Sistemi Operativi

Processi

Un processo è un programma in esecuzione, può cambiare stato tra new, ready, running, waiting, terminated. Un processo necessita del Program Counter (per memorizzare l'istruzione successiva da eseguire, questo rende il processo un'entità attiva), dei registri della CPU, uno stack (per gli indirizzi di rientro, ed altro), una zona di memoria per le variabili globali, e un heap (memoria dinamica per gli oggetti).
Ogni processo è rappresentato dal proprio Process Control Block (**PCB**), che memorizza principalmente lo stato del processo (uno dei 5), il Program Counter, i registri di CPU, le informazioni sullo scheduling (come priorità, code di scheduling, …), informazioni sulla gestione della memoria (page tables, …), informazioni sullo stato dell'I/O e sulla contabilizzazione delle risorse (il tempo in cui rimane attivo nella CPU, ecc).
Quando non è in esecuzione il PCB del processo può essere inserito nella **waiting queue** (una waiting queue può essere la I/O queue quando è in attesa di completamento di un processo di I/O) oppure nella **ready queue** (quando è in attesa di essere processato dalla CPU).
Se si vogliono eseguire più processi di quelli che starebbero nella memoria principale, si spostano nella memoria secondaria (**spooling**).
Il long-term scheduler (**job scheduler**) preleva processi pronti da eseguire dalla memoria secondaria per metterli nella memoria principale, determina quindi i processi che saranno eseguiti in futuro. Ha quindi la funzione di controllo del grado di multiprogrammazione (determina il numero di processi presenti in memoria centrale) e deve anche bilanciare processi con prevalenza di I/O e processi con prevalenza di elaborazione. Il short-term scheduler (**CPU scheduler**) preleva un processo dalla memoria centrale per portarlo in CPU ed eseguirlo (*dispatched*), questo scheduler interviene molto più frequentemente. Il **medium-term scheduler** è un livello di scheduling intermedio presente in sistemi operativi ad esempio con partizione del tempo, la sua funzione è rimuovere se necessario processi dalla memoria e successivamente reimmetterli (**swapping**).
I processi possono creare processi figli, vi è la possibilità che un processo ne termini un altro, ma può solo se quest'altro è suo figlio. Vi sono due possibilità di gestione dello spazio di indirizzi in caso di processi figli: il processo figlio è duplicato dal padre, nel processo figlio si carica un nuovo programma.
I processi in esecuzione possono comunicare tra loro (sono quindi **cooperanti**) o meno (sono **indipendenti**). Per quelli che comunicano tra loro vi sono due schemi principali: comunicazione tramite memoria condivisa o scambio di messaggi. Nel primo caso si attribuisce uno spazio di memoria a cui entrambi i processi possono accedere, nel secondo caso è il SO a occuparsi dello scambio dei messaggi. Tali schemi possono essere adottati insieme
La comunicazione **client/server** può impiegare: socket (coppia di socket che comunicano tra loro), RPC (invocazione di una procedura che risiede su un'altra macchina) o RMI (versione in Java delle RPC, la differenza è che queste possono trasferire oggetti, le RPC solo strutture dati standard).

L'indirizzo 127.0.0.1 noto come loopback simula un client/server sulla stessa macchina.

Thread

Un thread è un percorso di controllo d'esecuzione all'interno di un processo. Più thread in un processo condividono lo stesso spazio di indirizzi (mentre ogni processo ha il proprio spazio di indirizzi, diverso dagli altri processi) questo implica la condivisione della memoria su cui lavorano. I thread a livello utente richiedono minor tempo per essere creati e gestiti rispetto a quelli a livello kernel.
Vi sono quindi tre modelli che descrivono le relazioni thread-utente / thread-kernel: da molti a uno (più thread a livello utente sono associati ad un thread a livello kernel), da uno a uno (associa un thread a livello utente con uno a livello kernel), da molti a molti (più thread a livello utente sono associati a un numero minore o uguale di thread a livello kernel.

CPU Scheduling

Lo scheduling della CPU consiste nella scelta di un processo dalla ready-queue a cui assegnare la CPU.
Si parla di **scheduling preemptive** quando lo scheduling interviene solo se il processo termina o passa da solo dallo stato di esecuzione allo stato di attesa (si può sottrarre un processo alla CPU prima che lui non ne necessiti più), si parla di **scheduling non-preemptive** se lo scheduling interviene anche per portare un processo dallo stato running allo stato ready o dallo stato waiting allo stato ready (una volta data la CPU ad un processo gliela si lascia finché non ne ha più bisogno).
Il **dispatcher** è il modulo che passa effettivamente il controllo della CPU ai processi scelti dal short-term scheduler. Esegue: il context-switch, il passaggio alla modalità utente e il salto alla giusta posizione del programma per avviarne l'esecuzione.
I criteri di scheduling sono:
- **Utilizzo della CPU** (massimo), nella realtà si va da un 40 a un 90 percento di utilizzo.
- **Throughput** (massimo), il numero di processi completati nell'unità di tempo.
- **Turnaround time** [tempo di completamento] (minimo), l'intervallo che intercorre tra la sottomissione del processo a il completamento totale dell'esecuzione.
- **Waiting time** (minimo), è la somma degli intervalli d'attesa passati nella ready-queue.
- **Response time** (minimo), è il tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta. Algoritmi di scheduling:
First Come First Served (FCFS), è il più semplice, tuttavia può far si che brevi processi attendano processi molto lunghi, questo rende il tempo medio d'attesa abbastanza lungo. Questo algoritmo è non-preemptive.
Shortest Job First (SJF), è l'algoritmo di scheduling ottimale, ma il più difficile da implementare in quanto ordina i processi da eseguire in base alla lunghezza delle operazioni da effettuare sulla CPU, i processi più brevi hanno maggiore priorità. E' difficile però prevedere la lunghezza della successiva sequenza di operazioni della CPU.

Per stimare la lunghezza delle successive sequenze si usa la formula $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ (in cui si stima il futuro attribuendo molta importanza alla storia passata [$\alpha=0$] o molta importanza alla storia recente [$\alpha=1$]).
Questo algoritmo è un caso particolare dell'algoritmo generale di scheduling per priorità, che si limita ad assegnare la CPU al processo con priorità più elevata. Il SJF ma in generale lo scheduling per priorità hanno potenziali problemi di starvation (ovvero processi con priorità basse potrebbero non essere mai eseguiti), questo può essere risolto tramite l'aging, ovvero con il passare del tempo la priorità dei processi in coda aumenta. Questo algoritmo può essere sia preemptive che non-preemptive.
Round Robin (RR), è un algoritmo estremamente adatto per un sistema a tempo ripartito: si assegna la CPU al primo processo della coda dei processi pronti per q unità di tempo, dopodiché si ha la prelazione della CPU e si mette il processo in fondo alla ready-queue. In questo modo ogni processo ottiene 1/n del tempo di elaborazione della CPU in frazioni di al massimo q unità di tempo e non deve quindi attendere per più di (n-1)*q unità di tempo. Il problema principale è la scelta della durata del quanto di tempo, se è troppo lungo lo scheduling RR si riduce a uno FCFS, se è troppo breve il carico di scheduling dovuto al tempo dei context-switch diventa eccessivo. Questo algoritmo è preemptive.
Multi-Queue Scheduling, questo algoritmo permette l'uso di diversi algoritmi per diverse classi di processi (suddivisi in diverse code). Le più comuni sono la coda dei processi interattivi (foreground) con scheduling RR e la coda dei processi a lotti [batch] (background) con scheduling FCFS.
La versione delle multi-queue con retroazione (multilevel feedback queue scheduling) permettono a un processo di passare da una lista ad un'altra (cambiandovi l'algoritmo di scheduling adottato).
Spesso i sistemi supportano più processori, in genere ogni processore ha una propria coda di processi ready (o di thread). In caso di multi-processore si possono adottare ulteriori algoritmi:
- *predilezione per il processore:* per evitare svuotamento e ripopolamento delle cache dei 2 processori, si tende una volta iniziato con un processore a elaborare sempre il processo con lo stesso processore.
- *bilanciamento del carico*: in modo automatico, o per mezzo di un altro processo i processi attivi sui processori si muovono tra essi per bilanciare (e distribuire) il carico di elaborazione.
- *scheduling multi-core*: invece che avere più processori distinti (più chip) si ha un unico processore con più unità di calcolo al suo interno.
- *sistemi di virtualizzazione*: offre una o più CPU virtuali a ogni macchina virtuale in esecuzione sul sistema, spartendo la CPU fisica.
I sistemi operativi che gestiscono i thread a livello kernel devono occuparsi dello scheduling dei thread, e non di quello dei processi. Quasi tutti i SO attuali utilizzano scheduling basato su priorità e favoriscono in genere i processi interattivi rispetto ai processi a lotti (batch).

Sincronizzazione Processi

Dato un gruppo di processi sequenziali cooperanti che condividono dati, è necessario garantirne la mutua esclusione. Si tratta di assicurare una sezione critica di codice sia utilizzabile da un solo processo o thread alla volta, per evitare problemi di incoerenza dei dati. Quando più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi si parla di **race condition**.
La **sezione critica** è una parte di codice in cui il processo può modificare dati comuni, quando è in esecuzione tale porzione di codice non si deve permettere a nessun altro di entrare nella propria sezione critica. L'esecuzione delle sezioni critiche è mutuamente esclusiva nel tempo. Per risolvere il problema della sezione critica vanno rispettati tre requisiti:
- **Mutua esclusione**: se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
- **Progresso**: se nessun processo è in esecuzione nella propria sezione critica, e qualche processo desidera entrare nella sua, solo i processi che si trovano fuori dalle proprie sezioni non critiche possono partecipare come candidati ad accedere (quindi quelli che si sono "prenotati").
- **Attesa limitata**: se un processo ha chiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nella loro (prima o poi questo ci entrerà).
La **soluzione di Peterson** è una possibile risposta ai problemi di sincronizzazione (però non usata nelle macchine moderne). Per accedere alla sezione critica il processo P_i assegna innanzitutto a **flag[i]** (variabile condivisa dai processi) il valore true (indicando l'intenzione di entrare nella sezione critica), poi assegna alla variabile condivisa **turn = j** (indicando il turno dell'altro processo), infine esegue un **while(flag[j] && turn ==j)** vuoto, per rimanere in attesa di avere il permesso per procedere. Successivamente quando il while di ferma, entra nella sezione critica, al termine della quale esegue un **flag[i]** = false indicando l'intenzione di uscire dalla sezione. Il punto cruciale è l'assegnazione di turn all'altro processo, in questo modo se entrambi i processi arrivano in concorrenza a quel punto, l'ultimo a modificare turn farà eseguire l'altro processo. Tale soluzione soddisfa i 3 criteri sopra citati.
In generale a livello hardware sono presenti dei lock che si acquisiscono quando si vuole eseguire una sezione critica e si rilasciano quando si esce da tale sezione. Questi lock tramite la manipolazione degli interrupt creano un'esecuzione atomica della sezione.
I semafori sono utilizzabili per risolvere diversi problemi di sincronizzazione e sono realizzabili in modo efficiente, sopratutto se è disponibile un'architettura che permette le operazioni atomiche. I semafori possono essere binari (0 o 1 → **mutex locks**) o semafori contatore (0,infinito). Quando si vuole prendere il lock si effettua P(S) che decrementa il valore e quando lo si rilascia si effettua V(S) che lo reincrementa. Gli svantaggi dei semafori sono la loro condizione di attesa attiva, che spreca cicli della CPU tali semafori sono anche detti **spinlock** (perché "girano" mentre attendono il semaforo che si liberi). Per superare questo problema si può modificare l'implementazione, facendo si che quando un semaforo è in attesa si blocchi (viene

spostato in una lista di attesa apposita) e quando il semaforo sarà libero verrà riattivato, prendendolo dalla lista di attesa.
Quando ciascun processo di un insieme di processi attende indefinitamente un evento che può essere causato solo da uno dei processi dello stesso insieme si parla di **deadlock**.
Ci sono diversi famosi problemi di sincronizzazione, come il problema dei produttori-consumatori con bounded buffer, dei readers-writers (in cui si contempla che più processi lettori accedano agli stessi dati, ma non che uno scrittore e un altro scrittore o lettore accedano contemporaneamente gli stessi dati) e dei cinque filosofi (problema di assegnare varie risorse a diversi processi evitando situazioni di stallo e d'attesa indefinita). Il sistema operativo deve fornire dei mezzi di protezione contro gli errori di sincronizzazione, uno tra i costrutti proposti sono i monitor, che offrono un meccanismo interno di sincronizzazione per la condivisione di tipi di dati astratti (mutua esclusione).
Una variabile condizionale consente a una procedura di monitor di sospendere la propria esecuzione finché non riceve un segnale. I SO offrono meccanismi come semafori, mutex (mutual exclusion), spinlock e variabili condizionali per il controllo dell'accesso ai dati condivisi.
Una transazione è un'unità di programma da eseguire in modo atomico, cioè le operazioni ad essa associate vanno eseguite nella loro totalità o non si devono eseguire per niente. Per assicurare la proprietà di atomicità anche nel caso di malfunzionamenti, si può usare la registrazione con scrittura anticipata. Si registrano tutti gli aggiornamenti nel log, che si mantiene in una memoria stabile. Se si verifica un crollo del sistema, si usano le informazioni conservate nel log per ripristinare lo stato dei dati aggiornati (tramite le operazioni undo e redo).

Deadlocks

Un deadlock si verifica quando in un insieme di processi ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. Vi sono 3 principali metodi di gestione dei deadlock:
- impiegare un protocollo che pervenga o eviti le situazioni di deadlock, assicurando che il sistema non vi entri mai.
- permettere al sistema di entrare in deadlock e quindi effettuarne il ripristino.
- ignorare del tutto il problema fingendo che nel sistema non si verifichino mai situazioni di deadlock (tale soluzione è la più adottata, tra cui UNIX e Windows).
Un deadlock può presentarsi solo se all'interno del sistema si verificano contemporaneamente quattro condizioni necessarie: **mutua esclusione** (una risorsa non può essere condivisa da più processi in contemporanea), **possesso e attesa** (un processo che possiede una risorsa non deve possederne altre), **impossibilità di prelazione** (una volta assegnate delle risorse non si devono togliere a un processo per assegnarle ad un altro) e **attesa circolare** (imporre un ordinamento totale all'insieme di tutti i tipi di risorse e un ordine crescente di numerazione per le risorse richieste da ciascun processo).

Per prevenire deadlock è sufficiente che almeno una delle precedenti condizioni non si verifichi. Un altro metodo per evitare deadlock è di disporre a priori di informazioni su come ciascun processo intende impiegare le risorse.

Memoria centrale

Gli algoritmi di gestione della memoria per sistemi operativi multiprogrammati variano molto. Il fattore più rilevante che incide maggiormente sulla scelta del metodo da eseguire è l'architettura disponibile sulla macchina. E' necessario controllare la validità di ogni indirizzo di memoria generato dalla CPU, se tali indirizzi sono corretti, devono essere tradotti in un indirizzo fisico. Tali controlli per essere efficienti devono essere effettuati dall'architettura del sistema, che secondo le sue caratteristiche pone vincoli al sistema operativo.
Per aumentare le prestazioni è necessario tenere in memoria parecchi processi, rendendo quindi la memoria *condivisa*. Un tipico ciclo d'esecuzione di un'istruzione prevede che l'istruzione sia prelevata dalla memoria (fetched), decodificata (decoded) ed eseguita (executed), i risultati possono essere salvati in memoria.
La CPU può accedere ad un numero ristretto di aree di memoria: solamente alla memoria centrale e ai registri che ha incorporati. Si deduce quindi che i dati che non sono in memoria, per essere elaborati dalla CPU devono prima essere portati in memoria centrale. Gli accessi alla memoria sono frequenti ma lenti, perciò si è creata una memoria cache di veloce accesso che replica i dati più utilizzati della memoria centrale. E' importante determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, una semplice implementazione consiste nell'utilizzare due registri: detti **registro base** e **registro limite** (rispettivamente contengono il più piccolo indirizzo legale della memoria fisica e la dimensione dell'intervallo ammesso). In questo modo la CPU confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri, qualsiasi tentativo di accedere alle aree di memoria non assegnategli restituisce il controllo al SO tramite una trap.
In genere gli indirizzi del programma sorgente sono simbolici (es. parte da 0 e arriva a n, ma degli indirizzi veri non gli sarà assegnato l'intervallo 0-n), un compilatore di solito associa (bind) questi indirizzi simbolici a indirizzi rilocabili. Il *likage editor* oppure il *loader* fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (es indirizzo 14 diventa 7014). Tale associazione (binding) può essere effettuato in varie fasi del percorso:
- **Compilazione**: se in questa fase si sa dove il processo risiederà in memoria, si può generare **codice assoluto** (se si sa a priori che un processo utente inizia alla locazione r, anche il codice generato dal compiler inizierà alla posizione r).
- **Caricamento**: se in fase di compilazione non si sa dove risiederà il processo, il compilatore deve generare **codice rilocabile**. Si ritarda quindi l'associazione finale degli indirizzi alla fase di caricamento (a questa fase).
- **Run-time**: se durante l'esecuzione il processo può essere spostato da un segmento di memoria ad un altro, si deve ritardare l'associazione degli indirizzi fino al runtime (scelta maggiormente utilizzata).
Un indirizzo generato dalla CPU di solito si indica come **indirizzo logico**, un indirizzo visto dall'unità di memoria, cioè caricato nel Memory Address Register (MAR), si

indica come **indirizzo fisico**. Con i metodi di associazione nelle fasi di compilazione e caricamento, gli indirizzi logici prodotti sono uguali a quelli fisici, mentre con il metodo a runtime sono diversi. Lo spazio degli indirizzi logici è l'insieme di tutti gli indirizzi logici generati da un programma; mentre lo spazio degli indirizzi fisici corrispondenti a tali indirizzi logici è lo spazio degli indirizzi fisici. Durante l'esecuzione l'associazione da indirizzi virtuali a fisici è fatta dalla MMU, che somma all'indirizzo logico il valore presente nel registro di rilocazione, creando così l'indirizzo fisico (es. indirizzo logico 765, registro di rilocazione 14000, indirizzo fisico 14765).
Il **dynamic loading** permette di caricare una procedura solo quando viene richiamata, in modo che se una procedura non viene adoperata, non si carica.
Per essere eseguito un processo deve trovarsi in memoria centrale, ma si può trasferire temporaneamente in memoria ausiliaria da cui si riporta in memoria centrale al momento di riprenderne l'esecuzione, questo scambio si chiama **swapping**. Un altro momento in cui avviene lo swapping è quando si presenta un processo con priorità maggiore e prende la CPU al posto di quello attualmente attivo, o anche in caso di termine del quanto di tempo di un processo.
Un banale metodo di gestione della memoria è l'**allocazione contigua della memoria**. Il SO si può allocare in memoria altra o in memoria bassa, di solito si sceglie la seconda in modo che sia vicino all'interrupts vector. Con questa metodologia la MMU fa corrispondere dinamicamente l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione; questo consente al SO di cambiare dinamicamente le proprie dimensione (è sufficiente aumentare il registro di rilocazione e tutto viene shiftato). Questa allocazione avviene tramite la suddivisione della memoria in partizioni di dimensione fissa. Ogni partizione deve contenere esattamente un processo e quando una partizione è libera può essere occupata da un processo presente nella coda d'ingresso (questo metodo non è comunque più in uso).
In generale è sempre presente un insieme di buchi di diverse dimensioni tra i vari processi in memoria. Quando un nuovo processo necessita di memoria si cerca un buco in grado di contenerlo. E' sempre presente una lista che tiene traccia dei buchi e delle loro dimensioni. Per assegnare un buco a un processo ci sono tre criteri:
- **First-fit**: si assegna il primo buco abbastanza grande per contenere il processo.
- **Best-fit**: si assegna il più piccolo buco in grado di contenere il processo.
- **Worst-fit**: si assegna il buco più grande che si ha a disposizione, sperando che il rimanente spazio sia sufficiente per un altro processo.

Quando si caricano e si rimuovo i processi si crea **frammentazione esterna**, ovvero se lo spazio di memoria libera è sufficiente per soddisfare una richiesta ma non è contiguo. Una soluzione può essere la compattazione ma è possibile solo se la rilocazione è dinamica. Quando invece si assegna ad un processo più memoria di quella che effettivamente necessita e utilizza si parla di **frammentazione interna**.
Un altro metodo di gestione della memoria è la **paginazione**, che permette che lo spazio degli indirizzi fisici non sia contiguo. Consiste nel suddividere la memoria fisica in blocchi di dimensione costante, detti anche frame, e suddividere anche la memoria logica in blocchi di pari dimensione, detti pagine. Quando si deve eseguire un processo si caricano le sue pagine nei frame disponibili, prendendoli dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria centrale.
Ogni indirizzo della CPU è diviso in due parti: un **numero di pagina** (p) e uno

scostamento (**offset**). Il numero di pagina serve come indice per la tabella delle pagine, contenente l'indirizzo di base in memoria fisica di ogni pagina, questo indirizzo di base, combinato con l'offset definisce l'indirizzo della memoria fisica che si invia all'unità di memoria. Esempio se 2^m è lo spazio degli indirizzi logici e 2^n la dimensione di una pagina quantificata in numero di indirizzi allora $2^m / 2^n = 2^{m-n}$ è il numero di pagine utilizzare per rappresentare l'intero spazio di indirizzi. Quindi in un indirizzo logico m-n è il numero di bit per rappresentare la pagina e n è il numero di bit per rappresentare l'offset (che spazia all'interno di una singola pagina). La paginazione non è altro che una forma di rilocazione dinamica, a ogni indirizzo logico si fa corrispondere uno fisico. Con questo metodo si può evitare la frammentazione esterna ma non quella interna, la memoria risulterà come un insieme contiguo di frame che però anche se sono vicini appartengono a processi diversi. Ogni processo ha una page table dedicata, e viene usata per tradurre gli indirizzi logici in fisici ogni volta che il sistema operativo deve associare esplicitamente un indirizzo fisico a un indirizzo logico (oppure è usata dal dispatcher della CPU per impostare l'architettura di paginazione quando a un processo sta per assegnare la CPU). In pratica per accedere ad un byte occorrono 2 accessi alla memoria, uno per accedere alla page table e l'altro per accedere alla memoria fisica dopo che si ha l'indirizzo corretto preso dalla page table. Per risolvere questo problema si implementa una piccola cache detta TLB (tranlation look-aside buffer) che è una memoria associativa costituita da una chiave e un valore. Quando vi è un indirizzo da tradurre la TLB lo confronta con i suoi elementi contemporaneamente, se c'è è risolto, se no si accede alla page table e poi alla memoria. La hit ratio della TLB è solitamente alta.
Per quanto riguarda la protezione, di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, il **bit di validità**, che è valido se la pagina corrispondente è nello spazio d'indirizzi logici del processo, e invalido altrimenti (quindi se sta eseguendo un processo le sue pagine avranno bit 1, e le pagine degli altri processi avranno bit 0 così non vi si può accedervi). E' importante parlare di pagine condivise, ad esempio se più utenti utilizzano un software le pagine dei processi (di questo software) dei vari utenti puntano agli stessi frame in memoria, non è che ognuno crea le sue copie, sarebbe dispendioso.
Anche la page table dovrà stare in memoria centrale, se si vuole evitare di collocarla in modo contiguo (per i problemi detti in precedenza) si può pensare di paginare anche lei. Per l'architettura a 32 bit è realizzabile dividendo ulteriormente l'indirizzo logico: 12 bit sono l'offet della pagina, i 20 bit rimanenti sono divisi in 2 parti: 10 bit per la sotto-pagina e 10 bit per l'offset della sotto pagina (questa combinazione darà il numero di pagina richiesto che combinato con l'offset darà l'indirizzo fisico), creando due livelli.
Per l'architettura a 64 bit non è conveniente perché si parlerebbe di paginazione a 3/4 livelli e gli accessi in memoria sarebbe troppi (e il tempo crescerebbe). Si possono impiegare tabelle di tipo hash o tabelle invertite (non implementabile).

Ricapitolando i seguenti elementi sono utili per confrontare i metodi di gestione della memoria.
Architettura: per i metodi con partizione singola o con più partizioni è sufficiente un semplice registro di base oppure una coppia di registri (base e limite). Per i metodi di

paginazione e segmentazione sono necessarie tabelle di traduzione per definire la corrispondenza degli indirizzi.
Prestazioni: aumentando la complessità dell'algoritmo, aumenta anche il tempo necessario per tradurre un indirizzo logico in uno fisico. Nei sistemi più semplici infatti è sufficiente un confronto, o sommare un valore all'indirizzo logico. Per la paginazione e segmentazione invece per velocizzare si possono utilizzare registri veloci per la tabella, tuttavia se invece si trova in memoria, gli accessi possono essere decisamente lenti. Qui subentra una memoria associativa (TLB) che può limitare il calo delle prestazioni.
Frammentazione: Per un dato gruppo di processi, il livello di multiprogrammazione si può aumentare solo compattando più processi in memoria. Occorre ridurre lo spreco di memoria o la frammentazione. Sistemi con unità di allocazione di dimensione fissa, come lo schema con partizione singola e la paginazione, soffrono di frammentazione interna. Mentre sistemi con unità di allocazione di dimensione variabile, come lo schema con più partizioni e la segmentazione, soffrono di frammentazione esterna.
Rilocazione: Una soluzione al problema della frammentazione esterna è data dalla compattazione, che implica lo spostamento di un programma in memoria, senza che il programma stesso si accorga del cambiamento. Ciò richiede che gli indirizzi logici siano rilocati dinamicamente al momento dell'esecuzione, non del caricamento.
Swapping: I processi si copiano dalla memoria centrale alla memoria ausiliaria, e successivamente si ricopiano in memoria centrale a intervalli fissati dal sistema operativo. Ciò permette di inserire contemporaneamente in memoria più processi da eseguire.
Condivisione: Un altro mezzo per aumentare il livello di multiprogrammazione è quello della condivisione del codice e dei dati tra diversi utenti. Poichè si devono fornire piccoli pacchetti di informazioni è necessario l'uso della paginazione o della segmentazione.
Protezione: Con la paginazione o la segmentazione diverse sezioni di un programma utente si possono dichiarare di sola esecuzione, lettura o lettura e scrittura.

Memoria Virtuale

E' utile poter eseguire processi il cui spazio degli indirizzi logici superi quello disponibile per gli indirizzi fisici. La memoria virtuale è una tecnica che permette di associare grandi spazi degli indirizzi logici a quantità più ridotte di memoria fisica. In questo modo si aumenta il grado di multiprogrammazione, incrementando l'utilizzo della CPU. Grazie alla memoria virtuale, processi distinti possono condividere librerie di sistema e memoria. La memoria virtuale si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. L'espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre, per due o più processi, il vantaggio di condividere i file e la memoria, mediante la condivisione delle pagine.
La memoria virtuale è comunemente implementata tramite **paginazione su richiesta**, che trasferisce in memoria una pagina solo quando si incontra un riferimento alla pagina stessa, ovvero solo quando serve realmente. Con il **lazy swapping** si intende

che non si carica mai in memoria una pagina che non sia necessaria. Quando un processo sta per essere caricato in memoria, anziché caricare tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. Per distinguere le pagine presenti in memoria da quelle nei dischi è presente lo schema basato sul bit di validità. Con il bit valido si intende che la pagina corrispondente è valida ed è presente in memoria, mentre con il bit invalido si intende che o la pagina non è valida (è di un altro processo) oppure è valida ma è ancora nel disco (in memoria è vuota). In caso si voglia accedere a una pagina con bit invalido si lancia un **page fault**, che può essere dato dall'accesso a una pagina vuota o di un altro processo. In caso di page fault per pagina assente, si carica in memoria la pagina dal disco, e si riesegue l'istruzione che ha causato il page fault. Quindi la paginazione su richiesta utilizza 2 strumenti: la tabella delle pagine (che contiene anche il bit di validità) e la memoria secondaria (che contiene tutte le pagine di un programma, anche quelle non caricate in memoria centrale).
Il kernel del sistema operativo consulta una tabella interna per stabilire la locazione della pagina in memoria ausiliaria, quindi individua un frame libero e vi trasferisce la pagina prelevandola dalla memoria ausiliaria. La page table viene aggiornata per riflettere tale modifica e si riavvia l'istruzione che aveva causato l'eccezione di pagina mancante. La paginazione su richiesta si può usare per ridurre il numero dei frame assegnati a un processo. Questo metodo può aumentare il grado di multiprogrammazione, permettendo che più processi siano disponibili per l'esecuzione in un dato momento, migliora anche l'utilizzo della CPU. Consente inoltre l'esecuzione di processi i cui requisiti di spazio di memoria superano la memoria fisica disponibile, eseguendoli in memoria virtuale.
Il tempo di accesso effettivo alla memoria è = (1-p)*[tempo di accesso alla memoria] + p*[tempo di gestione dell'assenza di pagina], dove p è la probabilità che si verifichi un'assenza di pagina.
La memoria virtuale consente l'utilizzo di una modalità efficiente di creazione di processo conosciuto con il nome di copiatura su scrittura (**copy-on-write**), in cui i processi genitore e figlio condividono inizialmente le stesse pagine, poi se uno dei due scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina.
Il caso di **over-allocation** (sovrallocazione) si ha quando durante l'esecuzione di un processo utente si verifica un page fault, ma tutta la memoria è in uso. E' necessario quindi un algoritmo di sostituzione delle pagine, in cui quando è richiesta una pagina che non è caricata in memoria, e non ci sono frame liberi, sostituisca uno dei frame occupati con il frame corretto. In linea generale aumentando il numero di frame, si diminuisce il numero di assenze di pagine.
Gli algoritmi usati per la sostituzione delle pagine sono diversi:
- **FIFO**: associa ad ogni pagina l'istante di tempo in cui è stata portata in memoria, se si deve sostituire una pagina si seleziona quella presente in memoria da più tempo, il tutto tramite una coda FIFO: quando si carica una pagina in memoria la si inserisce nell'ultimo elemento della coda (è presente l'anomalia di Belady, con la quale la frequenza delle assenze di pagine può aumentare con l'aumentare del numero di frame assegnati).

- **OPT**: si sostituisce la pagina che non si userà per il periodo di tempo più lungo (è però impossibile da realizzare in quanto non si può prevedere quali pagine saranno richieste in futuro).
- **LRU**: si sostituisce la pagina che non è stata utilizzata per il periodo più lungo (approssimazione dell'OPT, invece che col futuro, col passato). Si associa ad ogni pagina l'istante in cui è stata usata per l'ultima volta, quando occorre sostituire una pagina si sceglie quella con il valore più lontano nel tempo. Si può realizzare con contatori (ogni elemento della tabella ha un valore che verrà aggiornato quando è utilizzata) o tramite una pila (ogni volta che si fa un riferimento a una pagina, la si estrae dalla pila e la si colloca in cima, in questo modo in cima alla pila si trova sempre la pagina usata per ultima e in fondo si trova la pagina usata meno recentemente).
- **Approssimazioni** dell'LRU (a ogni pagina è associato un bit di riferimento che viene modificato a 1 quando si fa riferimento a quella pagina)
→ Algoritmo con bit di riferimento supplementari: si hanno più bit per ogni pagina e se al momento n è usata una pagina, il suo bit in posizione n viene modificato (in questo modo si ha un ordine di utilizzo, es. 10011011 (su 8 momenti, 5 è stata usata).
→ Second-Chance: si usa un solo bit, dopo aver selezionato una pagina si controlla il bit di riferimento, se il valore è 0 si sostituisce la pagina, se è a 1 si da una seconda chance (azzerando il suo bit di riferimento) e si passa alla successiva pagina (una pagina a cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non sono state sostituite o non gli sia stata offerta una seconda chance).
Quest'ultimo algoritmo può essere migliorando aggiungendo un bit di modifica e considerando insieme bit di riferimento e modifica. Se entrambi sono a 0 il frame non è stato recentemente ne usato ne modificato (si può sostituire), se entrambi sono a 1 il frame è usato recentemente e modificato.
E' necessario anche stabilire un criterio per l'allocazione della memoria libera ai diversi processi, algoritmi comuni sono l'**allocazione uniforme**: ovvero suddividendo equamente m frame liberi tra n processi assegnando m/n frame a ognuno. Oppure l'**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione (più è grande più memoria gli si darà).
Il **dirty bit** è un bit associato ad ogni frame che indica che tale frame è stato modificato ma non salvato in memoria, quando un blocco di memoria dev'essere sostituito il suo corrispondente dirty bit è controllato per vedere se è necessario trascriverlo in memoria secondaria prima che sia sostituito o può semplicemente essere rimosso.
Se un processo non ha spazio di memoria sufficiente per il proprio insieme di lavoro, si ha una **thrashing**, ovvero quando si spende più tempo per la paginazione che per l'esecuzione dei processi. Se a ogni processo si devono fornire frame sufficienti per evitare tale degenerazione, è necessario swapping e scheduling dei processi. Se il grado di multiprogrammazione diventa troppo elevato i processi iniziano a rubarsi frame a vicenda senza eseguire computazioni, degenerando. Questo fenomeno si può limitare usando un **algoritmo di sostituzione locale**, o algoritmo di sostituzione per priorità: se un processo va in thrashing non può prelevare frame ad altri processi, ma solo a se stesso (ai frame che utilizza lui).
La strategia della frequenza delle assenze di pagine (page fault frequency PFF), se la frequenza delle assenze di pagine è eccessiva, significa che il processo necessita di più frame, se la frequenza delle assenze di pagine è molto bassa, il processo potrebbe

disporre di troppi frame. Si possono fissare un limite inferiore ed uno superiore per la frequenza desiderata delle assenze di pagine, se si scende sotto si sottrae un frame al processo, se si va sopra occorre allocare al processo un altro frame.

File System

Un file è un tipo di dati astratto definito e realizzato dal sistema operativo. E' una sequenza di elementi logici (o record), ciascuno dei quali può essere un byte, una riga di lunghezza fissa o variabile, oppure un elemento di dati più complesso, contenuto in memoria secondaria.
Gli attributi dei file sono: nome, identificatore, tipo, locazione, dimensione, protezione, ora, data, id utente. Le operazioni sui file solo: creazione, scrittura, lettura, riposizionamento, cancellazione, troncamento (cancellare il contenuto del file, ma non i file in se).
Ad ogni file aperto sono assicate diverse informazioni: puntatore al file (unico per ogni processo che opera sul file), contatore dei file aperti, posizione nel disco del file e i diritti di accesso.
Poiché le dimensioni dei blocchi fisici dei dispositivi possono non coincidere con quelle dei record logici, può essere necessario riunire un certo numero di record logici in modo da ottenere una dimensione pari a quella di un blocco fisico.
Ad un file vi si può accedere tramite: **accesso sequenziale** (le informazioni del file si elaborano ordinatamente, un record dopo l'altro), ad **accesso diretto** (un file è formato da record di lunghezza fissa, ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un particolare ordine). Altri metodi di accesso ai file possono essere tramite **indice**, che contiene puntatori ai vari blocchi e per trovare un elemento del file occorre prima cercare nell'indice e poi usare il puntatore corrispondente per accedere direttamente al file.
Una directory a livello singolo in un sistema multiutente causa problemi di nominazione, poiché ogni file deve avere un nome unico. Una directory a due livelli limita questo problema all'ambito relativo a ciascun utente creando una directory distinta per ciascun utente, che dispone di una directory contenente i suoi file. La generalizzazione del concetto di directory a due livelli è la directory con struttura ad albero. Le strutture delle directory a grafo aciclico permettono la condivisione di sottodirectory e file, ma complicano la ricerca e la cancellazione.
Nasce infatti il concetto di **link**, ovvero un puntatore in un file che punta ad un altro file o directory (in UNIX vi è differenza tra hard link e soft link).
Una struttura a grafo generale permette la massima flessibiltà nella condivisione dei file e delle directory, ma talvolta richiede operazioni di **garbage collection** per recuperare lo spazio inutilizzato nei dischi. I dischi sono suddivisi in uno o più volumi, ciascuno contenete un file system o privo di struttura. Una volta montati, i file all'interno del volume sono disponibili per l'uso. I file system si possono smontare per disabilitarne l'accesso o per attività di manutenzione.
I metodi di condivisione di file in rete possono essere ad esempio tramite FTP, file system distribuito (che permette la visibilità nel calcolatore locale delle directory remote, o il World Wide Web che tramite un programma di consultazione (browser) accede ai file remoti.

E' necessario che i file siano dotati di un meccanismo di protezione. Ogni tipo d'accesso ai file si può controllare separatamente: lettura, scrittura, esecuzione, aggiunta, elencazione del contenuto di directory, …
Si possono inoltre suddividere gli accessi tra: proprietario, gruppo e universo.

Realizzazione File System

Il file system risiede permanentemente in memoria secondaria.
I trasferimenti tra memoria centrale e dischi si eseguono per blocchi. Ciascun blocco è composto da uno o più settori. Secondo l'unità a disco, la dimensione dei settori è compresa tra 32byte e 4096byte, di solito è pari a 512byte. I file system consentono di memorizzare, individuare e recuperare facilmente i dati.
I dischi si possono segmentare in partizioni, allo scopo di controllarne l'uso e consentire più, anche diversi, file system per disco. Questi file system si montano in un file system logico per renderli disponibili all'uso. I file system si realizzano secondo una struttura stratificata o modulare: i livelli più bassi hanno a che fare con le caratteristiche fisiche dei dispositivi di memorizzazione; i livelli più alti hanno a che fare con i nomi simbolici e le caratteristiche logiche dei file; i livelli intermedi fanno corrispondere le caratteristiche logiche dei file alle caratteristiche fisiche dei dispositivi.
I file system è composto da molti livelli, quello più basso: **il controllo dell'I/O**, costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e secondaria.
Il **file system di base** deve inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco.
Il **modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi.
Il **file system logico** gestisce i metadati: tutte le strutture del file system, eccetto gli effettivi dati. Mantiene le strutture di file tramite i blocchi di controllo dei file (file control block FCB), contenenti informazioni sui file.
Le principali strutture presenti nei dischi sono: il **boot control block** (contiene le informazioni per l'avviamento di un sistema operativo da quel volume), i **volume control block** (ognuno contiene dettagli riguardo il relativo volume (o partizione), e altre informazioni), le **strutture delle directory** (una per file system, usate per organizzare i file) e i **file control block** (che contengono molti dettagli sui file).
Ciascuna partizione è priva di struttura logica (**raw partition**) se non contiene alcun file system. Nella fase di caricamento del sistema operativo si esegue il montaggio della **root partition**,che contiene il kernel del SO e in alcuni casi altri file di sistema [in Windows il montaggio di ogni volume è effettuato in uno spazio identificato da una lettera, es "F:"].
La realizzazione di una directory può essere effettuata tramite lista lineare (tutti i file concatenati uno dopo l'altro) o tabella hash.
Lo spazio dei dischi può essere allocato in tre modi: allocazione contigua, concatenata e indicizzata. L'**allocazione contigua**: ogni file deve occupare un insieme di blocchi contigui del disco. L'allocazione contigua dello spazio per un file è definita

dall'indirizzo del primo blocco e dalla lunghezza. Accedere a un file il cui spazio è assegnato in modo contiguo è facile, infatti sia l'accesso sequenziale che diretto sono permessi. Si sollevano problemi di frammentazione esterna (e possibili soluzioni best-fit, first-fit, worst-fit).
L'**allocazione concatenata**: ogni file è composto da una lista concatenata di blocchi del disco, i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Ogni blocco contiene un puntatore al blocco successivo (quindi dallo spazio per le informazioni in ogni blocco va tolto quello per la memorizzazione del puntatore). Qui è permesso solo l'accesso sequenziale in quanto non c'è modo di puntare direttamente all'i-esimo blocco. E' possibile allocare **cluster** (insieme di blocchi contigui) invece che i blocchi, riducendo lo spazio che i puntatori occupano e che sottraggono ai dati. Una variante è una file allocation tabl **FAT**, che ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco, si usa come lista concatenata.
L'**allocazione indicizzata**: si raggruppano tutti i puntatori in una sola locazione: il blocco indice. Ogni file ha il proprio blocco indice: un array d'indirizzi di blocchi del disco, dove l'i-esimo elemento del blocco indice punta all'i-esimo elemento del blocco del file. Per la gestione del blocco indice molto grande si può usare: uno **schema concatenato** (si collegano tra loro più blocchi indice), **indice a più livelli** (un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che puntano ai blocchi dei file), **schema combinato** (si hanno i primi 15 puntatori del blocco indice nell'inode del file, i primi 12 di questi puntano a blocchi diretti, gli altri 3 puntano a blocchi indiretti, il primo di questi è un blocco indiretto singolo che punta ad altri blocchi di indirizzi, il secondo a due livelli, e il terzo tre).
Per la gestione dello spazio libero, il sistema ha un lista dello spazio libero, dove sono registrati tutti gli spazi liberi. Una tecnica è il **vettore di bit**: ogni blocco è rappresentato da un bit, se è 1 è assegnato se 0 libero (es. 11010101). Un'altra tecnica è collegare tutti i blocchi liberi tra loro creando una **lista concatenata**.
I metodi di allocazione dello spazio libero influenzano anche l'efficienza d'uso dello spazio dei dischi, le prestazioni del file system e l'affidabilità della memoria secondaria.
Il danneggiamento di una tabella o il crollo del sistema possono causare discordanze tra le informazioni contenute nelle directory e il contenuto del disco, per il ripristino sono possibili algoritmi di **journaling** tramite i quali tutte le modifiche dei metadati si annotano in modo sequenziale in file di registrazione, dello log. Ogni insieme di operazioni che esegue uno specifico compito di chiama transazione, una volta che le modifiche sono riportate nel file di log, le operazioni si considerano portate a termine con successo. Quando un intera transazione è completata si rimuovono le annotazioni dal log, che è in realtà un buffer circolare, che scrive man mano, sovrascrivendo in modo circolare i dati. Se si verifica un crollo del sistema, nel log possono esserci transazioni, che dovranno essere completate al momento del riavvio.