecc..), se lo setto troppo lungo (100 ms) sarebbe come se i processi monopolizzassero la CPU. Un buon compromesso di durata del quanto è tra i 20 e i 50 ms.

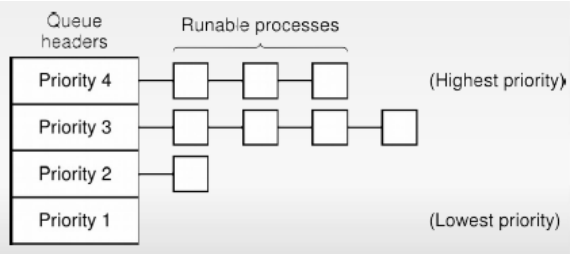


- Scheduling a priorità:
  Esistono processi con priorità differente, questa dipende da vari fattori: caratteristiche, modalità di utilizzo delle risorse (CPU bounded meno priorità, I/O bounded più priorità).
  La priorità può essere considerata come un numero (0 bassa priorità – 100 alta priorità per esempio), e può essere statica o dinamica:
  - Statica → Nasce e muore col processo.
  - Dinamica → Dipende da altri fattori (ereditarietà, determinati momenti della vita del processo, altri processi in funzione, ecc..)

Per evitare che possa subentrare una Starvation dei processi minori si usa la priorità dinamica (con Starvation si intende quando un processo non riceve mai il controllo della CPU a causa della sua bassa priorità e diventa "affamato").
Potremmo considerare una priorità dinamica legata al quanto di tempo utilizzato precedentemente dal processo: se il processo ha sfruttato al massimo il tempo e le risorse la priorità non varierà, altrimenti se ne sfrutta meno la priorità si abbasserà.
Altro modo è dare più priorità a task più veloci da completare.
In un sistema basato su prelazione potrebbe essere che il processo perda o acquisti priorità in base se si aggiungono o vengono a mancare i presupposti di esecuzione.

Esistono tecniche che tolgono priorità a processi che usano troppo spesso CPU:
Aging → Più la CPU viene usata da parte di un processo HP (High Priority), più la sua priorità scenderà man mano, quando un processo a bassa priorità in coda supera il processo in esecuzione, si effettua uno switch e processi con meno priorità potranno entrare in esecuzione.

È possibile partizionare la coda dei processi pronti in base alle loro prioriità, sono suddivisi dinamicamente: nelle classi più alte ci sono i programmi interattivi, nelle classi più basse ci sono servizi minori.
Nelle classi più alta si imposta un RR breve, nelle classi più basse si può usare un FCFS.



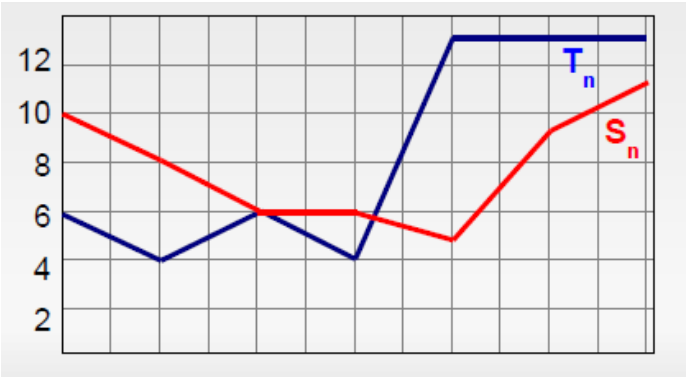
Un altro modo è partizionare il tempo alle varie classi di priorità:

- 60% alta priorità
- 25% media priorità
- 13% bassa priorità
- 2% bassissima priorità

Si risolve il problema dello Starvation in questo modo.

Shortest Process Next (SPN):

Noi non possiamo accedere alla lunghezza del task, ma possiamo confrontare la lunghezza del CPU burst, se cerchiamo di prevedere statisticamente la lunchezza del prossimo CPU burst (che dovrebbe essere simile alla lunghezza di quello precedente) è possibile schedulare meglio la lista dei processi. Ancora meglio è calcolare una media esponenziale che aggiorna la stima del porssimo CPU burst basanosi su tutti i precedenti valori.



$S_{n+1} = S_n (1 - a) + T_n a$

S_n → Media ponderata delle Stime precedenti
T_n → Durata effettiva dell'ultimo CPU burst
a → Valore arbitrario compreso tra 0 e 1, più è alto meglio si adatterà all'ultima stima ma oscillerà di più

Scheduling garantito:

Garantisce un uso equo della CPU stabilendo una percentuale di utilizzo. Avendo N processi, assegno ad ogni processo 1/N tempo di esecuzione.

Scheduling a lotteria:

Concretizza e semplifica l'idea precedente, crea periodicamente una lotteria e distribuisce un tot di ticket in base alla priorità che verranno estratti ciclicamente, la CPU verrà usata da un ticket alla volta, e all' estrazione il ticket viene rimosso. Rende possibile l'utilizzo della CPU da parte di tutti, si possono avere processi cooperanti che di scambiano i ticket.

Scheduling fair-share:

Ripartisce equamente la CPU tra tutti i gruppi di processi (utenti) del sistema. Utenti che lanciano più processi rispetto ad altri avranno la stessa CPU dedicata, indipendentemente dal numero di processi.

SCHEDULING DEI THREAD

- Thread utente:
  - Ignorati dallo scheduler del kernel;
  - Per lo scheduler del sistema run-time vanno bene tutti gli algoritmi non-preemptive visti;
  - Possibilità di utilizzo di scheduling personalizzato.
- Thread del kernel:
  - O si considerano tutti i thread uguali, oppure
  - Si pesa l'appartenenza al processo;
  - Lo switch su un thread di un processo diverso implica la riprogrammazione della MMU.

SCHEDULING SU SISTEMI MULTICORE

In questi sistemi abbiamo più core a disposizione e possiamo sfruttarli contemporaneamete, possiamo attuare diverse strategie:

- Multielaborazione asimmetrica:

  In queso modello abbiamo un CORE MASTER (SERVER) che gestisce le strutture dati interne, i dispositivi di I/O, ecc… Gli altri core sono SLAVE, che eseguono i compiti del SERVER.
- Multielaborazione simmetrica (Simmetric Multi Processing "SMP"):

  Tutti i core sono uguali e si ripartisce il carico equamente sui processori. In questo caso subentra la gestione delle Race Condition dato che le strutture dati sono condivise.
  C'è un'unica coda dei processi pronti (l'accesso alla coda deve essere effettuato in maniera oculata dato che più core potrebbero voler pescare un processo contemporaneamente, inoltre potrebbe succedere che un core debba aspettare altri processi e si iberni) oppure code separate per ogni core (più usato ma con il problema del bilanciamento del carico).

Con più core ho più cache private, se un processo viene schedulato in momenti diversi in core diversi, questi non possono accedere alla cache di altri core, perdendo tempo e lavoro per ripopolare la propria cache. Detto ciò è preferibile rischedulare un processo sullo stesso core, stabilendo un affinità del processo con un determinato core.

- Affinità debole:

  Il SO non conta molto l'affinità processo-core, è possibile violarla.
- Affinità forte:

  Il SO prende in serio conto l'affinità processo-core,non la viola. Sorge il problema del bilanciamento del carico: nel caso le code siano separate e in un core ci sia una lunga coda di processi con affinità forte il carico è fortemente sbilanciato.

Per questo motivo esistono dei meccanismi di migrazione di processi da una coda all'altra.

Migrazione guidata (push) → Esiste un servizio del SO che verifica il bilanciamento del carico, sposta i processi nel caso di uno sbilanciamento.

Migrazione spontanea (pull) → Il core stesso esamina le varie code e sottrae i processi ad una coda più carica.

In genere si implementano entrambe le strategie contemporaneamente poiché solo una delle due non riesce a gestire tutti i problemi che potrebbero sorgere in un sistema.

SCHEDULING SU WINDOWS 8

Il sistema Windows è abbastanza semplice, basato sulle applicazioni, è Preemptive basato su classi di priorità.
Api Win32 per la gestione delle priorità dei thread:
SetPriorityClass: chiamata di sistema che permette di assegnare una classe di priorità ad un processo (contenitore di thread).
SetThreadPriority: Permette ad un thread di assegnare a se stesso o ad un thread fratello una priorità relativa all' interno del processo, creando una gerarchia di priorità all'interno del processo. La priorità relativa è temporanea e può solo incrementare rispetto alla priorità base.

classi di priorità

priorità relativa	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

La priorità è compresa quindi da 0 a 31, l'algoritmo di selezione del processo da eseguire prima identifica la classe non vuota di priorità più alta e poi il processo con maggiore priorità secondo un Round-Robin interno alla classe.

I motivi per i quali la priorità relativa viene modificata sono:

1) L'utilizzo del quanto del tempo: se qusto viene usato parzialmente o il thread si blocca.
2) Ripresa da un I/O: si blocca in attesa di un I/O, si ha un salto di +3 di priorità relativa.
3) Processi in primo piano: se la GUI è in primo piano rispetto ad altri oltre ad avere più priorità, ha un quanto di tempo aumenteato.

Gli ultimi thread sono fittizzi, lo "zero page thread" (thread a priorità zero) è una routine periodica che entra in gioco quando il processore è libero.
A causa della scelta netta della schedulazione di determinati processi potrebbe dar luogo ad un inversione di priorità: se un processo di HP è bloccato da un processo a LP che non è in esecuzione perché è troppo LP, quest'ultimo subisce un Auto Boost temporaneo che ne incrementa la priorità sbloccando il primo processo.

SCHEDULING SU LINUX

Tutto è schedulato per "task", che può essere o un processo o un threead. È in realtà un agglomerato di flussi che condividono qualcosa.
Chiamata di sistema clone: duplica il flusso in esecuzione, questo può o meno condividere cose col padre, ha diversi flag che le donano granularità e potenzialità.
VM → condivide la memoria se impostato ad 1.
FS → avranno working directory condivisa se impostato ad 1.
FILES → files condivisi se impostato ad 1.
PARENT → si eredita l'ID del genitore se impostato ad 1.
Fork → Duplica fedelmente il padre condividendone i flussi, è il caso in cui tutti i flag sono settati ad 1.
Linux gestisce i thread come task, posso identificare i thread tramite degli ID:

- Process identidier (PID): per retrocompatibilità.
- Task identifier (TID): univoco per ogni task.

Thread distinti ma dello stesso processo hanno TID diversi ma PID uguali, un porcesso clonato ha sia PID che TID diverso dal padre

Scheduler O(1) di Linux

È stato abbandonato perché era difficilmente scalabile e le sue prestazioni non erano all'altezza dei processori multicore.

Usa 3 classi di priorità:

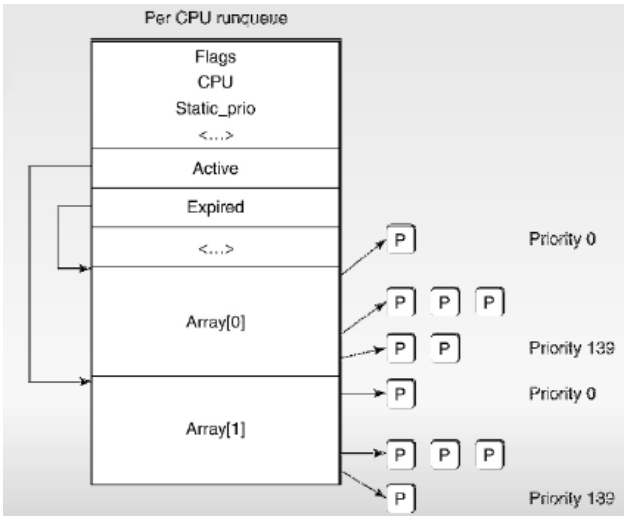
- Real-time FIFO (non preemptive): lascia ai processi il controllo.
- Real-time round-robin (preemptive): tenta la mediazione tra i due sistemi, è una real-time con un quanto di tempo.
- Timesharing (preemptive): è la classe usata.

Usa delle priorità numeriche:

- Real-time da 0(più alta) a 99;
- Timesharing da 100 a 139 (più bassa);

Che possono variare in modo:

- Statico (nice value: -20 a +19) l'utente può solo diminuire la propria priorità (aumentando il numero) dando più spazio agli altri, gli amministratori possono alzarla e diminuirla.
- Dinamico (da -5 per I/O boundness a +5 per CPU boundness) ai processi i/O boundness, che hanno quanti più piccoli rispetto ai CPU boundness, viene data più priorità.

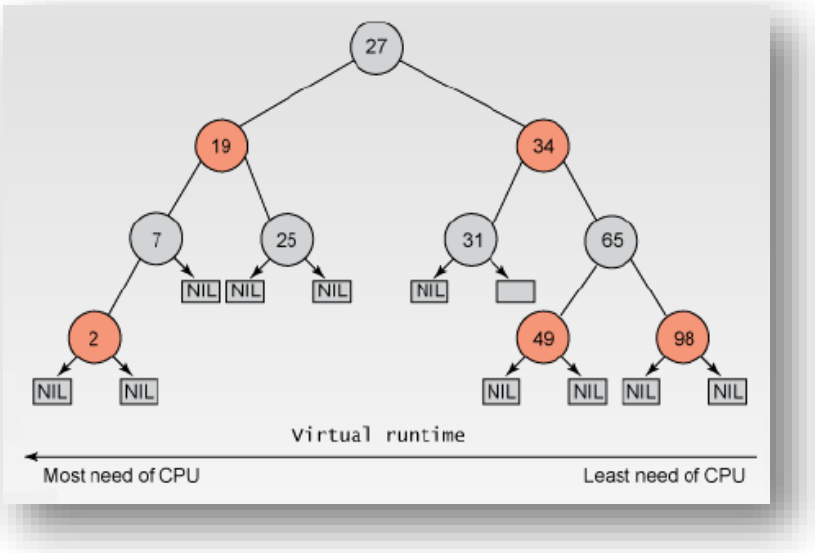


Per ogni CPU si ha una RUNQUEQUE, questa gestisce una serie di code, 140 code attive (active) e 140 code scadute (expired).
L'algoritmo di scheduling va tra le code attive e prende la coda con priorità più alta selezionandone il primo processo e eseguendolo fino all'esaurimento del suo quanto di tempo. Nel caso succeda qualcosa e non riesca a completare il suo quanto di tempo, al riavvio il quanto rimanente viene ripristinato (non succede su WIN). Nel caso esaurisca il quanto viene spostato nella coda expired. Una volta consumati i processi nella coda attiva, si swappano i due array e le code expired diventano code active rinnovando i proprio quanti di tempo.
I quanti oscillano tra i 5ms (corti) ai 800ms (lunghi)
Le classi ad alta priorità avranno quanti più lunghi, le classi a bassa priorità ce li avranno più corti (si parla sempre di processi in Timesharing).
La complessità è costante perché non dipende dagli elementi e non bisogna effettuare ricerche.
È una runqueque specifica per ogni CORE, è prevista la migrazione spontanea (dal processo) o guidata (attraverso demone) da CORE a CORE.
È necessario proteggere la struttura dati attraverso uno SpinLock, c'è quinid il busywaiting, ma è necessario per la consultazione della struttura dati.

Scheduler CFS di Linux

Questo scheduler è stato criticato per la gestione delle code, assegnazione e gestione delle priorità e fumosità.
Un tentativo di avere un progetto di qualtià superiore ha portato allo scheduler CFS (Completely Fair Scheduler). Questo cerca di dare risorse equamente, tenendo traccia delle risorse usate per un processo, delle risorse che gli spetterebbero.
Usa il Virtual Runtime (VRT), cioè il conteggio della quantità di tempo in cui il processo ha usato le risorse, tutti i processi debbono poter utilizzare la CPU con la stessa quantità di tempo, di volta in volta si esegue quindi il porcesso con virtuial runtime più basso.

La struttura dati usata è un albero di ricerca rosso-nero (autobilanciante; inserimenti, ricerche, cancellazioni con complessità logaritmica).
L'elemento minimo starà nel PCB in basso a SX. È uno scheduler con prelazione. Man mano che il processu utilizza la CPU il suo VRT cresce e se non è più il minimo si cambia il posto e si switcha il processo in esecuzione. C'è un tempo minimo di esecuzione e una granularità del controllo cofigurabile.
Viene costantemente mantenuto il VRT più basso, periodicamente viene fatta una normalizzazione di tutte le etichette, sottraendo il minimo a tutti, evitando di far crescere le variabili in modo incontrollato. Quando entra un nuovo thread questo avrà un VRT uguale al minimo o al valore medio (dipende dall'implementazione).



Il sistema da la possibilità ai processi che si bloccano spesso di avere un alta priorità (quando si interrompe il suo VRT resterà inalterato, in modo naturale si troverà vicino alla parte SX dell'albero, se non proprio sarà il prossimo processo in esecuzione).
Le priorità sono applicate come fattori di decadimento, sono delle costanti moltiplicative che influiscono il VRT. I processi a bassa priorità li avranno più alti cosicchè il VRT cresca più rapidamente lasciando prima la CPU agli altri.
Nella versione 2.6.24 è stato introdotto lo scheduling di gruppo, il fair share. Il sistema ha la possibilità di conteggiare e fare scelte di scheduling per gruppi di processi. Nei sistemi multicore è tutto molto in linea con ciò che abbiamo visto,ogni CORE ha il suo albero e la migrazione può essere spontanea o guidata.

GESTIONE MEMORIA

L'ideale per un PC sarebbe avere una memoria veloce, non volatile, spaziosa e non costosa… Tutto ciò però non esiste e non è fattibile, quindi le macchine odierne si appoggiano a tre tipi di memoria

1) Una piccola quantità di memoria molto veloce e molto costosa (CACHE).
2) Gigabyte di memoria principale volatile a velocità media e prezzo medio (RAM).
3) Terabyte di disco di memorizzazione non volatile, lento ed economico (HD).

La parte del SO che gestisce l'interazione tra queste memorie è il gestore della memoria, il suo compito è tener conto della memoria utilizzata e libera, allocare memoria quando un processo lo richiede, deallocarla quando non è più utile.
La RAM è il livello più basso direttamente utilizzabile dalla CPU.

MEMORIA CENTRALE (RAM)

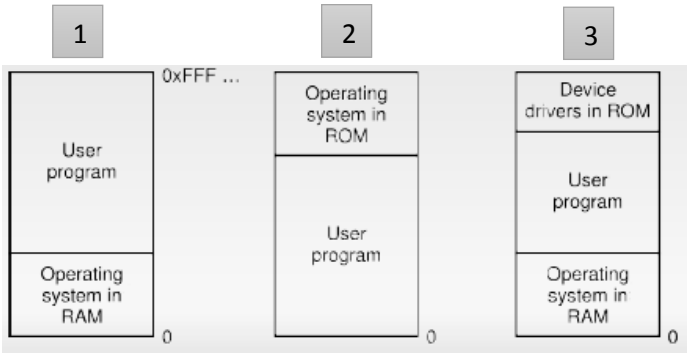
Esistono due sistemi della gestione della memoria:

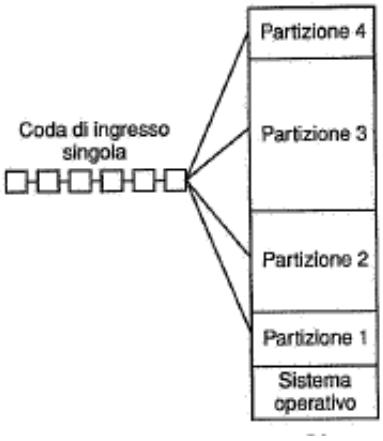
1) Sistemi che non effettuano swapping, più semplici.
2) Sistemi che effettuano Swapping o paginazione, cioè che spostano i processi avanti e indietro dalla memoria centrale.

SISTEMI SENZA SWAPPING E SENZA PAGINAZIONE:

I programmi utilizzano direttamente indirizzi fisici, sono eseguiti uno alla volta e solo al termine del primo può entrare in esecuzione il successivo.
Il sistema può essere organizzato in tre modi differenti:

1) Il SO stà nella parte bassa della RAM.
2) Il SO stà nella ROM (Read Only Memory).
3) Il SO stà nella RAM e il Gestore dei dispositivi nella ROM.

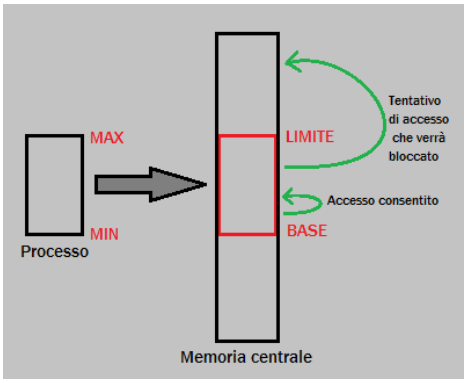




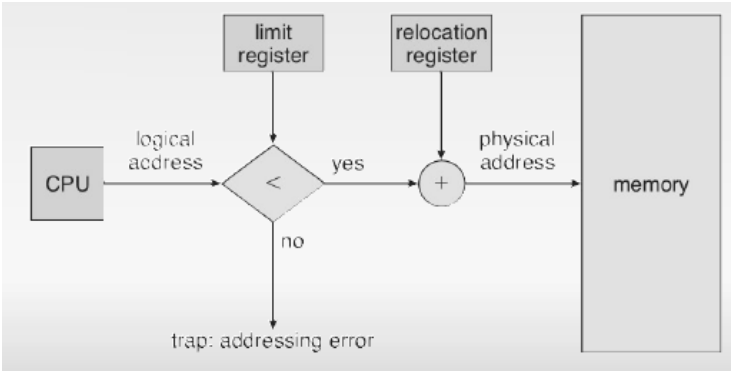
Nel caso di Multiprogrammazione in questi sistemi si presenta il problema di far coesistere più processi in esecuzione concorrente: mentre un processo è in attesa di un operazione di I/O, un altro che richiede la CPU può essere eseguito. I questo caso suddividimo la memoria in n partizioni, ognuna con grandezza diversa: processi piccoli saranno assegnati a partizioni piccole e processi che richiedono più memoria saranno assegnati a partizioi più capienti. Ci sarà una coda di processi che attendono che gli sia assegnata una partizione, nel caso di numerosi processi piccoli però non riusciremo a sfruttare le aree di memoria riservate ai processi più pesanti. Per risolvere questo problema potremmo usare un contatore interno al processo che indica quante volte è stato scartato perché la memoria richiesta da lui era troppo piccola rispetto alle aree disponibili, appena si raggiunge un valore K, il processo dovrà entrare in un area di memoria (la più piccola possibile) ed essere eseguito.

GESTIONE DELL ALLOCAZIONE (RILOCAZIONE) E PROTEZIONE

Cosa succede se devo avviare un processo? Dentro il processo ci sono allocazioni di memoria che rimandano ad altre del processo stesso (es una variabile X rimanda all'indirizzo di memoria 22 del processo), ma appena devo eseguirlo e lo sposto in memoria centrale, gli indirizzi non corrisponderanno più (se sposto il processo all' indirizzo 10000, la variabile X dovrebbe puntare al 10022, ma in realtà punta a 22 che è un area esterna al processo!). Questo può comportare gravi problemi sia al processo che al resto del sistema (ad esempio incrementare, decrementare CANCELLARE una variabile potrebbe distruggere il SO!). La soluzione è sommare a tutti i rimandi a variabili l'indirizzo MIN in cui il processo è stato allocato (22+10000= 10022 OK!).

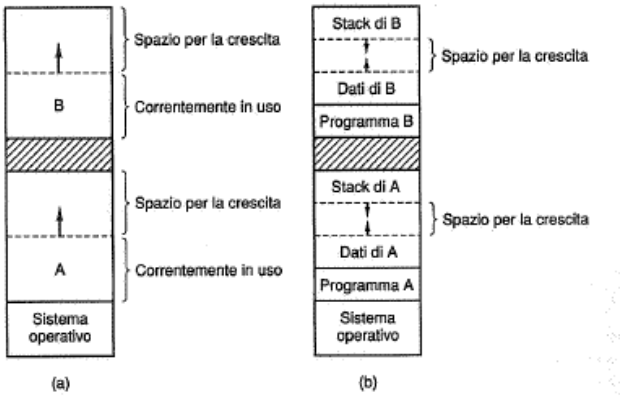


Dal punto di vista della protezione ciò non è abbastanza, il processo può facilmente riuscire ad arrivare fuori dalla memoria a lui asssegnata (basta che punti già inizialmente ad un indirizzo fuori dal suo range), per ovviare a questo problema si definisce un limite superiore oltre il quale il processo non può accedere (che corrisponde al MAX del processo), se il processo tenta di accedervi viene bloccato, se invece è entro il range, viene sommato il valore della BASE e il valore sarà rimesso in range.



SWAPPING

Con i sistemi Batch oganizzare la memoria in partizioni fisse è semplice ed efficiente, ma nei sistemi Timesharing (i più diffusi) non è disponibile abbastanza RAM per mantenere tutti i processi attivi, quindi si ricorre ad una tecnica di Swapping, che consiste nel caricare interamente un processo dal disco, eseguirlo per un certo periodo di tempo e rispostarlo nuovamente sul disco (sia nel caso che sia terminato o meno).



Un'altra strategia, detta Memoria Virtuale, rende possibile caricare solo una parte del processo sulla RAM e eseguirlo ugualmente (la vedremo più avanti).

La forza dello Swapping stà nell' "allocazione dinamica": non essendoci aree prestabilite di memoria, posso riempire la RAM con più processi possibili, siano essi tutti piccoli, tutti grandi o misti. L'unico problema da tenere in mente è che potrebbe succedere che a processi già in memoria potrebbe servire ulteriore memoria nel corso dell' esecuzione: per evitare spostamenti di processi già allocati in altre aree di memoria è bene allocare una quantità extra di memoria per ogni processo, prevedendo così ulteriori richieste.

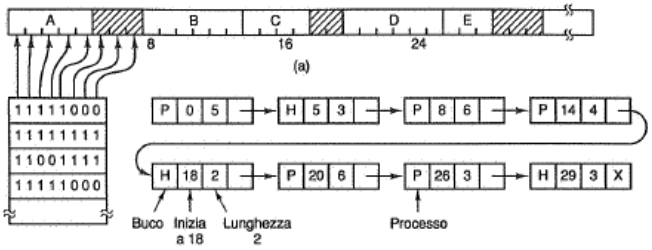
Inoltre è preferibile fare in modo che le aree di memoria disponibile siano contigue, così da evitare frammentazione.

GESTIONE DELLA MEMORIA

Con un assegnazione dinamica della memoria è necessaria una gestione adeguata dal parte del SO, in particolare è necessario sapere quali aree di memoria sono allocate e quali sono libere. è possibile assolvere a questo compito in due modi:

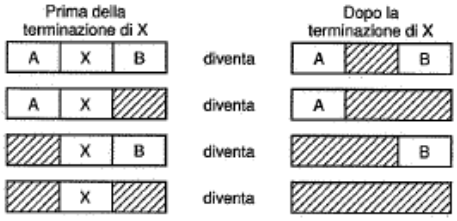
- Mappe di bit (Bitmap):

  La memoria è divisa in unità di allocazioni, che possono essere di varia grandezza in base alle scelte progettuali: più è piccla l'allocazione, più grande sarà la Bitmap; più è grande l'allocazione più saranno presenti sprechi di memoria dato che non allocherò la memoria in modo preciso. Una volta stabilita la grandezza dell'unità di allocazione la nostra Bitmap avrà una sua grandezza fissa, infatti la dimensione della Bitmap dipende solo dalla grandezza della memoria fisica e dalla dimensione dell'unità di allocazione. Quando si deve caricare un processo di K unità, si deve trovare una sequenza di almeno K unità di memoria liberi nella Bitmap (rappresentati come 0). Il problema principale di questo metodo è quando la memoria necessaria è a cavallo tra due parole e questo rallenta molto il processo.



- Lista concatenata dei blocchi:

  Si mantiene una lista doppiamente concatenata di elementi che indicano se l'area di memoria è allocata o libera, è utile ordinare la lista per indirizzi di memoria così da facilitarne l'aggiornamento quando un processo termina o si arresta. La lista è doppiamente concatenata per facilitare la ricerca di aree libere e la loro fusione. Per allocare la memoria ad un certo processo possiamo usare diversi metodi:



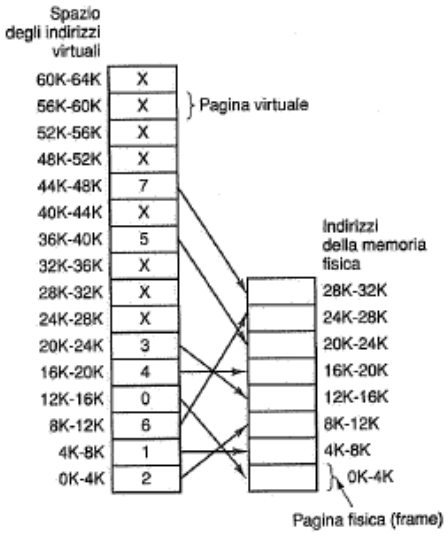
  - First Fit: viene considerato il primo posto abbastanza capiente disponibile, è molto veloce, ma potrebbe comportare sprechi di memoria.
  - Next Fit: identico al First Fit, si ricorda della prima allocazione libera trovata. Quando sarà necessario allocare memoria per un nuovo processo riprenderà la ricerca da li e non dall' inizio della lista. Sorprendentemente poco più lento del first fit
  - Best Fit: Cerca in tutta la lista l'allocazione più adatta, cioè quella che offre la memoria più piccola necessaria. È più lento del first fit, e sorprendentemente spreca più memoria: il best fit lascia libere piccole porzioni di memoria inutilizzabili.
  - Quick Fit: si mantengono due liste, quella dei processi allocati e quella delle aree di memoria libera. Al momento dell'allocazione si cerca l'area più adatta. In questo metodo in particolare si mantengono dei puntatori alle aree di memoria più richieste (es: 4K, 12K, 21K, ecc...)e si istanza un area in base alla grandezza del processo.

MEMORIA VIRTUALE

Inizialmente, quando i programmi divennero troppo grandi per essere caricati in memoria centrale, il programmatore doveva dividere il suo programma in moduli da far girare uno dopo l'altro sulla RAM, ciò ovviamente era complicato e molto lungo (Overlay). Il metodo per ovviare a questo problema divenne noto come Memoria Virtuale. L'idea di base consiste di caricare le parti del programma che devono eseguire job, lasciando il resto sul disco, le parti del programma vengono quindi coninuamente caricate e scaricate dalla RAM in base alle necessità.

PAGINAZIONE

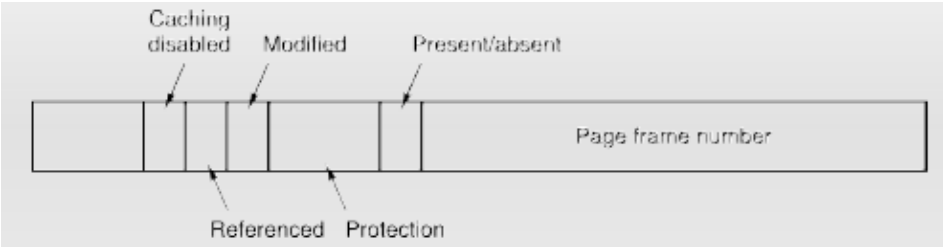
Il principale problema della gestione della memoria è raccordare gli indirizzi virtuali con gli indirizzi fisici. Questa mappatura è gestita dalla Memory Management Unit (MMU), che possiamo immaginare frapposta tra CPU e memoria. Si suddivide la memoria fisica in frame (o pagine fisiche) e la memoria virtuale in pagine, ad ogni pagina corrisponde un frame. Per tener traccia della mappatura si usa la tabella delle pagine, una funzione che mappa le pagine virtuali con il numero di frame che contiene la pagina. Esiste un bit di presenza che dice se la pagina è mappata in memoria. Nel caso la pagina incontri l'errore di Page Fault, subentra il SO che si occupa di portare il contenuto della Page Fault in uno slot libero.
Di tabelle delle pagine ne esistono tante quante sono i processi.
Usualmente il comando per prendere il contenuto di uno spazio di memora e assegnarlo a REG è: MOV REG, X
Dove X è l'indirizzo di memoria virtiuale. Quando l'MMU riceve questo comando calcola a quale allocazione di memoria fisica corrisponde e manda quest'indirizzo in uscita sul BUS.
Come funziona la mappatura?
Alla MMU viene passato un inidirizzo (virtuale) da prendere,
per esempio gli viene pssato il comando MOV REG, 20, cioè un programma cerca di accedere alla memoria di indirizzo 20, che cade nella pagina (virtuale) 0 (da 0 a 4095 K) la quale, secondo una funzione di traduzione (che potrebbe ricercare l'indirizzo libero più vicino), corrisponde alla pagina fisica 2 (8192 K – 12287 K). Di conseguenza trasforma l'indirizzo in (8192+20=) 8212, mandandolo in uscita sul BUS. Effettivamente l'MMU ha trasfoermato gli indirizzi virtuali da 0 a 4K in indirizzi fisici tra 8K e 12K.



In questo modo è evidente che non tutte le frame possono essere mappate nelle pagine, quindi abbiamo bisogno di un bit di controllo presente/assente che ci indica se la paggina è mappata o meno.
Nel caso di un page fault (nel caso si tenti di accedere ad una pagina non mappata) l'errore è gestito dal SO, questo prende una delle pagine meno usate, ne salva il contenuto sul disco, la rialloca, fa riferirire la pagina nella memoria appena liberata, cambia i bit di controllo e quindi fa ripartire l'istruzione interrotta.
Per esempo se cerco di accedere all'indirizzo virtuale 25K, questo avrà bit di presente/assente = 0 (infatti non è mappato ad un indirizzo fisico).

DETTAGLI SU UNA VOCE DELLA TABELLA DELLE PAGINE

Non in tutti i sistemi gli elementi delle tabelle sono uguali, ma seguono pressappoco tutti lo stesso schema.



- Numero di pagina fisica (Page Frame Number): Il campo più importante, lega la frame alla pagina.
- Bit presente/assente (Present/absent): Se è uguale ad 1 l'elemento è valido, se è 0 si andrà incontro ad un Page Fault.
- Bit Protezione (Protection): Indica i tipi di permessi: se è 0 si ha diritto di lettura/scrittura, se è 1 solo lettura.
- Bit Modificato (Dirty Bit): Inizialmente la pagina ha questo bit settato a 0, comunica se la pagina è stata mai modificata dopo esssere stata portata in memoria, cioè se la copia della pagina della memoria centrale è uguale a quella della pagina che sta leggendo (se è "sporca"). Viene modificato quando l'MMU vede che la pagina viene aperta in modalità scrittura. Quando è 0 la pagina può essere scartata tranquillamente, se nel caso fosse 1, il SO

deve prima preservare il vecchio contenuto, sincronizzando il contenuto con la memoria centrale.

- Bit di Referenziamento (referenced): inizialmente posto a 0, se si fa un qualunque referenziamneto (lettura/scrittura) a quella pagina. Serve a decidere quali pagine in memoria sacrificare e quali no, servono a fare una statistica su quali pagine sono utilizzate e quali no. L'informazione è relativa ad un giro di clock.
- Bit per disabilitare la chace (Chaching disabled): Può essere utile per disabilitare una pagina in cui avvengono mappature di I/O.
- Bit di validità o allocazione: un processo può utilizzare tutto lo spazio di indirizzamento, ma materialmete nell'allocazione gia si perde della memoria. Il bit di validità indica se quella pagina è già stata allocata per il processo.

TABELLA DEL FRAME

Ha tante voci quanti sono i frame, ogni voce dice se un frame è allocato o meno e rende possibile distinguere i frame liberi da quelli occupati e sapere da cosa sono occupati. Viene chiamata in causa nel momento in cui è necessario allocare un nuovo processo o ulteriore memoria di un processo già esistente.
Per mantenere un accesso rapido si potrebbe proevedere un set di registri ad alta velocità nell' MMU idonea a contenetre la tabella stessa.
Questo modello appesantisce il context switch man mano che le tabelle crescono di grandezza.
Un'altra idea è avere un registro in memoria fisica ed un puntatore che punta all'indirizzo fisico ove si sviluppa la tabella. L'MMU calcola l' I-esimo elemento e crea il fetch per la traduzione di quell'elemento. Ogni volta che devo accedere alla memoria, devo fare un referenziamento al registro e un altro tramite l'MMU, rendendo l'accesso molto più lento dato che sono 2 accessi alla memoria.

USO DI MEMORIA ASSOCIATIVA

Si usa una cache la Translation Lookaside Buffer (TLB) (o memoria associativa), una piccola tabella contenuta all'interno dell'MMU. Contiene un numero limitato di voci, queste sono i registri più utilizzati, si frappone tra MMU e Memoria Centrale. Viene consultata dall'MMU: questa prima vede se ciò che cerca è nella TLB (dove la ricerca è ottimizzata), altrimenti fa un fetch alla memoria centrale.
La TLB viene gestita dall' HW e dovrebbe contenere il set di pagine attualmente in uso dal processo in esecuzione.
In una TLB trovo:

- Numero di pagina virtuale: è il campo di ricerca chiave.
- Numero di frame: ciò che cerco.
- Maschera dei diritti: sempre a disposizione dell'MMU.
- Dirty Bit: può succedere che il bit non sia propriamente sincronizzato, e quindi devo tenerlo in conto.
- Bit di validità

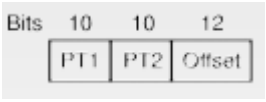
È possibile vincolare voci della TLB in modo che non cambino (ad es. pagine che si riferiscono al SO)

Quando avviene un content switch tra un processo ed un altro, è necessario svuotare laTLB?
Se uso l'Adress Space IDentifier (ASID) distinguo le pagine dei singoli processi e non è necessario svuotarla.

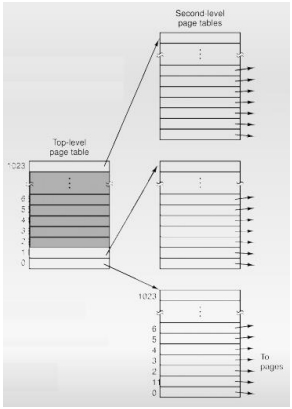
EFFECTIVE ACCCES TIME (EAT)

Lookup Associativo (LA): tempo di accesso alla memoria.
Hit Ratio (HR): percentuale di volte che ho un "page hit" (calcolato in base al numero dei registri totali).
TM: ritardo del canale (?).
Tempo di Accesso Effettivo (TAE): HR x (LA + tm) + (1 – HR) x (LA + 2tm)

TABELLA DELLE PAGINE MULTILIVELLO

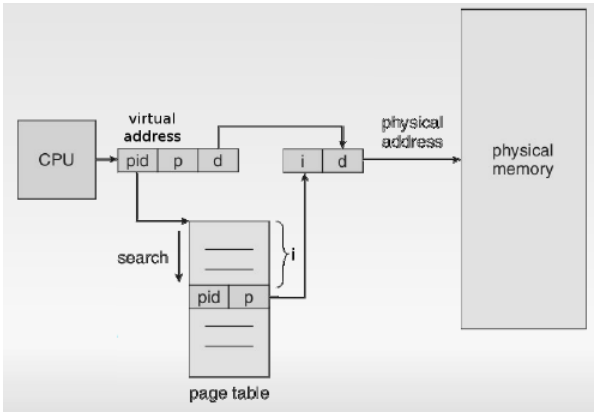


Questo tipo di tabella è utile in quanto consente di risparmiare spazio, le pagine inutili infatti non vengono allocate e quindi non appesantiscono il sistema.



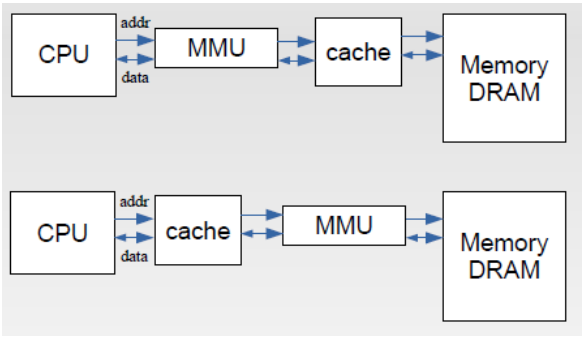
È possibile suddividere la tabella in gruppi omogenei trattati come figli di un nodo radice. La struttura è gestita tramite una tabella, che contiene dei puntatori alle pagine virtuali dei gruppi più bassi. È possibile che elementi della tabella non puntino a nulla (così da risparmiare memoria). Il numero di accessi alla memoria sono tanti, data l'elevata complessità della struttura dati: un accesso alla tabella di primo livello, una altro accesso per la tabella di secondo livello ed un ulteriore accesso al dato.

TABELLA DELLE PAGINE INVERTITA

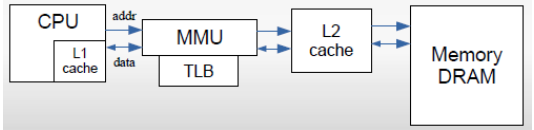


Esistono altri approcci che gestiscono gli spazi di indirizzamento a 64 bit. Una tabella delle pagine mi permette di mappare i numeri delle pagine virtuali ai numeri di frame, per fare ciò potremmo utilizzare una struttura diversa:
Immaginiamo una tabella con tante voci quanti sono i frame e non le pagine virtuali. Al suo interno deve contenere un bit di controllo (se il frame è allocato o meno), il numero di pagina virtuale e un identificativo univoco che distingua gli spazi di indirizzamento (PID). Questo è il numero minimo di bit di referenziamento che mi servono. Per supportare la ricerca è necessaria una struttura interna decente, altrimenti la ricerca non sarebbe efficiente, si utilizza quindi una Tabella Hash: una tabella più snella dove più voci della tabella corrispondono ad una sola, quindi vengono messe in lista. Ovviamnete è necessaria solo una tabella delle pagine invertita.

CACHE DELLA MEMORIA VS. MEMORIA VIRTUALE



Come deve comportarsi la Cache in presenza dell'MMU?
Se è messa dopo l'MMU l'informazione è già passata dall'MMU, e, in caso di cache-miss, passa il compito alla Memoria Centrale. In questo caso non sono obbligato ad azzerare la cache perché non ho pericolo di confusione. In questo modo però questa cache, prima di poter essere consultata, devo pagare il costo della traduzione dell'MMU da indirizzo virtuale a indirizzo fisico (possiamo vederla come un collo di bottiglia).
Se pongo la cache tra CPU e MMU, la cache viene consultata per indirizzo virtuale, nel caso di cache-miss passa il dato all'MMU. Il non azzerare questo tipo di cache è un problema, SIAMO OBBLIGATI AD AZZERARLA poiché in ogni linea di cache ho un identificativo indicativo, e nel caso di cambio di contesto, gli spazi di indirizzamento di un altro processo esistono, ma non contengono i dati giusti e ciò da vita a problemi. Per ovviare a ciò è necessario identificare ogni spazio non solo con l'ID, ma anche con un identificativo del processo (ASID). Questo tipo di cache sembra più indicata, ma in realtà questo schema scala male: al crescere delle dimensioni il meccanismo perde efficenza, quindi è utile utilizzarlo in cache di primo livello (L1) poiché sono molto piccole.
Cosa si usa in pratica?
Cache L1 basata su indirizzi virtuali, Cache L2 può essere di entrambi i tipi. La CPU potrebbe passare la richiesta sia alla cache L1 sia alla MMU (così da anticipare un possibile cache-miss)



ALGORITMI

ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Cosa accade nel caso di page fault? Bisogna trovare una pagina vittima in cui memorizzare la nuova pagina richiesta in modo da non generare ulteriori page fault.
È un porblema simile al cache-miss, ma questo è da gestire da lato SW (il cache-miss è gestito dall HW).
Bisogna scegliere la pagina meno interessante come pagina vittima in modo da minimizzare i futuri page-fault (e l'overhead), si sceglie una pagina poco utilizzata da ora nel futuro. Questa scelta non è implementabile in modo efficiente: bisognerebbe numerare le pagine indicando il numero di istruzioni che verranno eseguite prima di riferirsi nuovamente alla pagina, pagine con numeri più alti sarebbero potenziali pagine vittime.
Ora vediamo soluzioni reali e implementabili.

ALGORITMO NOT RECENTLY USED (NRU)

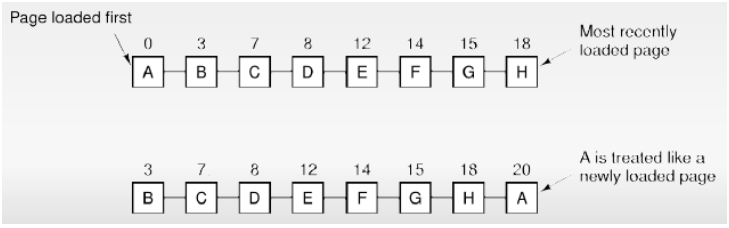
Raccoglie delle analisi statistiche riguardanti l'uso passato della pagina, vede quali pagine sono sacrificabili.
Usa il bit di Referenziamento (R) e il bit di Modifica (M) (questi bit sono azzerati periodicamente), suddivide le pagine in 4 classi:

- Classe 0: Gruppo di pagine che hanno R=0, M=0;
- Classe 1: Gruppo di pagine che hanno R=0, M=1;
- Classe 2: Gruppo di pagine che hanno R=1, M=0;
- Classe 3: Gruppo di pagine che hanno R=1, M=1;

Classi con indici più piccoli sono più sacrificabili, tra le classi posso scegliere con un FIFO quale sacrificare.

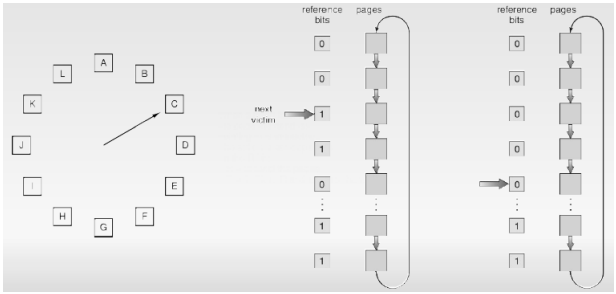
ALGORITMO FIRST IN FIRST OUT (FIFO) E DELLA SECONDA CHANCE

Si scarta la pagina più vecchia portata in memoria, ma materialmente questa pagina potrebbe ancora essere molto usata, quindi si implementa la seconda chance.
È un FIFO che tiene conto del bit di Referenziamento (R): se è impostato ad 1 la pagina in testa (che è etata recentemente referenziata) viene saltata, non scartata, inserita in coda e il suo bit (R) è reimpostato a 0.



ALGOITMO DI CLOCK

Simile all' algoritmo precedente, ma ho una coda circolare, invece di spostare gli elementi uso un puntatore e quindi avrò un risparmio di risorse.



ALGOITMO LEAST RECENTLY USED (LRU)

Guarda alle pagine come sono usate di recente, stila una classifica e sceglie la pagina che è stata usata meno di recente.
Mantiene un contatore associato alle pagine (potrebbe essere il numero di istruzioni eseguite dalla CPU dopo aver consultato la pagina), pagine con contatori maggiori saranno potenziali vittime. È un ottima approssimazione dell'algoritmo ottimale, ma è complicata e dispendiosa: ad ogni nuovo referenziamento le istruzioni devono essere aggiornate. Vediamo ora varie soluzioni, sia HW che SW.

- Con supporto HW:

  Potremmo avere un registro dedicato che viene incrementato ogni istruzione eseguita dalla CPU. Valori più piccoli saranno i più vecchi. Devo fare una sovrascrizione ad ogni accesso. È impraticabile.

  Posso usare una matrice di bit, unica per tutte le pagine e mantenuta all'interno dell'MMU. Ha N righe e N colonne, dove N è il numero di frame. Dovrei fare un aggiornamento della matrice ogni volta che effettuo un accesso alla pagina dell' Iesimo frame, se tto ad 1 tutti gli elementi dell'Iesima riga e poi a 0 tutti gli elementi dell'Iesima colonna. La scelta della pagina vittima cade sulla pagina la cui riga ha più cifre poste a zero. Rispetto alla soluzione precedente è più fattibile perché la struttura è unica.

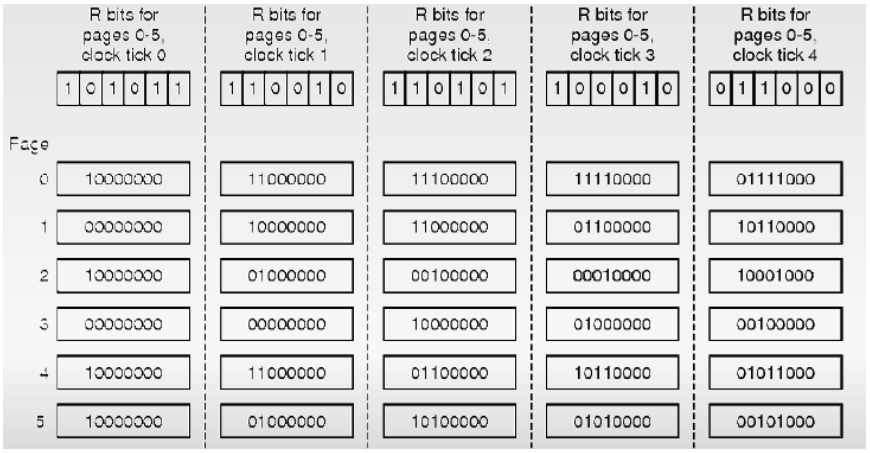
- Con supporto SW:

  ALGORITMO NOT FREQUENTLY USED (NFU)

  Uso un contatore associato ad ogni pagina, li aggiorno periodicamente in modo SW con frequenza bassa sfruttando il bit di referenziamento: prima di resettarlo periodicamente tramite la procedura, lo sommo al contatore della rispettiva pagina, cosicchè pagine che sono rilevate con il bit di referenziamento pari ad 1 avranno il contatore maggiore, quindi le pagine vittime saranno quelle col contatore minore. Il problema è che pagine molto usate nel passato e non usate nel presente avranno comunque contatori molto alti.

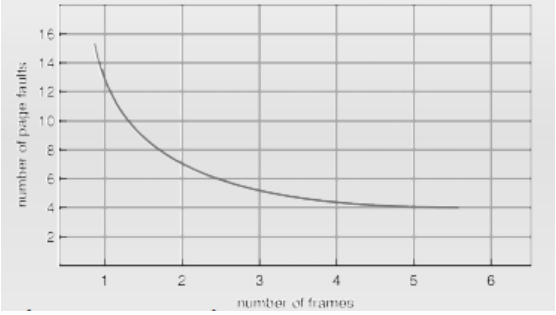
  ALGORITMO DI AGING

  Questo algoritmo prevede di effettuare uno shift a destra del contatore binario e portare il bit di referenziamento come bit più significativo e scartare quello meno significativo. La scelta della vittima cade sul contatore più piccolo.
  È una buona approssimazione, ma non è perfetta:
  Lo storico è limitato ad N cicli (N= numero di bit del contatore).
  Non ho idea del numero di referenziamenti nel ciclo, ma solo se in quel ciclo c'è stato almeno un referenziamento.

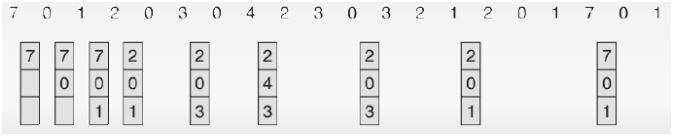


CONFRONTO DELLE PRESTAZIONI

Vedo quale degli algoritmi genera minori page-foult, per stabilire il miglior algoritmo. Meno frame disponibili avrò, maggiori saranno i page-fault.
Come caso base userò 3 frame in memoria e una sequenza precisa di 20 indirizzi virtuali senza doppioni consecutivi a cui farò accesso.

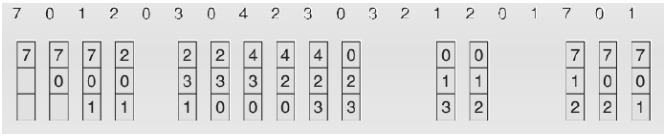


ALGORITMO OPT



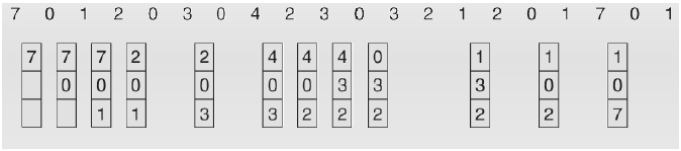
9 PF
Non posso fare di meglio

ALGORITMO FIFO



15 PF, molto peggiore.
L'algoritmo FIFO (e quelli riconducibili al FIFO) ha un anomalia detta anomalia di Bedamy, in cui all'aumentare dei frame i page foult POTREBBERO crescere

ALGORITMO LRU

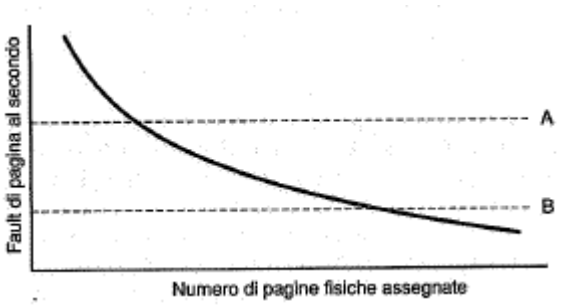


12 PF, migliore a FIFO.
L'LRU (e gli algoritmi che presentano la proprietà di inclusione) non soffrano dell'anomalia di Bedamy in quanto lo stesso set di pagine con N frame lo ritrovo in N+1 frame, ed in più avrò un frame libero.

ALTRI ALGORITMI
NFU, aging: godono della proprietà di inclusione (simile all'LRU).
Seconda chance, clock: soffrono dell'anomalia (riducono a FIFO).
NRU: soffre dell'anomalia (riduce a FIFO).

ALLCOAZIONE DEI FRAME

Vediamo ora che logica segue il SO per l'allocazione della memoria, e le strategie migliori.
La Paginazione su richiesta (pure demand paging) è una strategia che risolve il problema di capire quali pagine allocare in fase di avvio del processo: inizialmente non ne viene caricata nessuna: il sistema affida al meccanismo del page fault il compito di allocare le pagine a cui il processo si riferisce. In questo modo avremo un picco di page fault frequency all'inizio dell'allocazione della memoria, che poi si stabilizzerà, è una strategia lazy, ma efficace.



Ad un processo vengono assegnate dei frame:
c'è un Minimo numero di frame che devono essere assegnati al processo, che dipende dal processo stesso e dall'architettura, poiché l'esecuzione di un istruzione può interessare più pagine e se non ho abbastanza frame disponibili non posso eseguire l'istruzione. Le pagine generalmente sono:

- 1 per contenere l'istruzione.
- 1 per ogni operando.
- N per gli accessi indiretti alla memoria.

Nel caso il SO non possa offrire tutti i frame richiesti, si fa diventare il processo inattivo e la memoria a lui dedicata viene rilasciata.

C'è un Massimo numero di frame, che corrisponde alla memoria libera.
Esistono diversi modi per suddividere le allocazioni di memoria tra i processi:

- Allocazione equa: Ogni processo ha N frame su cui può allocare memoria, se genera un page fault il frame da riallocare viene pescato tra quelli assegnatigli. Ovviamente il SO si riserva più memoria rispetto ai processi.
- Allocazione proporzionale: Più realisticamente al processo viene allocata memoria in proporzione alla sua taglia, cioè al processo i di dimensione s_i vengono assegnati $a_i = s_i / S \times m$ frame (dove S è la dimensione dei vari processi e m è il numero di frame)
- Allocazione per priorità: Si favoriscono perocessi a priorità più elevata, così da rendere più efficiente l'esecuzione.

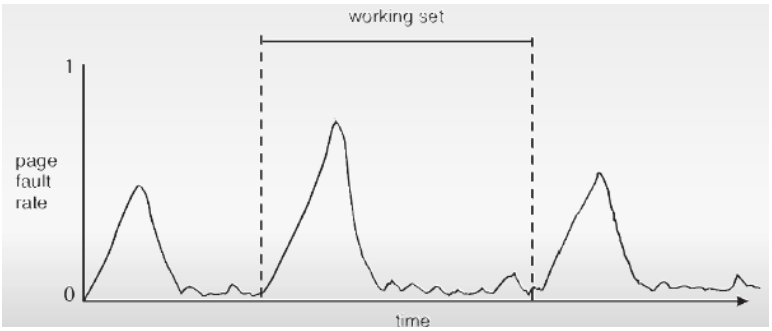
In realtà mitigando l'allocoazione proporzionale all' allocazione per piorità si ottiene un ottimo risultato di allocazione.
Per la scelta della pagina vittima posso scegliere o pagine locali (dello stesso processo, allocazione locale) o pagine appartenenti anche ad altri frame (allocazione globale). Nella stragrande maggioranza si usa l'allocazione globale, in questo modo il numero di page fault non dipenderà più dalla bontà dell'algoritmo o dalle pagine dedicate inizialmente, ma anche dalla probabilità che un altro processo "rubi" pagine fisiche.
Nel caso un processo disponga di pochi frame sopra il suo minimo indispensabile, entra in Trashing, cioè crea molti page fault (che hanno un alto costo in termini di overhead) e si rallenta moltissimo il comparto di I/O, conivolgendo anche gli altri processi. Per risolverlo è necessario dargli più allocazioni di memoria, ma se non ho abbastanza frame per soddisfare questa richiesta l'unica soluzione è fare uno swap-out o del processo in trashing oppure di un processo vittima che ha una più bassa priorità.
Il problema sorge quando ho più processi in trashing, cioè quando un eccessivo grado di multiprogrammazione porta la stragrande magioranza dei processi ad andare in trashing, generando un sovraccarico. A questo punto è necessario terminare qualche processo manualmente.
Per decidere quanta memoria allocare ad un processo ci si rifà al Principio di località: una porzione ben precisa di dati e di risorse che il processo necessita per essere eseguito correttamnete in un preciso momento. È convenienete che una località si caricata intermante in memoria, così da minimizzare i page fault.
Le località sono molte e possono essere condivise o cambiate: quando cambio località avrò un alto numero di page fault causati dal caricamento delle nuove pagine appartenenti alla località.
Quando è necessario allocare i frame ad un processo bisogna adeguare il loro numero alla località attuale, deve essere abbastanza grande per contenerli (meglio eccedere di un po in quanto la località non ha sempre dimnsione fissa).

WORKING SET (WS)

Un tentativo di approssimare il concetto di località è il Workig Set (WS) che corrisponde, in un dato istante, ad un insieme di pagine ben preciso. Fissato un parametro Λ, questo indica le pagine usate negli ultimi X accessi alla memoria. Se scelgo opportunamente Λ, il Working Set contiene il concetto di località del processo in maniera più adeguata.
Se riesco a calcolare la dimensione dell' I-esimo working set (Working Set Size$_i$ (WSS_i)) di tutti i processi, rieso a calcolare la richiesta globale di frame, che posso comparare con la memoria disponibile, rendendomi più facile allocare più frame o prevenire il trashing abbassando il grado di multiprogrammazione (facendo lo swap-out di qualche processo).
Posso usare quindi la tecnica della prepaginazione, che consiste nel riportare in memoria il WS prima dello swap out, per poi essere riprisitnato più rapidamente, questo abbassa il numero di page fault. Un tipo simile di prepaginazione può avvenire all'avvio del processo, si raccolgono delle statistiche delle pagine (relative alle risorse e ai dati più che altro) tipicamente utilizzate dall'applicazione in fase di avvio, così da velocizzarne l'inizzializzazione.
Il working set si può calcolare tramite:

- Interrupt periodici
- Bit di referenziamento R
- Log che conserva la storia del processo in base alle istruzioni effettuate nel tempo di Λ

PAGE FAULT FREQUENCY



Quando il WS è totalmente in memoria avrò poche page fault, mentre se non è totalmente in memoria ne riscontrerò tanti. Quindi posso mettere in relazione il WS con il Page Fault Frequency (PFF): processi con bassa PFF (Lower Bound) possono cedere frame a processi con PFF più alta (Upper Bound), così da adeguare la quantità di frame alla dimensione del working set, e di fatto approssima esattamente ciò di cui abbiamo bisogno. Ovviamente se ci sono troppi processi con PFF troppo alta questo modello non può fare miracoli. Tramite la PFF posso individuare i working set del processo.

POLITICA DI PULITURA

Quando avviene un page fault, il caso migliore è trovare immediatamente un frame libero dove assegnare il nostro processo. Per far in modo di trovare questo caso spesso, si usa la strategia della Pulitura per mantenere sempre un numero minimo di frame liberi in memoria (devo far in modo di non dover ricorrere al cercare una pagina sporca da usare). In realtà sto anticipando il lavoro di liberazione di memoria di una pagina vittima (se è spoca si deve pulire prima sincronizzandola e poi marcandola come frame libero). Perché farlo? Perché riduce i tempi di reazione al page fault e posso applicarla tramite un Paging Daemon che entra in funzione nei momenti di I/O, cioè di scarso uso della CPU. Alcuni SO effettuano solo la sincronizzazione e non la liberazione del frame, anticipando del lavoro, ma rendendo meno efficiente la pulizia. Nel caso la pagina appena liberata sia oggetto di un page fault, può essere ripescata e ritorna subito disponibile senza necessità di I/O dato che materialmente i dati all' interno della frame non sono stati cancellati.

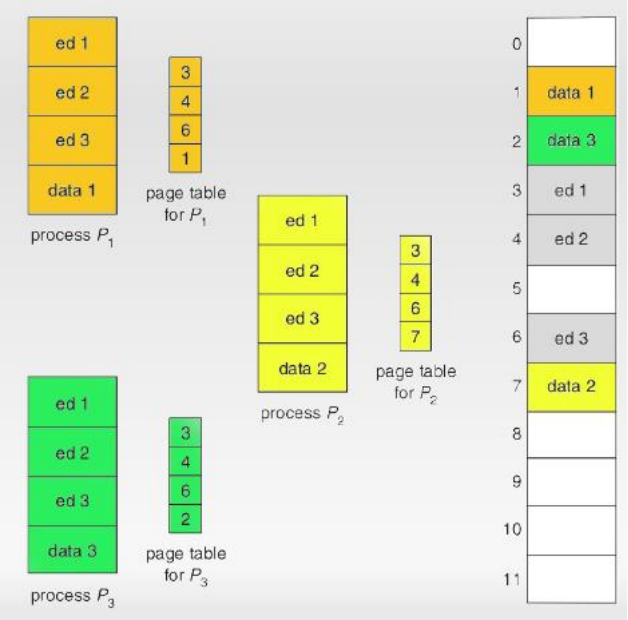
DIMENSIONE DELLA PAGINA

Come si deve scegliere la dimensione della pagina e del frame? Questa è molto importante ed ha delle ripercussioni, può essere fissa o variabile (di pochi valori) ed è stabilita come taglia dai proggettisti dell'HW. Comunemente può essere scelto dal SO in un range tra 512Byte - 64Kbyte a oltre (sempre a potenze di due ovviamente). Quali sono i vantaggi?

- Pagine Grandi: La tabella delle pagine è più piccola: più sono grandi i frame, più memoria occupa ognuno di questi, meno elementi in tabella troverò;
  Migliora il modo in cui il sistema utilizza l'I/O: se lavoro con pagine di 1Kb e devo leggere questa pagina dal disco: le tempistiche dipendono dal sick time (tempo necessario per posizionare la testina nell'area di interesse, circa 20ms), dal latency time (tempo di rotazione del disco 6-8 ms) e dal transfer time (tempo di trasferimento dei dati 0.1ms per blocchi da 1K), l'unica in relazione con la dimensione della pagina è il transfer time: grandezze più grandi implicano trasferimenti più grandi. Pagine grandi fanno in modo che sick e latency time siano pagate meno volte per il trasferimento degli stessi dati rispetto a pagine più piccole;
  Al crescere della dimensione si riducono i page fault: questo perché se ho variazioni nella grandezza dei dati contenuti nella frame non ho problemi nel dover allocare più frame e quindi generare meno page fault.
- Pagine Piccole: C'è minore frammentazione interna: se ho bisogno di più memoria rispetto a quella contenuta in un frame ho bisogno di allocarne un altro, se alloco piccoli frame posso arrotondare lo spreco al minimo, se uso frame più grossi avrò sprechi più importanti;
  Migliore risoluzione nel definire il working set: spreco meno memoria dato che uso frame più piccoli, agevolo così la multiprogrammazione.

PAGINE CONDIVISE

Vediamo come viene implementata la posibilità di andare a violare il modello di isolamento dei processi, cioè di condividere delle pagine, queste possono essere condivise:



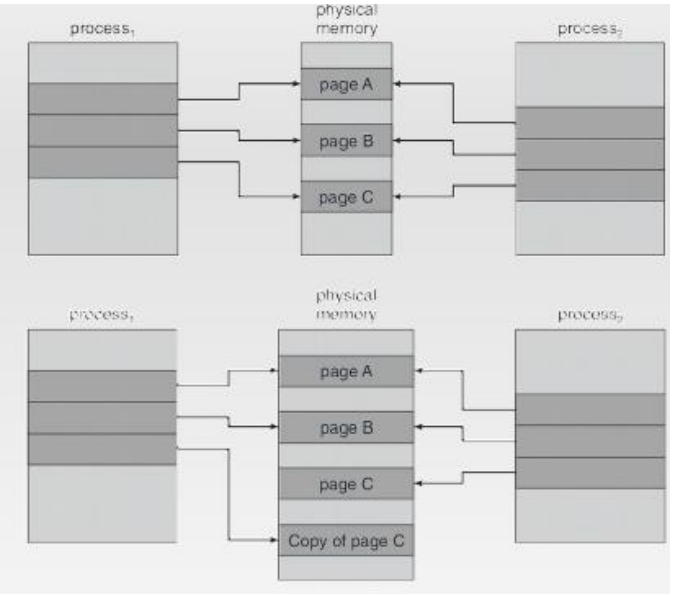
- Sola lettura: Nel caso si istanzino due o più processi uguali, questi avranno in comune il codice (ed X), quindi è inutlile istanziarlo più volte (codice rientrante). Questa soluzione è perfetta quando nessuno dei due processi ha necessità di effettuare scrittura.

  Nel caso dell' esempio le tabelle delle pagine dei processi P_1 e P_3 hanno gli stessi indirizzi nelle aree del codice(ed1, ed2, ed3), mentre puntano a pagine differenti nel caso dei dati (data1, data3). Questo "riutillizzo" di pagine consente un grande risparmio di memoria nei sistemi multiutente.
- Lettura/scrittura: Questo rapprensenta una violazione del modello di isolamento dei processi, ma consente una comunicazione. Quando un processo tenta una scrittura sui dati, la violazione della protezione Read Only causa una trap al SO, viene quindi fatta una copia della pagina cosicchè ogni processo abbia la sua copia privata, quindi entrambe le copie sono impostate in Read-Write, così da non dare problemi in successive operazioni di scrittura. Faciendo ciò non duplico tutte le pagine comuni, ma solo quelle che vengono man mano modificate.
  Si può pensare anche ad un implementazione che non necessita di duplicazioni, dove i due processi condividono i dati, anche modificandoli. L'idea è che i processi possano interagire tramite segmenti di memoria condivisa, per scambiarsi dati. Nei due processi avrò due pagine che si riferiscono all'area di memoria condivisa, entrambi hanno i diritti di scrittura e lettura (race condition), se scrivo qualcosa nella pagina del processo, si aggiornerà anche l'area di memoria.

Quali sono le complicazioni?

- Gestione della cache: Quando due processi condividono un area di memoria, questi comunque saranno due pagine diverse ed entrambe le pagine dovranno essere salvate nella cache: avrò uno spreco di memoria, ma soprattutto se entrambi i processi stanno lavorando, se P1 modifica qualcosa e nello stesso momento P2 deve leggere, P2 leggerà ciò che c'è nella sua area della cache, senza vedere le modifiche di P1. La soluzione è passare prima dalla MMU che mi dà l'indirizzo fisico e vedere la entry che sto per caricare ha lo stesso tag fisico di un'altra entry già in memoria, se ciò accade non salverò la seconda entry ma lavorerò sulla prima.
- Tabella delle pagine invertita: è una tabella globale che prevede tante entry quanti sono i frame e ne contiene delle informazioni. Nel caso di due processi che condividono memoria (P1 e P2) hanno sicuramente ASID diverso (uno è P1 e l'altro è P2) e anche indirizzo virtuale diverso, ma entrambe dovrebbero riferirsi alla medesima memoria, ciò che non può succedere dato che la tabella delle pagine invertita si riferisce al ASID e indirizzo virtuale. Per risolvere il problema basta modificare le voci della entry cosicchè se P1 sta lavornado si riferiscono a questo, ma quando si effettua un context switch si modificano in modo che si rifersicano a P2. È una toppa ma una buona soluzione. Questo approccio è impraticabile nei sistemi multicore: se P1 e P2 lavorano contemporaneamente sulla tabella da due core diversi, si avrebbero troppi context switch. Teoricamente sarebbe possibile risolvere il problema con tabelle con corrispondenze moti-a-uno, mai realizzate in realtà.

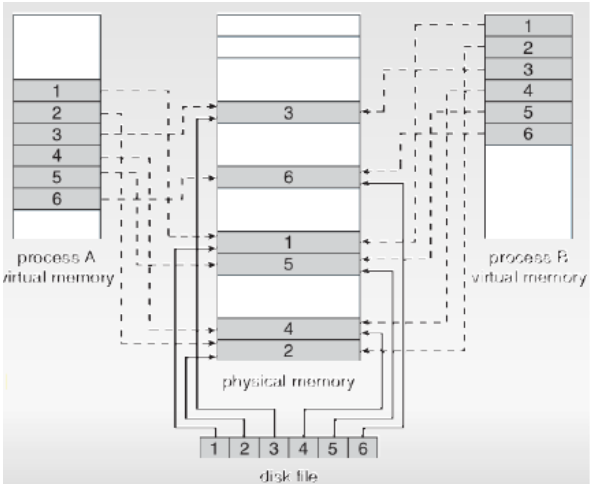
COPY-ON-WRITE E ZERO-FILL-ON-DEMAND



Cosa succede quando effettuo una fork? Clono un processo (a meno dell'ID) e ne copio dati, codice, indirizzi, ecc… entrambi quindi condivideranno i frame nella memoria centrale, il problema sorge quando effettuerò una scrittura, altererò una word e quindi il sistema di condivisione del mendesimo frame non funzionerà più, bisognerà trovare un nuovo frame, allocarlo e mapparlo per far in modo che un processo non modifichi i dati dell' altro.

- Copy on Write: I processi hanno diritti in sola lettura, quando cercano di scrivere si attiverà il meccanismo di copy-on-write che allocherà un nuovo frame a cui arà puntare il processo e ne garantirà il diritto di scrittura.
- Zero fill on demand: Quando un processo necessita di allocare nuova memoria è necessario allocarla al processo e mapparla su un frame che deve essere azzerato. Le pagine sono riferite ad un frame vuoto fisso ed in modalità lettura, appena qualcuna cerca di scrivere, le viene allocata una pagina dedicata solo a lei (come succede nel Copy on Write). Si può creare un pool di pagine libere e vuote pronte per essere allocate: questo permette di risparmiare risorse (si libera nei momenti di I/O)

LIBRERIE CONDIVISE E FILE MAPPATI



Le librerie sono una seire di procedure che si usano per risolvere problemi comuni, unificano e condividono il codice tra più applicazioni.

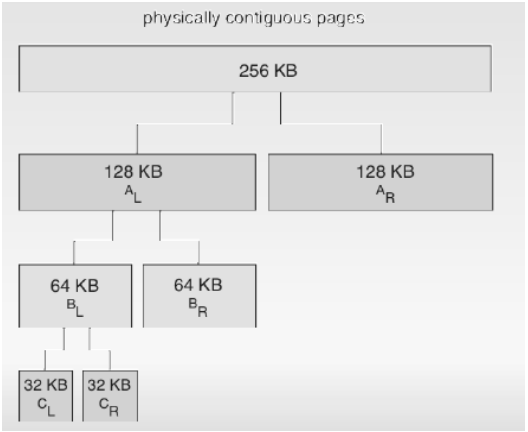
- Linking statico: In fase di compilazione includo il codice della libreria nel codice sorgente, non ha bisogno di elementi esterni e di caricare parti di librerie in corso di esecuzione. (poco usato).
- Linking diammico: Quando l'appilcazione ha bisogno di una certa funzionalità, la libreria viene caricata in memoria. Ciò che avveniva in fase di compilazione ora avviene in runtime. Ciò si traduce in risparmio in memoria nel caso altre applicazioni usino medesime librerie dato che ne tengo in memoria solo una copia. È possibile aggiornare la libreria, in questo caso è necessario specificare la versione della libreria a cui volgiamo accedere.

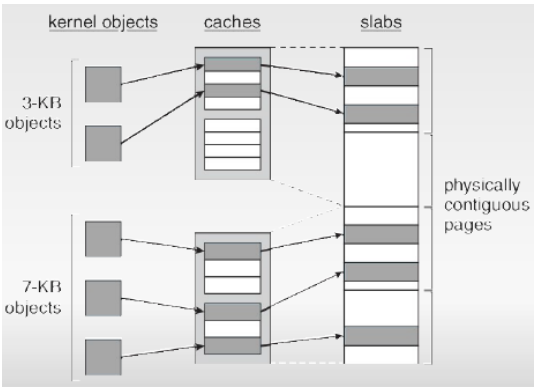
Mappatura dei file in memoria: è un modo diverso di utilizzare un file: il SO offre una chiamata di sistema che mappa un file che si trova sul disco in un area di indirizzamento (puntano al file).

ALLOCAZIONE DELLA MEMORIA PER IL KERNEL

Quando il SO usa delle strutture dati deve allocare memoria, è necessario quindi adottare un sistema che sia veloce, stabile, senza sprechi o frammentazione.

- Buddy system: Supparta le richieste di allocazione del SO limitando il meno possibile la frammentazione interna. L'idea è quello di usare segmenti di memoria (potenza di 2) che è possibile dividere a metà: ogni richiesta viene arrotondata alla potenza di 2 più vicina e viene allocato il segmanto di memoria disponibile uguale all'arrotondamento.

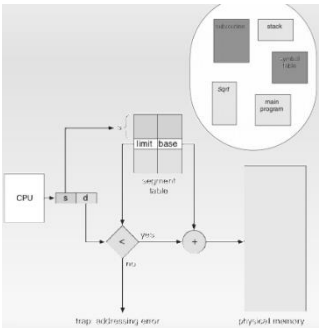




-Slab allocator: Ho una cache per ogni struttura dati, che sia libera o occupata. Posso suddividere le richieste in "taglie". Ogni cache è una collezione di slab.
Uno slab è un pezzo di memoria dedicato a contenere strutture dati di una certa grandezza, è contiguo con altri slab, deve essere un multiplo della dimensione di base della struttura dati e del frame (struttura=3K, frame=4K -> slab=12K)
All'inizio uno slab è vuoto, appena è necessario si alloca il primo record libero e lo si alloca. Se si riempie uno slab è possibile ricorrere a slab che appartengono alla stessa cache. L'allocazione e deallocazione è rapida, non c'è frammentazione interna ne esterna.

SEGMENTAZIONE

La segmentazione cerca di far coincidere il modello fisico con quello logico: la memoria è una collezione di segmenti che contengono oggetti di dimensione variabile. Per identificare avremo l'ID dell' oggetto e l'offset, cioè la sua posizione (#Segmento, Offset). Si usano dei registri nella CPU per i segmenti per generare la coppia di indirizzi (Base, Limite) (la base coincide con l'Offset, il Limite indica l'ultimo bit di memoria del segmento. Ad ogni segmento vengono associati questi valori e sono salvati nella Tabella dei Segmenti. L'HW suporta sia paginazione che segmentazione, è il SW che decide quale delle due sfruttare.



SEGMENTAZIONE SU PERNTIUM INTEL (32bit)

Il ruolo dell'MMU è coperto da 2 altre unutà: una segmentation unit che va in pipeline con una Paging unit: la prima mappa gli indirizzi logici in indirizzi lineari che sono allocati in modo contiguo, la seconda alloca in maniera discontigua gli indirizzi lineari.

Esistono 16000 segmenti (tra riservati e non) di dimensione massima di 4GB, questi non arrivano mai alla loro capienza massima e l'HW non si appura del fatto che non ci siano intersezioni. Questi segmenti sono divisi in due:

- 8000 utilizzati in modalità utente dai programmi registrati in una Local Descriptor Table (LDT), unica per ogni processo
- 8000 utilizzati dal SO contenuta in una Global Descriptor Table (GDT), unica per il SO.

Ogni segmento nell'architettura Intel ha un livello di protezione stabilito, da 0 a 3 (quattro in tutto)
Ora il nostro indirizzo viene spezzettato in pagine e dislocato in memoria fisica come pagine tra i 4Kb e 4Mb.

I primi 10 bit sono detti Dir, che stabilisce la posizione nella Page Dierctory del frame, nel frame avrò il punto di partenza per calcolare la Page dalla Page Table, e da li infine trovre la Word nella Page Frame tramite l'Off-set. Questi bit ci aiutano a trovare la pagina, sia essa in memoria fisica o sul disco.
Questo schema fa uso della TLB, nell'architettura a 64bit ci sono 4 livelli e la sua segmentazione è resa meno importante.

GESTIONE DELA MEMORIA SU LINUX

In linux abbiamo solo 6 segmenti (contro i 16000 della tecnologia Intel):
2 per l'utente, 2 per il kernel e 2 per la struttura.
Gli indirizzi gestibili sono a 48 bit, quando alllochiamo spazio per un processo non teniamo in considerazione l'heap

FILE-SYSTEM & DISCHI

I FILE

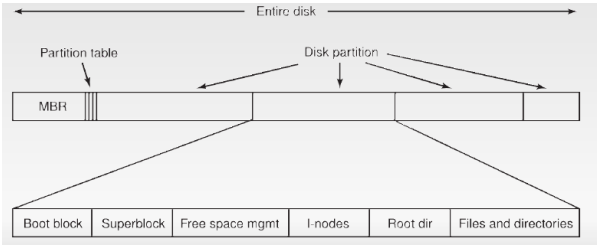
Il tutto è gestito dal SO, contiene grandi quantità di informazioni in maniera persistente e condivisa.
Il FILE è l'unità più piccola che possiamo trovare, ma è da considerarsi come un astrazione: l'utente non si interessa di come o dove il FILE sia memorizzato o come funzionano realemtne i dischi. Un aggegato di più FILE su disco è una directory.
Per identificare i vari FILE è necessario nominarli: a questo scopo è possbile identificarli con una stringa di lunghezza variabile ed è possibile usare caratteri più o meno speciali in base al SO. I FILE possono avere il nome diviso in 2 parti: la prima è il nome effettivo, la seconda è l'estensione: in WINDOWS indica il tipo di FILE , in UNIX l'estensione è solo una convenzione e non ha validità per il SO.
Il SO tratta i FILE come sequenze di byte, non è interessato dal contenuto, sarà l'applicazione a decodificare: questo è molto importante perché mantiene il sistema molto flessibile.
I FILE possono avere o una codifica binaria o ASCII, il secondo caso è preferibile perché processi diversi possono scambiarsi input ed output in ASCII tranquillamente e senza problemi di compatibilità.
Ogni FILE è composto da 5 parti:

- INTESTAZIONE:
  - Magic Number (numero magico): identifica il FILE come eseguibile o meno, indica il formanto del FILE.
  - Vari interi: danno la dimensione delle varie parti del file, l'indirizzo da cui parte l'esecuzione e dei flag.
- TESTO
- DATI
- BIT DI RILOCAZIONE
- TAELLA DEI SIMBOLI

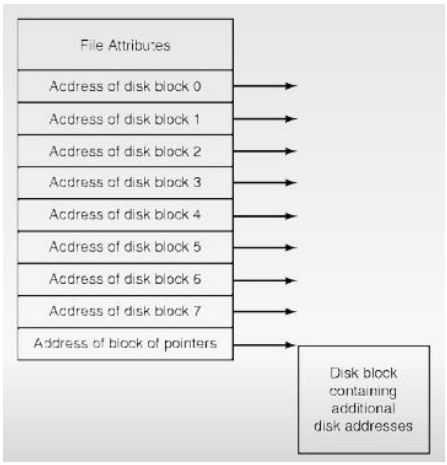
I SO supportano varie modalità di accesso ai dati:

- Lock Esclusivo: Il FILE viene bloccato quando qualcuno lo sta usando, mentre è bloccato nessun altro può accedervi.
- Lock Condiviso: Il FILE non viene bloccato e piò avere più lettori e più scrittori contemporanei.
- Lock Mandatory: Il SO gestisce il lock.
- Lock Mandatory: Il SO deve autoimplementarsi.

STRUTTURA DEL FILE SYSTEM



Il File-System è l'insieme dei dettagli tecnici, di gestione e di implementazione presenti sul disco.
Alcuni SO accedono l file attraverso delle coordinate, tre numeri che indicano la posizione del blocco sul dico. Si usa il Logical Block Addressing (LBA): Ogni sezione del disco è identificata tramite questo sistema di indirizzamento (LBA 0 ,LBA 1, LBA 2, LBA 3, ecc…).
Il Disco può essere partizionato per organizzare meglio i dati o per contenere SO diversi, le varie partizioniu possono essere formattate diveramente.
Il primo record del file system contiene una tabella delle partizioni (Master Boot Record (MBR)) che identifica quante parizioni ci sono e la loro disposizione.
Come alloca i FILE il SO?
I FILE possono essere allocati in maniera Contigua: risultano più facili e semplici da leggere.
Questo tipo di allocazione permentte di individuare un FILE con sole due informazini: l'indirizzo di partenza e le sue dimensioni, inoltre è molto efficiente e semplice. Dopo un certo tempo di uso di questo tipo di allocazione si riscontrerà della frammentazione interna del disco, dovuta alla continua cancellazione e riallocazione di FILE.
Nel caso di un allocazione non Contrigua (Liste Concatenate) incontro diversi problemi, tra i quali il fatto che devo tenere in memoria dei puntatori alle altre parti del file e la testina deve spostarsi ogni volta. Risolve il problema della frammnetazione in quanto posso mettere ogni blocco in aree di memoria libere adatte a lui.

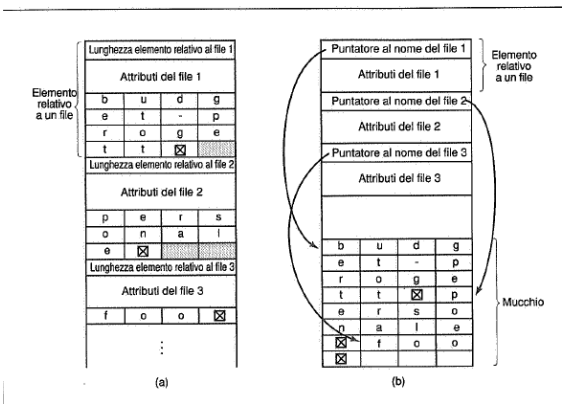


L'ultimo tipo di allocazione è l'uso degli i-node: è una piccola struttura dati che si riferisce ad un FILE e tiene tracca di dove siano allocate le sue varie parti in memoria, occupa meno memoria rispetto ai puntatori dell'allocazinoe non contigua e, dato che è è piccolo, posso scorrerlo velocemente. Ogni i-node ha una lunghezza prefissata, ma un FILE potrebbe eccedere e sforare la dimensione che un i-node può coprire, in questo caso l'unltimo campo dell'i-node ricondurrà non ad una parte del FILE, ma ad un altro i-node dove sono conservati i restanti indirizzi di allocazione del FILE.

IMPLEMENTAZIONE DELLE DIRECTORY

Una directory è un contenitore di altri oggetti, è un file costituito da sottostrutture capaci di "archiviare" i dati. La voce stessa della directory contiene riferimenti al file (data creazione, ultima modifica, dimensione, ecc…). In generale le directory hanno un informazione riguardannte il nome, una riguardante la posizione del file e infine i vari metadata.

Le informazioni relative al file possono essere salvate direttamente nella sua directory (Staticamente): questa si compone di una lista di elementi di dimensione fissa, uno per file, che contengono gli attributi, gli indirizzi sul disco dei blocchi del file e il suo nome che è variabile e si effettua il padding (riempie l'ultima linea se questa non è del tutto piena.
Nel caso si utilizzino gli i-node si risparmierà spazio e la directory memorizzerà il nome del file e il suo i-node.



C'è da dire che i nomi dei file non sono di lunghezza fissa (nei moderni SO il limite è di 255 caratteri) quindi non è ottimale un metodo che riservi uno spazio per così tanti caratteri che non verrà mai usato, per questo si è pensato bene di rendere dinamica la situazione: ci sarà un puntatore al nome del file invece che il nome stesso che perscherà il nome in uno spazio apposito (heap) dove sono allocati tutti i nomi di tutti i file, facendo così avrò uno spreco minimo.
L'HEAP è una struttura lineare, quindi le ricerche per molti elementi sono lente, è possibile anche dedicare una parte della cache in cui salvare le ricerche effettuate così da rispondere con prontezza alle richieste.

CONDIVISIONE DI UN FILE SU UN FILE SYSTEM

Quando il file system cerca di aprire un collegamento, effettivamente viene reindirizzato al file originario ed opera su quello (soft-link).
Limitatamente ad alcuni file system e in base all'allocazione dei file, posso creare dei riferimenti secondari che sono identici a quelli originali (i-node, hard-link)

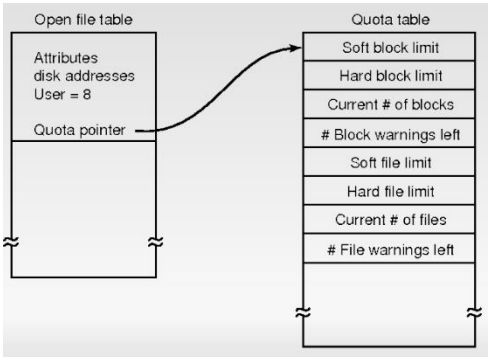
FILE SYSTEM VIRTUALI

Ad oggi i sistemi possono ospitare più file system(FAT, NTFS, ecc…), anche di sistemi operativi diversi. Ogni singolo SO può implementare i dettagli implementativi dei file system supportati. Ovviamente è un progetto enorme e difficile implementare tutto in modo tale che tutta la diversità non crei problemi all'utente, anche facendo in modo che questo non si accorga di nulla.
Linux usa il Virtual File System (VFS), tutto viene visto ad oggetti. Usa l'interfaccia POSIX (tutti i comendi di shell) per accedere alla VFS che riesce a comunicare con i vari file system. Il SO resce a nascondere questa differenza attraverso la modularizzazione degli oggetti facendo in modo che ogni oggetto si occupi dei dettagli implementativi in base al file system che si sta usando.
Si usano i V-node (un estensione degli i-node), quando si apre un FILE il SO, nella tabella dei file aperti, crea un riferimento V-node a quel file.

GESTIONE BLOCCHI LIBERI

È possibile segnare quale memoria sia libera tramite le bitmap (0 libera, 1 piena). È possibile far in modo che blocci appartenenti allo stesso file stiano vicini per aumentare la velocità di ricerca e lettura. La bitmap è gestibile solo se non troppo estesa.

QUOTE SUL DISCO



Per evitare che un utente occupi troppo spazio sul disco, il SO utilizza il metodo delle Quote: ogni utente ha una quota, un limite di file e blocchi a lui dedicati che non può superare.
Ogni volta che un utente apre un file, il SO localizza e sistema i suoi attributi e i suoi indirizzi del disco in una tabella dei file aperti che è nella memoria centrale, tra gli attributi c'è una voce che indica il proprietario del FILE, qualunque variazione della dimensione sarà attribuito a questo. Un altra tabella ha registrati gli utenti e le loro quote usate in quel momento dai file aperti.
Ogni volta che si aggiunge una voce alla tabella dei file aperti viene fatta puntare al record con la quota del proprietario: un aumento della dimensione del file è priontamente aggiornata nel conto delle quote totali e si fa un controllo rispetto a due limiti, uno hard (rigido) e uno soft (variabile) (80% dell'hard): passato il limite soft l'utente viene avvistato che la sua quota sta per essere raggiunta, se l'utente supera il limite hard riceverà un messaggio di errore e verrà bloccato. Ogni volta che viene superato un limite soft l'utente riceve un avvertimento e il contatore dei "Warnings" viene decrementato, se al termine della sua sessione l'utnete non rientra sotto il soft il nuovo valore degli avvertimenti sarà salvato permanentemente, se invece l'utente riesce a rimuovere gli eccessi prima di chiudere il collegameneto il nuovo valore dei warnings non viene salvato. Ragiunto il valore 0 nel campo Warnings viene tolta la possibilità all'utente di collegarsi e dovrà contattare l'amministratore di sistema per risolvere il blocco.

CONTROLLI DI CONSISTENZA

Quando il SO si avvia, la prima cosa che fa è settare un flag ad 1, mentre all'arresto è settare lo strsso flag a 0: se quindi in fase di esecuzione il SO crasha all'avvio troverà il flag settato ad 1 e dovrà effettuare un controllo sulla consistenza dei dati. Per controllare la consistenza dei blocchi usa due tabelle: una che indica i blocchi allocati ed una che indica i blocchi non allocati; tramite un controllo incrociato posso vedere se ci sono errori:

- Nel caso entrambi abbiano lo stesso blocco settato ad 1 il SO preferirà settare a 0 il campo dei blocchi liberi.
- Nel caso entrambi abbiano lo stesso blocco settato ad 0 il SO preferirà settare a 1 il campo dei blocchi liberi.
- Nel caso un blocco nella tabella dei free sia settato a 2 il SO lo setta ad 1.

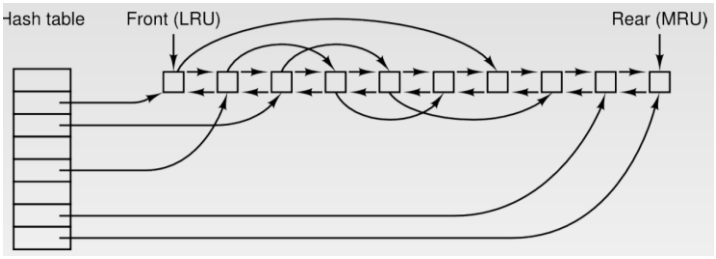
Per controllare la consistenza degli i-node il SO fa un controllo incrociato: partendo dalla radice conta quanti file con lo stesso nodo esistono (solo gli hard link) e compara il risultato con il numero di file puntato dall' i-node:

- Se l'i-node ha un numero maggiore di puntatori devo trovare il puntatore a NULL ed eliminarlo.
- Se il numero di file è superiore a quello dei puntatori dell'i-node basterà aggiungere un puntatore al file mancante.

FILE SYSTEM E I LOG

Nelle moderne macchine abbiamo una grande capacità di memorizzazione, ma i tempi di accesso al disco (lettura/scrittura) sono limitati. Per evitare di eseguire operazioni puntuali si cerca di "accorparle" ed eseguirle tutte insieme: i file aperti in scrittura e modificati vengono conservati nella RAM finchè non si raggiunge una dimensione prestabilita in modo da eseguire una scrittura sul disco di una quantità maggiore di dati in una sola volta. Ogni volta che devo effettuare la scrittura controllo se il blocco è stato realmente modificato o meno per risparmiare tempo.
Nei moderni SO si una il Journiling: ho una lista di task da eseguire (i metadati dei file) in ordine e, una volta eseguiti, li elimino dalla lista. Nel caso di crash posso riprendere la lista dall' ultima operazione effettuata. Il journiling funziona solo per operazioni idempotenti, cioè che se sono eseguite più volte di seguito non creano problemi.

CACHE DEL DISCO



La prima tra le tecniche per migliorare l'efficienza del disco è la cache del disco (buffer cache). È una memoria veloce, volatile che si frappone tra il file system e qualsiasi cosa cerchi di accedere alle informazioni che contiene (operazioni di I/O sul disco). La struttura è basata su una tabella hash che effettua una ricerca sulla presenza o meno dei file in cache. I moderni SO usano una parte della memoria volatile a loro disposizione (RAM) per fare caching su disco in modo da velocizzare le richieste d'accesso. Generalmente i SO usano a tale scopo la porzione di memoria inutilizzata in quel momento dai processi (che comunque sono i principali utilizzatori della RAM). In particolari situazioni il caching sel disco può occupare anche il 50% della memoria.
Il buffer cache è costituito da "pagine di memoria": quando un processo cerca di leggere o referenziare un elemento del disco prima passa dalla cache, se è presente lo prende da li altrimenti la richiesta viene reindirizzata al disco (cache miss). È necessario quindi, ad un certo punto, sacrificare blocchi vecchi per aggiungerne di nuovi, si usa la strategia dell'LRU un po modificata. Non cercheremo di approssimarlo, ma lo utilizzeremo appieno.
Ogni pagina di memoria è messa in una graduatoria, quando viene usata sale in testa alla coda, l'ultimo elemento è quello sacrificabile. È possibile farlo qua perché gli aggiornamenti sano rapidi rispetto al resto della situazione in cui si trovano.
Queste cache possono essere utilizzate in scrittura in due modi:

- Sincrona: La scrittura viene effettuata sul disco stesso, senza tener conto che il contenuto sia o meno in cache. Dal punto di vista delle prestazioni non è buona, in quanto, anche in lassi di tempo piccoli, è possibile che un blocco subisca numerose piccole scritture e accedervi ogni volta direttamente sul disco è molto costoso.
- Ascincrona: La cache è usata anche in modalità di scrittura, ogni aggiornamento è fatto sulla cache così da evitare i costi elevati di numerose piccole scritture, accumulare più scritture per rendere più efficiente la scrittura sul disco dal punto di vista dello spreco delle risosre elettro-meccanico. Comunque il SO garantisce la scrittura dei blocchi sporchi in circa 30sec.

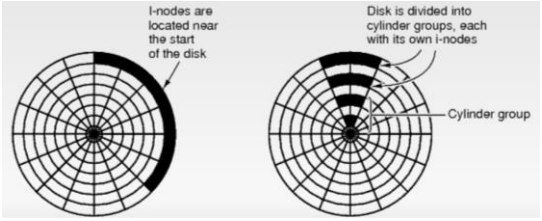
Il problema del caching è il fatto che se il sistema si arresta bruscamente è possibile che i dati vengano persi: i metadata sono blocchi molto importanti risptto agli altri, quindi è necessario salvaguardarli e si cerca di scriverli in maniera Sincrona.

Usando la tecnica read-ahead la testina leggerà non solo il blocco richiesto, ma anche quelli a lui successivi, questo è buono perché in termini di prestazioni non costa nulla e, a volte, i blocchi successivi saranno proprio quelli che servono in futuro.

La tecnica free-behind fa in modo che teoricamente i blocchi precedenti a quello letto non servono più, quindi i loro riferimenti nella cache del disco possono essere spostati negli ultimi posti, pronti per essere sacrtificati.

Quando mappiamo un file riserviamo un certo numero di frame nella memoria centrale per mantenere in memoria una copia di parti di questo. Nel caso il sistema utilizzi sia il page caching che il buffer cache il SO le tiene separate, ma in questo caso se un processo richiede un file che non è presente nella page cache ma è presente nella buffer cache, il trasporto e conversione è costoso, è costosa anche la doppia copia e la sincronizzazione.
I SO uniscono un unico buffercache da cui si attinge in entrambi i casi, in questo caso il frame e il blocco di memoria non differiscono per nulla, non ha i problemi precedenti.

DESIGN EL FILE SYSTEM E MIGLIORAMENTO DELLE PRESTAZIONI



Ogni lettura di un oggetto comporta letture di aree distinte del disco, il costo è totalmete dovuto agli spostamenti della testina.
Per ottimizzare al meglio la lettura/scrittura è possibile:

- Collocare gli i-node all'inizio del cilindro: appena devo aprire un file leggo l'i-node all inizio e poi mi sposto nel cilindro del blocco dei dati. In media impiegherò n/2.
- Collocare gli i-node alla metà del cilindro: stesso casi di prima, impiegherò n/4.
- Raggruppo gli i-node e creo dei gruppi di blocchi vicini agli i-node a cui riferiscono. Limita la frammentazione del file

FILE SYSTEM MS-DOS (FAT)

Nato come MS-DOS si è evoluto nell famiglia FAT, è tuttora utilizzato nei sistemi windows ed è consigliato sulle memorie flash, macchine fotografiche, sistemi embemmed, smartphone, ecc… questo per la sua compatibilità e fruibilità data la diffusione.
Noto il numero di blocco iniziale, risco a seguire la catena per identificare qualunque blocco appartenente al file.

- FAT-12→ 12 è la capienza del numero di blocco, influiva sulla dimensione del blocco che poteva supportare
- FAT-16→Utilizza blocchi più grandi, la singola voce è 16 bit
- FAT-32→Penultima versione, in cui le voci fisiche sfruttavano 28 bit su 32, il limite è 2TB, questo perché le dimensioni della partizione dei master boot record non possono descrivere partizioni superiori ai 2TB.
- FAT-64, exFAT→ Supprotata dalle ultime verisoni di Windows, è un ultieriore evoluzione della FAT, mette pezze su tanti aspetti. È protetta da brevetti, quindi non è utilizzabile su Linux. È sconsigliato per file system interni, usato per memorie flash e HDesterni.

FILE SYSTEM NTFS

È uno dei file system più evoluti e complicati sul mercato, è nato con Winfdows-NT.
Un Volume corrisponde a una partizione o più partizioni aggregate.
Un Cluster è l'unità più piccola leggibile e scrivibile sul disco da parte del SO.
La Master File Table (MFT) è una tabella che contiene la lista dei file sul disco, è composta da record di dimensione fissa.
Un FILE è una collezione di più attributi di varia natura, possono essere metadati, flussi di dati, ecc… e non necessita di un nome.
Gli Attributi del File possono essere contenuti nello stesso record (residenti) o in record diversi (non residenti).
Le directory sono basate sulla struttura dati B+ Tree per ricerche efficienti.
La gestione blocchi liberi è basata su bitmap (visione d'insieme sullo stato di allocazione).

Supporta hard-link, soft-link e montaggio di altri File system (nelle ultime versioni).
Supporta il journaling;
Può fare compressione e cifratura selettiva dei file, di recente anche la cifratura a livello di volume.
Posso fare copie shadow di volumi con tecnica copy-on-write (ridondanza dei dati e ripristino di file vecchi).

FILE SYSTEM UNIX (V7)

V7 È il primo file system unix, è basato su i-node e il numero di accessi è limitato anche per file molo grandi. C'era la limitazione sulla partizione, sulla lunghezza del nome del file. Il numero di i-node era un intero a 16 bit.
Man mano si è evoluto passando da v7 a ext, a ext2, a ext3, fino a ext4.

Con ext2 si è introdotta la possibilità di aumentare i caratteri del file, estesi i limiti sulla dimensione del file, si è evoluta la strategia di allocazione: i blocchi sono suddivisi in gruppi, ogni gruppo ha il suo i-node. Si tende ad allocare i blocchi tutti vicini. Nella creazione di un nuovo file, oltre alla ricerca di un i-node disponibile, si ricerca spazio nel gruppo. Ogni gruppo ha una bitmap per tener conto dei blocchi liberi. L'ideale sarebbe conoscere a priori la dimensione del file e allocarlo contiguamente. Per cercare di perseguire questo obbiettivo (pur non conoscendo a priori la dimensione del file) si cercano tipicamente 8 blocchi liberi contigui e si inizia ad allocare li. Il sistema alloca più spazio del necessario nella speranza che il file al massimo li utilizzi tutti ma non di più.

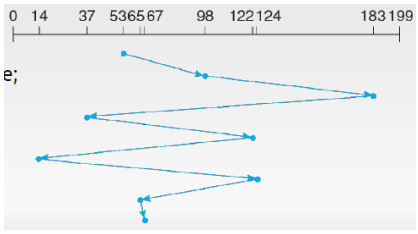
Con ext3 si mantiene la retrocompatibilità con ext2 e si introduce il journalig.

Con ext4 gli i-node sono lenti per file molto grossi, per ovviare a ciò si affiancano gli extent, un segmento contiguo di blocchi allocati appresso al file, questi funzionano appieno se il file è contiguo. Introduce l'allocatore multiblocco, se il processo richiede un grande numero di blocchi, inizialmente si allocavano man mano, col multiallocatore si ricerca un area libera del disco e si alloca tramite extent (deve essere specificato). Con l'allocazione rtardata cerca di allocare i blocchi in maniera ritardata nel tempo: quando un processo richiede blocchi per usarli, il SO apparentemente soddisfa la richiesta, ma in realtà usa il buffer cache per ritardare la scrittura e l'allocazione: in questo modo li scrive o quando scade il timer dei 30 sec per mantenrere i dati integri oppure quando finisce il processo e così sceglie un area contigua il più preciso possibile. Si può fare una deframmentazione on-line cioè mentre il file system sta ancora lavorando (utile nei server).

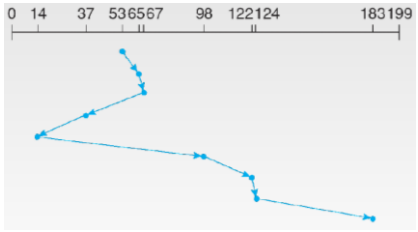
Il BTRFS (B-Tree FS o Butter FS) è il futuro dei file system. È in sviluppo ed è utilizzabile in beta. È del tutto nuovo, è retrocompatibile ed è rivoluzionario. L'idea fondamentale è l'uso del copy-on-write, si ha la possibilità di clonare un file o un gruppo di questi (cartelle o volumi) in paritcolare permette di prendere un intero dsco (o più) e creare un pullBTRFS da cui posso creare volumi (distribuito su più dischi) che posso clonare a loro volta (snapshot) creandone uno storico.
Occupa lo spazio solo necessario ed è basato sui B-Tree.
È possibile utilizzare le primitive send/recive sui volumi: il file System sa le differenze tra i vari snapshot, se il disco si guasta si perde tutto: ciò per cui si usa è copiare i dati in un altro disco, si può inviare tramite la rete un istantanea del volume, quando ne creo un'altra e devo backupparla non invio l'intera istantanea, ma con questa primitiva, capisco le differenze tra gli snapshot e invio solo le differenze risparmiando lavoro, traffico in rete ecc…
si possono effettuare operazioni pesanti anche on-line (deframmmentazione, aggiunta nuovo disco, rimozione vecchio, ecc..).
Tutti i dati prevedono un checksum (CRC) per avere una ridondanza dei dati per accorgersi se si sono subite manipolazioni o presenza di errori.
Permette di comprimere trasparentemente i file.
Tramite RAID aggrega più dischi in un pull.

SCHEDULING DEL DISCO

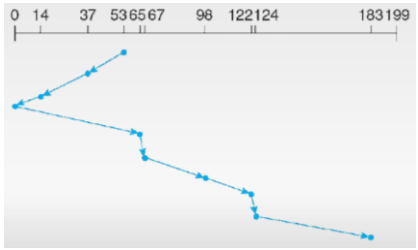
Una tecnica per ottimizzare i movimenti elettromeccanici della testina è lo scheduling del movimento di questa sul disco. L'obbiettivo è il massimizzare il numero di richieste soddisfatte nell'unità di tempo e il minimizzare il tempo di accesso. Il SO può ignorare dove si trovi il blocco da leggere, ma può richiedere dove si trovi fisicamente la testina per massimizzare la lettura. Il SO gestisce una coda di richieste che possono essere esaudite una alla volta. Man mano che il controller si libera ne esaudisce un'altra. Il SO può modificare la coda avendone una visione globale in modo da ottimizzare i costi di posizionamento. Per otitmizzare il seek-time basta ordinare le richieste in modo crescentre.



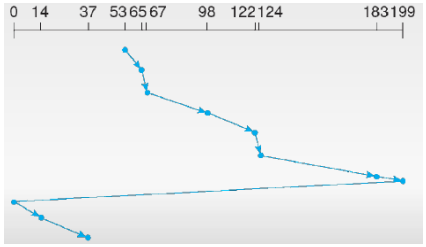
FCFS: in ordine di richiesta, semplice ma totalemte inefficace.
L'ordine usato è: *98, 183, 37, 122, 14, 124, 65, 67;*
La posizione iniziale della testina: cilindro *53;*
La distanza totale percorsa: 640 tracce;



SSTF: in ordine di vicinanza, piu veloce ma non ottimale, non è equo e genera starvation.
L'ordine usato è: 65, 67, 37, 14, 98, 122, 124, 183;
La distanza totale percorsa: 236 tracce;



SCAN: schedula le richieste assumendo il fatto che la testina debba sempre muoversi in una direzione finchè non raggiunge l'estremo, si conosce la posizione della testina e la sua direziione di spostamento. Tra le richieste pendenti viene ricercata la posizione più vicina in direzione dello spostamento, arrivato allo 0 si cambia verso. Scanssione uniforme.
L'ordine usato è: 37, 14, 0, 65, 67, 98, 122, 124, 183;



C-SCAN: Usa la SCAN in maniera circolare, una volta a 0 passa al max (199).
L'ordine usato è: 65, 67, 98, 122, 124, 183, **199**, **0**, 14, 37;

SISTEMI RAID

Serve per avere prestazioni superiori, più capacità di lettura porta ad una lettura piu veloce, migliore resistenza ai guasti, ecc..
I dischi lavorano indipendente e vengono visti come un'unica unità logcia, è possibile aumentare il numero di dischi senza dare problemi o stoppare il servizio. Al livello superiore il sistema RAID è visto come un unico disco.
Esistono varie configurazioni raid (usiamo per es. 4 dischi base):

- Raid0 (striping): Unisco i 4 dischi idealmente in un unica unità logica che ne rappresenta la distribuzione, l'unità logica è suddivisa in "stripe" (lo stripe è un multiplo del blocco di base). Uso il Round Robin, non ha ridondanza, è semplice, prestazioni ottimali con letture di grandi volumi poiché i dato sono spalmati sui vari dischi.
- Raid1 (mirroring): Ho tanti dischi copia quanti quelli originali, è piu veloce del raid0 in lettura, in scrittura si deve fare il lavoro anche nella copia, non ho problemi di guasti dato che ho una copia del disco, ma devo usare il doppio della memoria (overhead di memoria).
- Raid2 (sriping a livello di bit): Introduco ridondanza, ma riduco il numero di dischi a mia disposizione. Uso un codice di correzione degli errori (tipo hamming): ho 4 dischi di dati e 3 disch di ridondanza (posso rilevare al massimo un solo errrore). Ho ottime prestazioni a discapito di dover sincronizzare la rotazione dei dischi.
- Raid3 (sriping a livello di bit con bit di parità): uso un solo disco di parità per gli errori, li individuo, ma non li posso correggere.
- RAID4 (striping a livello di blocchi con XOR sull'ultimo disco): uso i primi N-1 dischi in Raid0, l'ultimo lo uso per la ridondanza, se si guasta un disco posso recuperarne i dati facendo lo XOR tra gli altri.
  Es.: ho i dischi dallo 0 al 3 e il disco di ridondanza P0-3. Se si guasta il disco 2 basterà fare lo XOR del disco 0,1,3,P0-3 per recuperarne i dati.
- RAID5 (striping a livello di blocchi ma con informazioni di parità distribuite): i blocchi di ridondanza non sono contenuti solo in un disco, ma sono distribuiti così da salvarsi nel caso in cui il disco a guastarsi è proprio l'ultimo.

MEMORIE FLASH

I dispositivi basati su memoria flash sono molto differenti rispetto ai dischi elettromeccanici, non hanno ritardi fisici quindi hanno un accesso ditipo costante. Rispetto alle memorie fisiche sono veloci da leggere, consumano poco, sono resistenti agli urti e sono silenziosi. Il loro contenuto è suddiviso in blocchi, a loro volta i blocchi sono sono suddivisi in pagine.
Ha però anche alcuni difetti:

- Prima di essere sovrascritto il blocco deve essere cancellato.
- Ogni blocco ha un numero limitato di riutilizzi.
- L'unità fondamentale è la pagina, ma per la scrittura devo usare i blocchi, quindi se devo sovrascrivere una pagina devo agire sull'intero blocco.

Per ovviare a quest'ultimo problema è possibile spostare dati contenuti nelle pagine in altri blocchi così da limitare gli sprechi, inoltre finchè il SO non ricica i blocchi dei file cancellati, per il controller questi sono ancora occupati.
Tramite la TRIM il SO comunica con il controller i blocchi che possono essere liberati.

LABORATORIO

COMANDO	OPERAZIONE
CAT **CAT [file]**	Senza parametri aggiuntivi, prende l'input e lo restituisce come output. Restituisce in output il contenuto del file.
ECHO -e	Rende possibile utilizzare i caratteri speciali.
MORE [file]	Comando identico a CAT, ma con tabulazione a pagina.
LESS [file]	Comando identico a MORE, ma permette la ricerca.
PIPELINE ($) C1|C2	Invoca processi separati in successione, prima di essere eseguiti scambia input e output tra C1 e C2.
LPR	Comando per stampare documenti.
SORT **SORT -r**	Ordina alfabeticamente un flusso testuale. Ordinamento random.
HEAD	Seleziona il primo elemento.
TAIL	Seleziona l'ultimo elemento.
CUT	Elabora un flusso di dati, posso specificare un separatore ad ogni riga e filtrare i campi.

Es: CUT -d: -f 1-4	Dal primo al quarto campo.
SCRIPT	Eseguibile.
./	Esegue, bisogna specificare il path da dove prendere il file eseguibile.
GZIP	Comprime i dati, può comprimere sul file o sui dati.
GUNZIP	Decomprime.
TAR	Flusso non compresso tar. Comprime ciò che c'è
EXIT STATUS (?) **Es: echo $?**	È un intero che indica se il processo è andato a buon fine, se è settato a 0 non ci sono stati errori. Visualizza l'exit status dell'ultimo comando.