Sistemi Operativi (appunti)

prof. Claudio Maccherani - Perugia – 2009

Indice

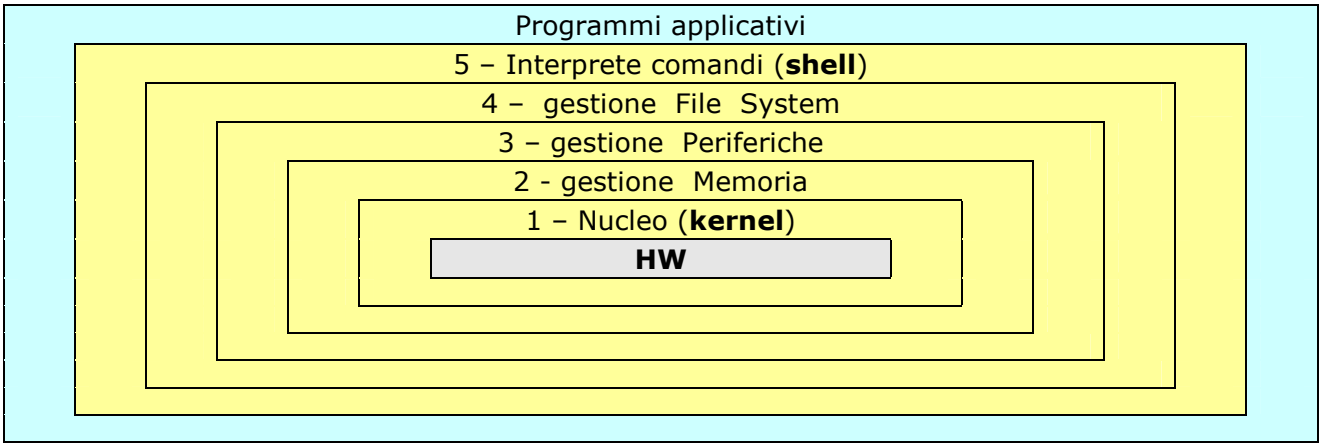
Bibliografia consultata:

- appunti personali del corso di *Sistemi per l'elaborazione dell'informazione II*, corso di laurea in "Scienze dell'Informazione", Università degli Studi di Pisa, 1980
- G.Callegarin, L.Varagnolo, *Corso di informatica generale, volume III*, CEDAM, 1984
- F.Sanpietro e O.Sanpietro, *Sistemi operativi e reti di computer*, Tramontana, 2003
- F.Scorsoni, *Informatica, Sistemi operativi, reti e intenet per l'azienda*, Loesher, 2006

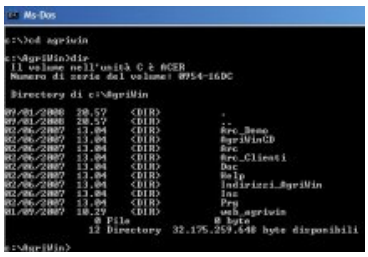
Introduzione

Un Sistema Operativo (SO) è un insieme di programmi che gestiscono le risorse sia hardware che software del sistema di elaborazione rendendone più semplice l'uso (***supporto per l'utente***) ed ottimizzano l'utilizzazione delle stesse (***gestore di risorse***).

Un Sistema Operativo può essere visto come un insieme gerarchico di macchine astratte o macchine virtuali, ciascuna delle quali poggia su quella sottostante, strutturato nei seguenti livelli:



1 - Nucleo (**Kernel**)	gestione processori; creazione, avanzamento, comunicazione e sincronizzazione processi; gestione interruzioni
2 - gestione Memoria	gestione della Memoria Centrale e sua allocazione dinamica ai programmi in esecuzione (anche con virtualizzazione)
3 - gestione Periferiche	uso di periferiche virtuali tramite programmi - driver - che adattano le periferiche virtuali con quelle reali effettivamente usate
4 - gestione File System	organizzazione (logica e fisica) dei file e delle cartelle su memoria di massa o secondaria (disco)
5 - Interprete comandi (**Shell**)	l'interfaccia utente attraverso la quale si impartiscono i comandi al Sistema Operativo



Interfaccia a linea di comando di MS-DOS

L'interfaccia utente (o **shell**) può essere "a linea di comando", tipo MS-DOS, o "grafica" (GUI - *Graphics User Interface*), tipo Windows. Il SO vero e proprio è formato dal **nucleo** o **kernel** che gestisce le funzioni principali e gli altri **moduli** del SO stesso.

screenshot di
Debian Gnu/Linux

In base alle modalità di funzionamento un Sistema Operativo può essere:

- SO Mono Tasking, Mono Utente (un solo programma, un solo utente), uniprogrammazione, tipo MS-DOS
- SO Multi Tasking, Mono Utente (più programmi, un solo utente), multiprogrammazione, tipo Windows. Evoluzione dei sistemi Multi Tasking sono i sistemi Multi Threading (ogni processo è costituito da uno o più thread che possono essere eseguiti contemporaneamente). Windows appartiene a questa categoria di sistemi operativi.
- SO Multi Tasking, Multi Utente (più programmi, più utenti), multiprogrammazione, tipo Unix/ Linux, VMS, MVS (per main frame)
- SO in Tempo Reale (Real Time), per il controllo dei processi industriali, non interagiscono con l'utente, ma con i dispositivi che controllano il processo
- SO a Macchine Virtuali, che permettono di simulare hardware e/o sistema operativo diverso (computer virtuale), ad esempio la finestra DOS di Windows
- SO di Rete (da non confondere con i SO Multi Utente), per la gestione delle LAN (reti locali), ad esempio Windows Server (NT, 2000, 2003, ..); il SO di rete "convive" con i SO dei singoli computer della rete

In questi appunti si esamineranno alcune caratteristiche di un generico SO in multiprogrammazione e multiutente. In questo contesto si hanno più programmi che sembra vengano eseguiti "in parallelo", contemporaneamente, anche se in effetti l'unica CPU li avanza uno alla volta, "a turno".

L'idea di base della multiprogrammazione è quella di sfruttare i "tempi morti" delle lentissime operazioni di Input/Output non costringendo la CPU ad attendere il loro completamento, ma impiegandola nell'avanzamento di altri processi.

Perché sia possibile la multiprogrammazione occorre quindi che la CPU sia svincolata dall'esecuzione delle operazioni di I/O, delle quali si occuperanno appositi processori di I/O (canali, DMA) che lavorano in parallelismo effettivo con la CPU. La CPU (processore centrale) può così occuparsi esclusivamente dell'avanzamento dei processi. È chiaro che in mancanza di processori di I/O non è possibile la multiprogrammazione.

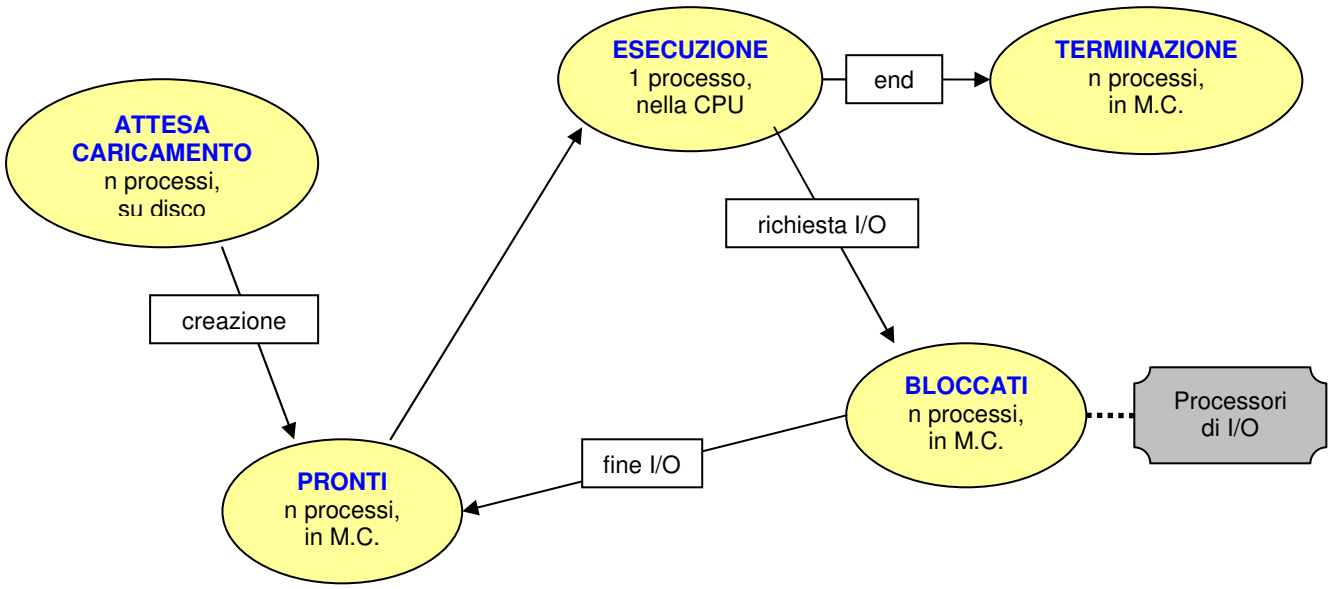
Processi e risorse

Un **processo** è l'insieme delle azioni compiute dal processore per eseguire un programma. Si può semplificare dicendo che "un processo è un programma in esecuzione".
Ogni processo, per l'esecuzione (o, più propriamente, per l'avanzamento), necessita di **risorse** HW e SW. È compito del SO, oltre all'avanzamento dei processi, la gestione di tali risorse.
Le risorse possono essere "consumabili" (ad esempio i messaggi scambiati tra processi) o "riusabili" (quelle che vengono date in uso ai processi e controllate dal gestore di risorse del SO).
Una risorsa che può essere usata da un solo processo alla volta è detta "seriale" (CPU, stampante,...)
Spesso può essere utile costringere il processo a rilasciare la risorsa (prerilascio o preemption).
Lo "**scheduling**" è l'algoritmo di gestione della risorsa.

Stati di un processo

Un processo, dopo la sua creazione ed attivazione, può trovarsi in diversi stati di avanzamento:

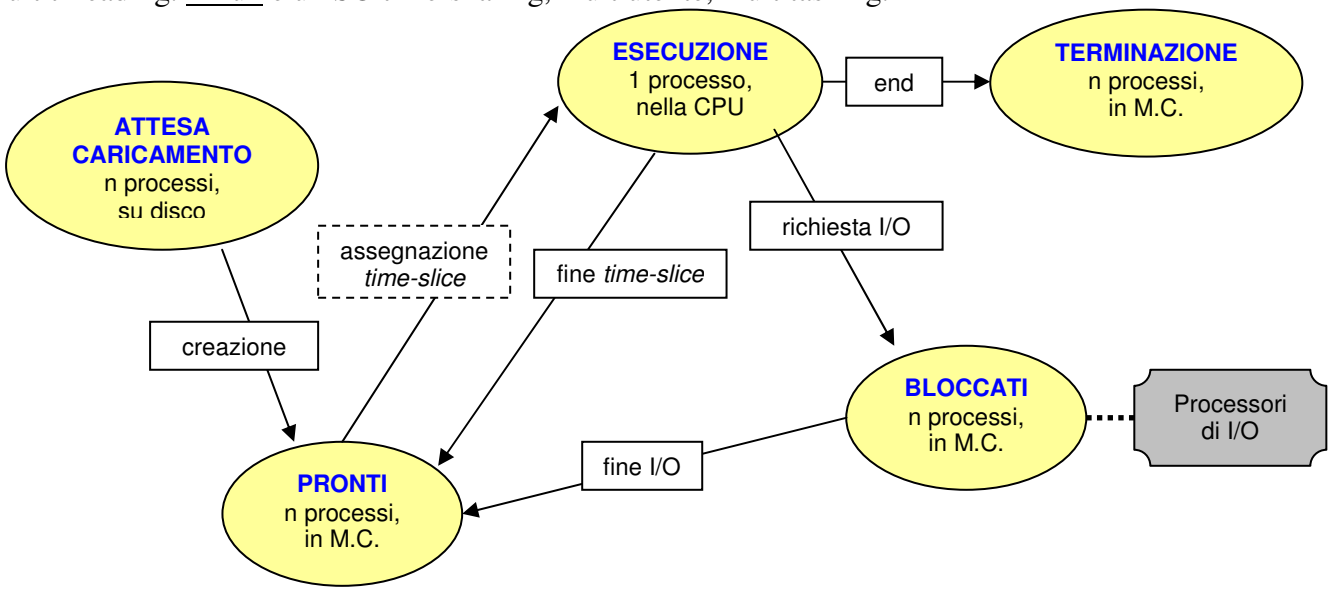
- **Attesa di caricamento** (è stata richiesta l'esecuzione del programma memorizzato su disco, attende di essere caricato in memoria centrale; più processi in attesa di caricamento)
- **Pronto** (il processo è in memoria centrale ed attende che gli venga assegnata la CPU; più processi in stato di pronto)
- **Esecuzione** (il processo è in esecuzione sulla CPU; un solo processo in esecuzione)
- **Bloccato** o in **Attesa** (il processo ha richiesto una operazione di I/O e ne attende il completamento; più processi in attesa; le operazioni di I/O vengono eseguite dai processori di I/O in parallelismo effettivo con la CPU)
- **Terminazione** (il processo ha terminato l'esecuzione può rilasciare le risorse utilizzate, quali la memoria centrale)



Stati dei processi in un sistema operativo in multiprogrammazione a semplice richiesta di I/O

Lo schema illustra un sistema operativo in multiprogrammazione "a semplice richiesta di I/O": un processo libera la CPU solo per richiesta di I/O o per terminazione. Ciò, in alcune situazioni, può

causare il blocco dell'intero sistema (se, ad esempio, si esegue un ciclo infinito - per errore del programmatore - all'interno del quale non sono presenti istruzioni di richiesta di I/O).
Per ovviare a questo inconveniente occorre "costringere" il processo al ***prerilascio*** della CPU (preemption). Quando un processo passa da Pronto ad Esecuzione (gli viene data la CPU) si assegna ad esso un "quanto di tempo" (***time slice***) di qualche decina di millisecondi scaduto il quale, se ancora in esecuzione, il processo viene costretto a rilasciare la CPU.
Un sistema di questo tipo (tutti i moderni sistemi operativi in multiprogrammazione) è un sistema ***time-sharing*** (a suddivisione di tempo). Windows è un SO time-sharing, monoutente, multithreading. Linux è un SO time-sharing, multiutente, multitasking.



Stati dei processi in un sistema operativo time-sharing

Descrittore del processo

Ogni processo ha associato un proprio descrittore di processo (o **PCB**. – Process Control Block) che viene creato in fase di "creazione" del processo e mantenuto in memoria centrale, in un'area riservata al sistema operativo, per tutta la durata del processo. Tale descrittore contiene:

- PID (Process ID, identificatore del processo, un numero progressivo assegnato dal sistema operativo al momento della creazione del processo)
- identificatore del padre (PID del padre, puntatore al suo PCB)
- puntatore alla lista dei processi figli
- stato di avanzamento
- priorità
- limiti della memoria centrale utilizzata
- puntatore alla lista delle risorse in uso
- copia di tutti i registri della CPU (PC, IR, MAR, MDR, accumulatori, …)

Quando un processo rilascia la CPU nel suo PCB viene salvato il contenuto dei registri della CPU. Quando a un processo viene data la CPU la copia dei registri contenuta nel suo PCB viene copiata nei registri fisici della CPU. Ciò consente al processo di riprendere l'esecuzione dal punto in cui era stata sospesa, dall'istruzione puntata dal PC (Program Counter) al momento della sospensione.

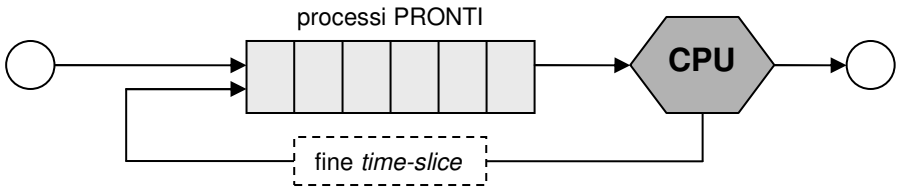
Per ***Creare*** un processo occorre assegnare un'area di memoria centrale, copiare il programma in tale area, creare il suo PCB, porre il processo in stato di "pronto".

Per ***Terminare*** un processo occorre liberare la memoria centrale allocata, rilasciare tutte le risorse da esso utilizzate, aggiornare il PCB del processo che l'ha creato.

Scheduling della CPU

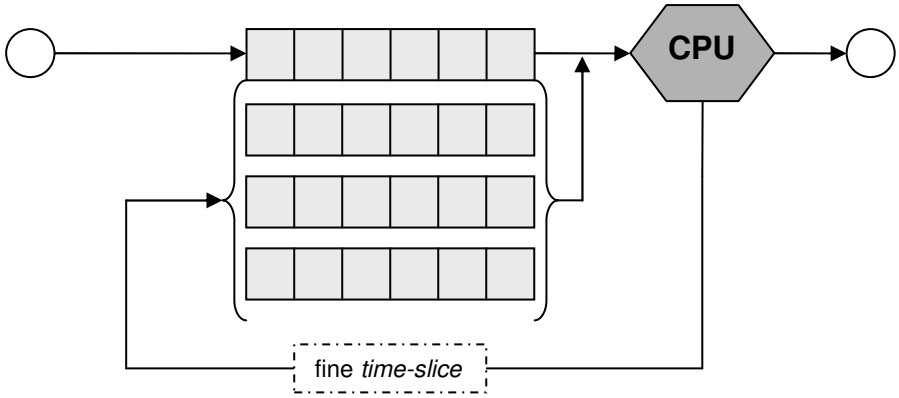
Esistono diverse tecniche di scheduling della CPU (scheduling a basso livello), cioè politiche di assegnamento della CPU a uno dei processi Pronti. Lo scheduling della CPU viene effettuata dal modulo del sistema operativo denominato ***Dispatcher***.

Round Robin



I processi "pronti" vengono inseriti in una coda (FIFO)
Si può tener conto della priorità iniziale del processo ed inserirlo non in fondo alla coda, ma un po' più avanti (in funzione, appunto, del suo livello di priorità).

Multilevel Feedback



I processi "pronti" vengono inseriti in N code. La prima ha priorità massima e slice minimo, l'ultima ha priorità minima e *slice* massimo
(da coda 1 a coda N la priorità diminuisce e lo *slice* aumenta).
I processi della coda i-esima vengono eseguiti quando le code precedenti sono vuote.
Un processo che lascia la CPU per fine *slice* viene inserito nella coda seguente quella dalla quale era stato prelevato (diminuisce la priorità ed aumenta lo *slice*). Un processo che entra in "pronti" per fine I/O o per caricamento viene inserito nella prima coda, quella a priorità maggiore. Questo meccanismo favorisce i processi con alto tasso di I/O, cioè i processi interattivi.

Overhead di sistema

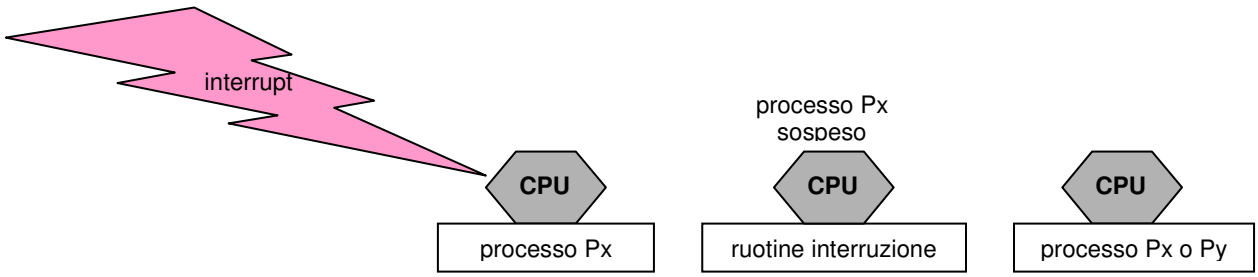
L'assegnazione della CPU a un nuovo processo comporta il salvataggio dello stato (PCB) del processo vecchio, la scelta del processo da avanzare e il caricamento dello stato (PCB) di questo nuovo processo. Il tempo di cambio di contesto (*context-switch time*) è l'***overhead*** di sistema.
Gli algoritmi di *scheduling* debbono essere efficienti, ma non eccessivamente complessi, così da minimizzare il consumo di CPU da parte dell'algoritmo di *scheduling* stesso (e minimizzare l'*overhead* di sistema). Questo vale per tutti i processi del sistema operativo.

Interruzioni

Il passaggio dei processi da uno stato all'altro, durante il loro avanzamento, è causato dal verificarsi di eventi (richiesta di I/O, fine *time slice*, fine I/O, etc.). A parte la richiesta di I/O, che viene fatta dal processo in esecuzione, gli altri eventi sono esterni alla CPU. Per segnalare tali eventi è previsto il ***meccanismo delle interruzioni*** (o ***interrupt***).

Una interruzione è un segnale hardware (a parte la richiesta di I/O, che è una interruzione software, una SVC, Supervisor Call con la quale il processo richiede al sistema operativo una operazione di I/O) che viene inviato dai vari dispositivi e periferiche alla CPU.

Quando arriva una interruzione alla CPU si ha l'automatica sospensione del processo in esecuzione sulla CPU (con il salvataggio del suo PCB) e l'attivazione di una **routine di gestione interruzioni** (caricata in memoria centrale, nell'area riservata al sistema operativo, all'accensione del computer). Tale routine, in base al tipo di interruzione arrivata, prende i provvedimenti del caso. Ad esempio, se l'interruzione segnala la fine di un I/O individuerà il processore di I/O che l'ha inviata ed il processo per il quale è stata inviata - cioè il processo che ha terminato l'I/O -, lo metterà nello stato di pronto e riprenderà l'esecuzione del processo che aveva temporaneamente sospeso; se l'interruzione segnala la fine del time-slice metterà il processo sospeso in pronti ed attiverà lo *scheduling* per scegliere il prossimo processo da avanzare; etc.



Durante l'esecuzione della routine di trattamento delle interruzioni le interruzioni vengono "mascherate" - per non interrompere proprio l'esecuzione della routine di gestione interruzioni - ed accodate per essere esaminate successivamente. Le interruzioni hanno priorità diversa.

Si possono raggruppare le interruzioni in 5 classi, di priorità decrescente:

1) **Machine Check**, malfunzionamento o guasto hardware
2) **SVC**, Supervisor Call, tipicamente una richiesta di I/O
3) **Program Check**, errore di programma (quale overflow, divisione per 0, ...)
4) **External**, causata dal tasto CTRL-C premuto dall'operatore o dal Timer del computer che segnala la fine del time slice
5) **Input/Output**, proveniente dai processori di I/O che segnalano la fine di un I/O

Ciclo di polling

Se si deve gestire un dispositivo periferico con un processore sprovvisto del meccanismo delle interruzioni occorre ricorrere al cosiddetto "ciclo di polling", consistente nel testare periodicamente, da parte del processore, lo stato del dispositivo per vedere se l'operazione di I/O è terminata. Questa "attesa attiva" è molto più dispendiosa, in termini di tempo, dalla "attesa passiva" del meccanismo delle interruzioni.

Gestione delle Risorse

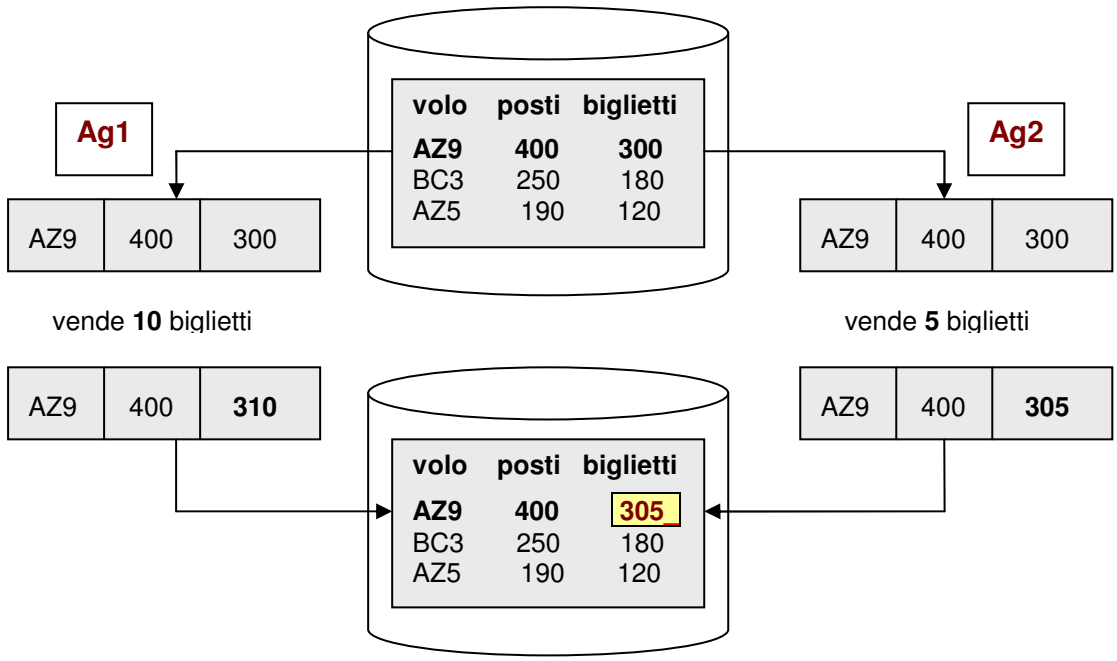
Una risorsa, sia essa HW (memoria, disco, stampante,…) o SW (file, PCB,…), è una qualche cosa necessaria al processo. Essa può essere assegnata in modo statico (per tutta la durata del processo, ad esempio l'area di memoria centrale) o in modo dinamico (viene assegnata, usata e rilasciata dopo l'utilizzo da parte del processo; se il processo può essere costretto anche a rilasciare la risorsa, allora la risorsa è detta "prerilasciabile" - come la CPU).
Il nucleo (o Kernel) del sistema operativo si occupa, come visto precedentemente, della gestione della CPU - scheduling della CPU - e dell'avanzamento dei processi.
La gestione delle altre risorse è effettuata da appositi gestori delle risorse quali il gestore della memoria centrale, il gestore delle periferiche, e il gestore delle informazioni (o File System).

Mutua esclusione

In genere una risorsa riusabile e non prerilasciabile può essere assegnata ad un solo processo per volta, in ***mutua esclusione*** (se un processo ha la risorsa R, nessun altro può utilizzarla finché lui non la rilascia), garantendo un tempo di attesa finito ed evitando situazioni di stallo (o deadlock).

La mutua esclusione va garantita, pena il verificarsi di errori non controllabili.
Prendiamo ad esempio una procedura in rete, funzionante in varie agenzie turistiche, che gestisce la vendita di biglietti aerei per una determinata destinazione:

L'agenzia 1 contratta la vendita del volo AZ9 (del quale le risultano 300 biglietti venduti) , contemporaneamente all'agenzia 2 (alla quale risultano sempre 300 biglietti venduti). La 1 vende 10 biglietti e memorizza la transazione; la 2 vende 5 biglietti e memorizza la transazione.
Il risultato è che alla fine risulteranno venduti solo 305 biglietti (e non 315, come è realmente). Questo è accaduto perché non è stata garantita la "mutua esclusione".



*Esempio di errore causato dalla mancata applicazione della **mutua esclusione***

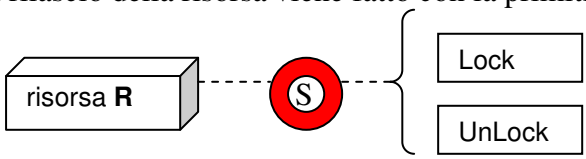
Competizione tra processi (interazione indiretta)

S1) Primitive **Lock** e **UnLock** (attesa attiva)

Per garantire la mutua esclusione (ad esempio ad una variabile contatore in memoria centrale) si associa alla risorsa (in questo caso al contatore) un ***semaforo*** binario che può valere 0 ("verde", risorsa libera) o 1 ("rosso", risorsa occupata).

La richiesta della risorsa viene fatta con la primitiva Lock che testa il semaforo: se il semaforo è a 0 lo pone a 1 ed assegna la risorsa al processo, altrimenti resta in attesa che il semaforo diventi 0 (testandolo ad intervalli regolari).

Il rilascio della risorsa viene fatto con la primitiva UnLock che mette il semforo a 0.



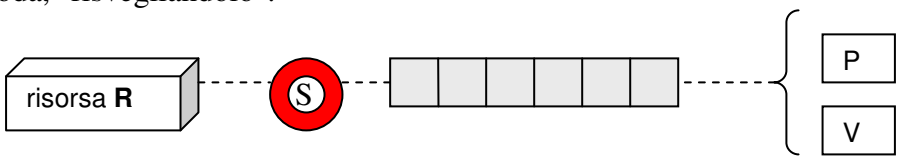
Questo meccanismo, che prevede il test ripetuto del semaforo - attesa attiva - garantisce la mutua esclusione, ma non l'attesa finita (se il semaforo diventa verde prima che venga riesaminato ed un altro processo chiede la risorsa, la ottiene, rimettendo il semaforo a rosso; quando si va a testare il semaforo lo si trova ancora rosso, ma nel frattempo la risorsa è stata data ad un alto processo; in teoria questo potrebbe verificarsi per un tempo indefinito).

S2) Primitive **P (wait)** e **V (signal)** (attesa passiva)

Con questo meccanismo alla risorsa, oltre al semaforo, è associata anche una coda destinata a contenere i PID dei processi che l'hanno richiesta.

La richiesta della risorsa viene fatta con la primitiva V che testa il semaforo: se il semaforo è a 0 lo pone a 1 ed assegna la risorsa al processo, altrimenti pone il PID del processo nella coda di attesa e pone in wait il processo.

Il rilascio della risorsa viene fatto con la primitiva V che controlla se la coda di attesa è vuota: se si pone il semaforo a verde, altrimenti lo lascia a rosso ed assegna la risorsa al primo processo della coda, "risvegliandolo".

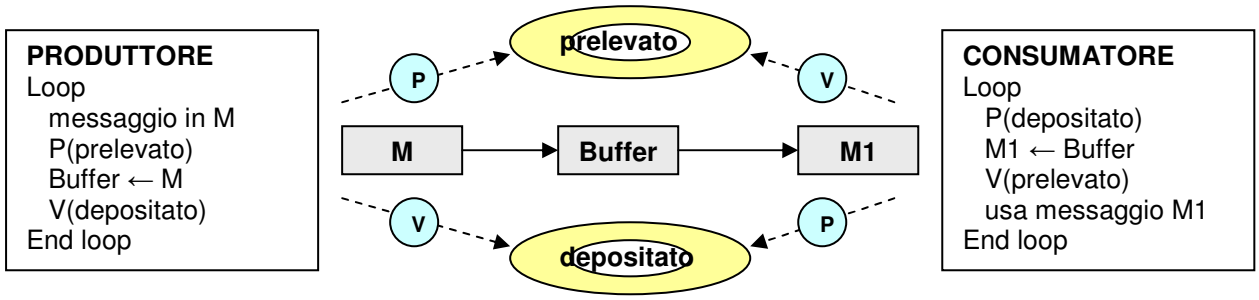


Questo meccanismo - di attesa passiva - garantisce sia la mutua esclusione che l'attesa finita.

Cooperazione tra processi (interazione diretta)

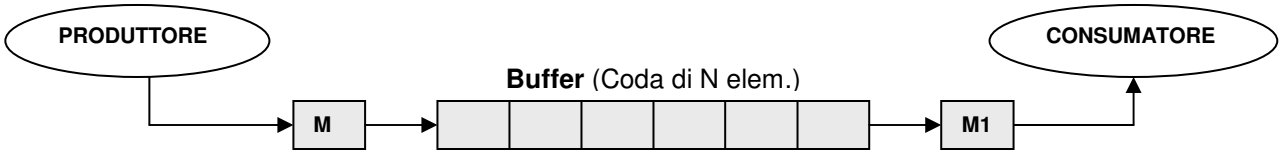
S3) Soluzione tramite **Buffer**

Un caso di cooperazione è dato da un processo che invia di messaggi ad un altro processo che li deve elaborare nello stesso ordine di ricezione. Ad esempio la copia di un file, se implementata in cooperazione, può durare la metà del tempo che impiegherebbe un unico processo che legge e scrive alternativamente. I processi debbono essere "sincronizzati", cioè la "produzione" di un messaggio da parte del processo trasmettente deve avere sempre lo stesso tempo del "consumo" da parte del processo ricevente. La sincronizzazione viene fatta tramite due semafori, "prelevato" inizializzato a verde e "depositato" inizializzato a rosso, e le primitive P e V.



S4) Soluzione tramite **Buffer "circolare"**

Se i tempi di produzione e di consumo sono diversi la precedente soluzione potrebbe essere dispendiosa in termini di tempo, sia per il "produttore" che per il "consumatore".
In questo caso si ricorre ad un buffer circolare ad N posizioni organizzato come una coda FIFO. Il "produttore" si blocca solo quando la coda è piena ed il "consumatore" solo quando la coda è vuota; inoltre "produttore" e "consumatore" possono operare in modo asincrono



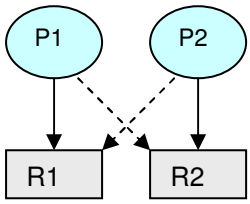
SX) L'uso di semafori risolve sia i problemi di interazione diretta (cooperazione) che quelli di interazione indiretta (competizione) tra processi. Tuttavia esistono costrutti a più alto livello - a livello di linguaggi e di compilatori - rispetto ai semafori per l'interazione fra processi, i **Monitor** e le **Regioni Critiche**.

NB: primitive (Lock, UnLock, P, V), Sezioni Critiche e Monitor sono "indivisibili" (non possono essere interrotte durante la loro esecuzione).

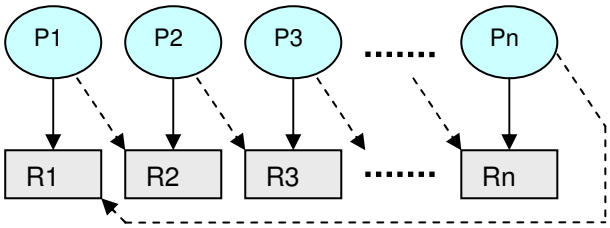
Stallo (deadlock)

L'avere più programmi che competono per l'uso delle risorse può causare situazioni di stallo (deadlock), situazioni nelle quali i processi non possono più avanzare.
Le condizioni per le quali si può verificare lo stallo sono:

- necessità di garantire la mutua esclusione
- attesa di risorse aggiuntive (i processi richiedono le risorse una alla volta e non tutte insieme)
- non prerilascio delle risorse ottenute
- verificarsi di una situazione di "attesa circolare"



Ad esempio il processo P1 ha in uso la risorsa R1 che rilascerà solo dopo aver ottenuto la risorsa R2; P2 ha in uso R2 che rilascerà solo dopo aver ottenuto R1: STALLO!



Per ovviare allo stallo ci sono alcune diverse tecniche:

- la ***Prevenzione*** dello stallo (*deadlock prevention*) consiste nell'invalidare una o più condizioni di stallo (ad esempio si obbligano i processi a chiedere le risorse tutte insieme); in genere si previene l'attesa circolare con l'**allocazione gerarchica delle risorse**
- la ***Fuga*** dallo stallo (*deadlock avoidance*) permette di rifiutare la richiesta di risorsa che causa (o potrebbe causare) stallo; l'algoritmo più noto di questa strategia è l'**algoritmo del banchiere**
- il ***Riconoscimento e recupero*** dello stallo (*deadlock detection and recovery*) non previene lo stallo, ma riconosce quando esso avviene (ad esempio esaminando periodicamente il **grafo di allocazione delle risorse**) e lo elimina (ad esempio forzando uno dei processi a terminare)

Avvio e chiusura

La procedura di avvio del sistema operativo, all'accensione dell'elaboratore, si chiama **bootstrap** o **IPL** (*Inizial Program Load*). In genere avviene in tre fasi:

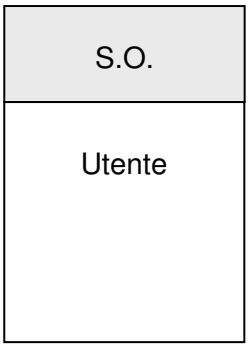
- sequenza di pre-boot o **POST** (*Power On Self Test*) che esegue controlli sull'HW
- esecuzione della routine **loader** della ROM BIOS che individua l'unità da cui avviare il sistema e quindi copia in memoria il caricatore del sistema operativo - ***boot sector***
- esecuzione del **boot sector** che carica in RAM la parte del sistema operativo (nucleo e shell) che deve sempre essere residente in memoria

Normalmente anche in fase di spegnimento del computer occorre eseguire una procedura di chiusura che aggiorna alcune informazioni di sistema.

Gestione della memoria

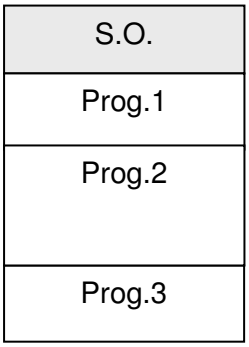
Ogni processo, per avanzare, necessita che il relativo programma risieda in memoria centrale (immagine del processo). Tale programma, in linguaggio macchina, è costituito da istruzioni e dati.
Lo **spazio fisico** è l'insieme degli indirizzi accessibili fisicamente e costituisce la **memoria fisica**; lo **spazio logico** è uno spazio astratto di indirizzi e costituisce la memoria logica.
Nel programma si fa riferimento ad indirizzi logici (l'indirizzo logico della prima istruzione sarà 0), ma durante l'esecuzione tali indirizzi debbono essere fisici (l'indirizzo fisico della prima istruzione sarà l'indirizzo della locazione di memoria a partire dalla quale è stato caricato il programma). Occorre quindi "trasformare" gli indirizzi logici in indirizzi fisici (o **assoluti**). Questa "traduzione" è detta ***Mapping*** o ***Rilocazione degli indirizzi***. Ci sono diverse tecniche di gestione della memoria.

Contigua singola



La memoria è divisa in due parti, una riservata al sistema operativo ed una ai programmi utente. Il programma utente sarà sempre caricato a partire dalla stessa locazione fisica. In questo caso non c'è possibilità di multiprogrammazione, ma solo di SO uni programmati (es. MS-DOS).
La **rilocazione** è **Assoluta**, gli indirizzi vengono trasformati da logici a fisici al momento della creazione dl programma eseguibile, dal compilatore o dal linker.

Partizioni Fisse



La memoria è divisa, al momento dell'installazione del SO, in partizioni fisse (non necessariamente uguali) ciascuna delle quali potrà contenere un programma.
Il SO, per la gestione, si avvarrà di una **tabella delle partizioni** di tante righe quante sono le partizioni contenente, per ogni partizione, indirizzo iniziale, indirizzo finale (o lunghezza), PID del programma che la occupa (0 se libera).
La **rilocazione** è **Statica**, gli indirizzi vengono trasformati da logici a fisici al momento del caricamento del programma, dal caricatore (loader), che trasformerà gli indirizzi sommando all'indirizzo logico l'indirizzo iniziale della partizione dove viene caricato il programma.

Il grado di multi programmabilità è fisso (N partizioni, N programmi), spreco di memoria (un solo programma per partizione), impossibilità di eseguire programmi più grandi della partizione maggiore.

Partizioni Variabili

Le partizioni vengono create dinamicamente, in base alle richieste. Inizialmente si hanno due partizioni, una del SO ed una (tutto il resto della memoria) libera. Quando si crea una nuova partizione, dividendo la partizione libera in due. E così via. Quando un programma termina si libera la sua partizione (e se quella precedente e/o seguente sono libere si accorpano in un'unica partizione libera). Dopo un certo tempo si può avere il problema della "frammentazione" (tante piccoli frammenti di partizioni libere, ciascuno dei quali troppo piccolo per contenere un altro programma).
Il SO, per la gestione, si avvarrà di una tabella simile a quella delle partizioni fisse, solo con un numero variabile di righe.

La scelta della partizione libera da utilizzare può essere fatta con diverse strategie:

a) **first fit** (viene scelta la prima partizione sufficiente a contenere il programma; favorisce il formarsi di una grande partizione libera verso il fondo della memoria)

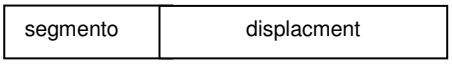
b) **best fit** (viene scelta la più piccola partizione sufficiente a contenere il programma, così da non frammentare inutilmente le grosse partizioni)

c) **worst fit** (viene scelta la più grande partizione che può contenere il programma)

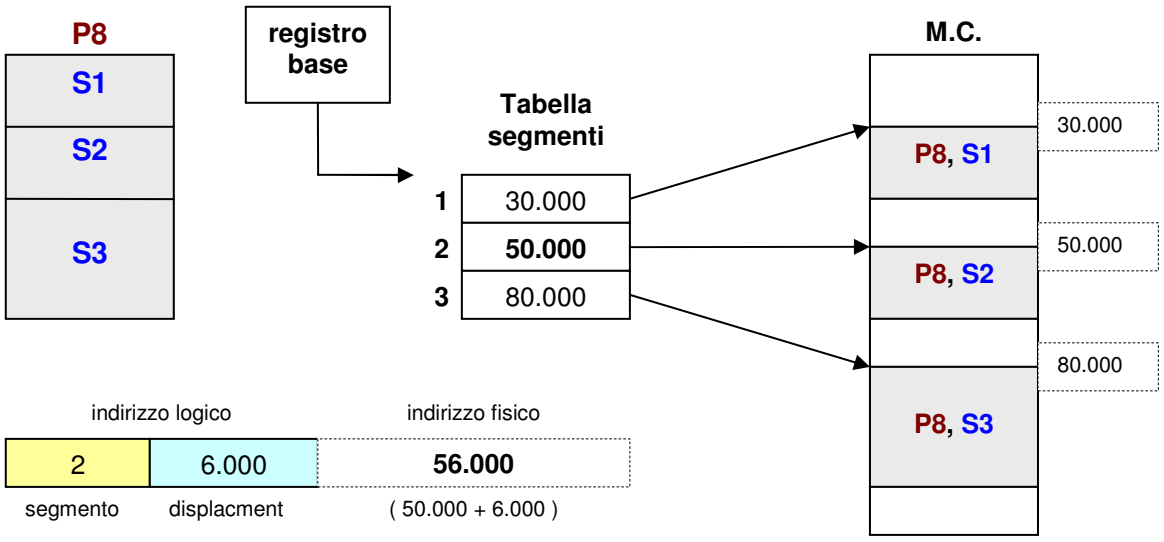
La **rilocazione** può anche essere statica, ma in genere è **Dinamica**: il programma viene caricato in memoria senza alcuna trasformazione degli indirizzi, che vengono trasformati da logici a fisici al momento dell'esecuzione. L'indirizzo iniziale della partizione viene posto in uno speciale **registro base** e durante l'esecuzione l'indirizzo fisico viene calcolato sommando all'indirizzo logico il contenuto del registro base.
In presenza di rilocazione dinamica, per eliminare la frammentazione, è possibile effettuare il *compattamento* delle partizioni occupate in modo da creare una unica grande partizione libera, risultato della somma di tutti i frammenti liberi, operazione questa però abbastanza "costosa" in termini di overhead di sistema e non può essere utilizzata in sistemi interattivi.

Segmentazione

Le tecniche di gestione memoria fin qui illustrate prevedono che ogni programma sia logicamente lineare e venga caricato in un'area contigua di memoria. Con la segmentazione questo vincolo cade: un programma viene suddiviso in più segmenti (ad esempio dal compilatore, anche su indicazione del programmatore) ciascuno dei quali potrà essere successivamente caricato in una diversa partizione. In ogni segmento gli indirizzi logici partono da 0;
il campo indirizzo delle istruzioni è formato numero del ***segmento*** [1 byte] e ***displacment*** (indirizzo all'interno del segmento) [2 byte].



La memoria centrale viene gestita a partizioni variabili, con l'ausilio della **tabella delle partizioni**, contenente, per ogni partizione, indirizzo iniziale, indirizzo finale (o lunghezza), PID e numero segmento del segmento del programma che la occupa (0 se libera).
La **rilocazione** è **Dinamica** e per risolvere gli indirizzi, per ogni programma si usa una **tabella dei segmenti** (di tante righe quanti sono i segmenti del programma e avente, per ogni segmento, l'indirizzo della partizione nella quale è stato caricato il relativo segmento del programma) il cui indirizzo è contenuto nel **registro base**.



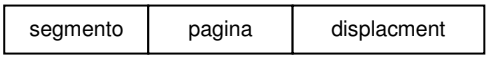
Segmentazione con partizioni variabili

Paginazione

La segmentazione ridimensiona il problema della frammentazione, ma non lo elimina. La tecnica della paginazione è simile a quella della segmentazione, con la differenza che le pagine sono a lunghezza fissa (1KB, 2KB, 4KB). Il programma viene, dal compilatore, suddiviso in pagine a lunghezza fissa. La memoria centrale è suddivisa in "sedi pagina" - o "blocchi" - di eguale dimensione (delle pagine) adatte a contenere le pagine dei programmi. Le pagine di un programma possono essere caricate ovunque in memoria. Con la paginazione non esiste più il problema della frammentazione e la memoria viene sfruttata al 100% (un po' di "spreco" ci può essere sull'ultima pagina di ciascun programma). Il campo indirizzi delle istruzioni, come per la segmentazione, è formato è formato da numero ***pagina*** e ***displacment*** (indirizzo all'interno della pagina).
La memoria centrale viene gestita con l'ausilio di una **tabella di occupazione pagine**, (di tante righe quante sono le "sedi pagina") contenente, per ogni riga, l'indicazione di chi occupa la "sede pagina" corrispondente [PID e numero pagina del processo che la occupa] o 0 se libera.
La **rilocazione** è **Dinamica** e per risolvere gli indirizzi, per ogni programma si usa una **tabella delle pagine**, (di tante righe quante sono le pagine del programma e avente, per ogni riga, il numero della sede pagina nella quale è stata caricata la relativa pagina del programma) il cui indirizzo è contenuto nel **registro base**.

Paginazione e Segmentazione

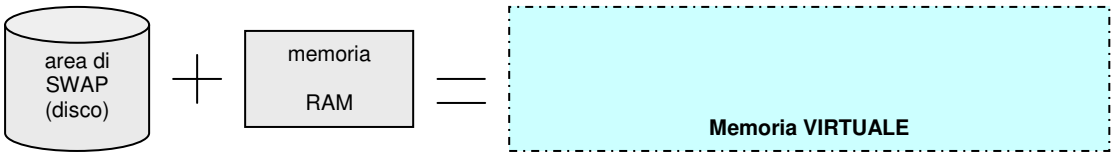
Questa tecnica prevede sia la segmentazione che la paginazione: i programmi sono suddivisi in segmenti che, a loro volta, sono suddivisi in pagine e la memoria centrale è suddivisa in "sedi pagina". Il campo indirizzo delle istruzioni è composto da numero ***segmento***, numero ***pagina*** e ***displacment***.



Per ogni programma c'è una tabella dei segmenti, puntata dal registro base, ogni riga della quale punta alla tabella delle pagine del relativo segmento.

Caricamento dinamico e Memoria Virtuale

Con le precedenti tecniche di gestione memoria i processi in esecuzione debbono essere caricati completamente in memoria. Il **caricamento dinamico**, invece, consente che un processo sia parzialmente presente in memoria centrale e che le sue parti siano caricate e scaricate da disco quando serve. Ad ogni processo è assegnata un'area di memoria centrale (che lo contiene parzialmente) ed un'area su disco - **area di swap** - che contiene la sua immagine, istruzioni e dati, e che viene aggiornata durante l'esecuzione (il contenuto dell'area di swap del processo è diverso da quello del file contenete le istruzioni del programma, che non viene modificato).
I processi, oltre alla memoria centrale, usano anche l'area di swap su disco, usano cioè una **Memoria Virtuale** più grande della memoria fisica. Il loro spazio indirizzabile - indirizzi logici – può essere maggiore dello spazio fisico della memoria centrale e possono essere eseguiti contemporaneamente tanti processi dei quali è presente in memoria solo una piccola parte.
A fronte di questo indubbio vantaggio, occorre pagare un prezzo: lo swapping - carico e scarico di parti di processo da/su area di swap - rallenta l'esecuzione del processo (il tempo di accesso alla RAM è dell'ordine dei nanosecondi, mentre quello al disco è dell'ordine dei millisecondi).
I processi permanenti del SO non vengono mai gestiti dinamicamente.



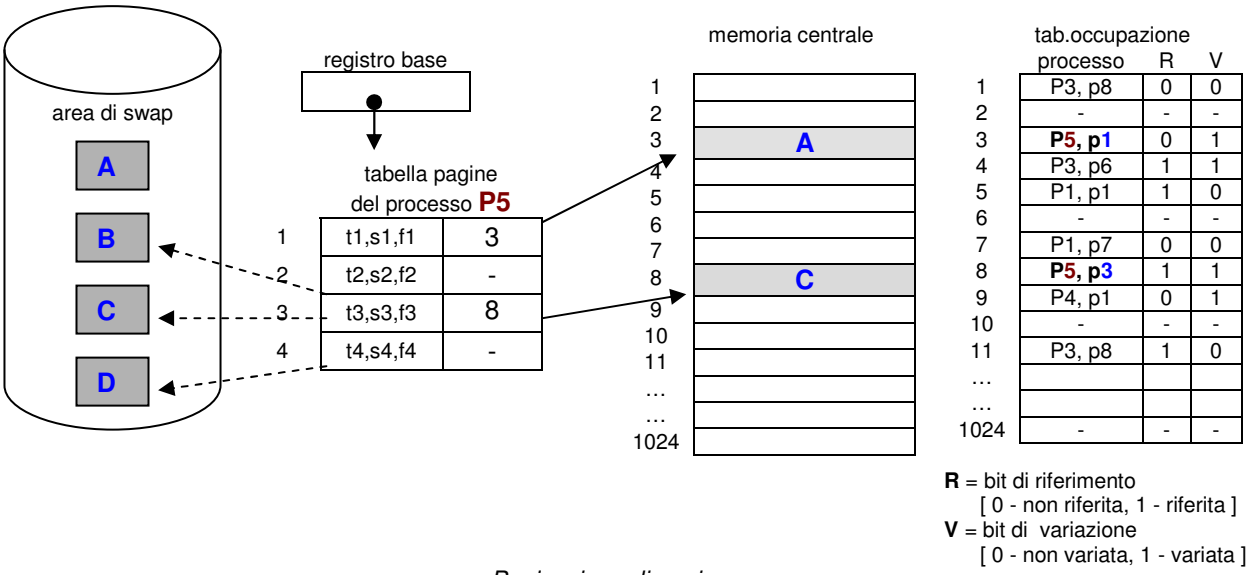
Memoria Virtuale

Overlay (memoria virtuale)

Un problema che ha sempre afflitto i programmatori della "notte dei tempi" è stata la quantità di memoria fisica. Era molto frequente infatti che un programma fosse troppo grande per essere caricato in RAM e quindi, la soluzione generalmente adottata, era quella di dividerlo in parti più piccole dette **overlay**. L'esecuzione iniziava dunque dall'overlay 0 per poi proseguire con i successivi fino al termine dell'esecuzione. Gli overlay venivano continuamente caricati e scaricati dalla memoria. Una possibile suddivisione è di avere il programma principale sempre in memoria e di strutturare ogni sottoprogramma come overlay. Il compito di suddividere il programma è a carico del programmatore. Solo nel 1972 fu realizzato il primo sistema operativo, per l'IBM 370, che gestiva autonomamente e automaticamente la memoria virtuale, svincolando gli sviluppatori da tale onere.

Paginazione Dinamica (memoria virtuale)

Questa tecnica permette di eseguire un programma anche se non tutte le sue pagine sono presenti in memoria; le altre pagine restano nell'area di swap e vengono caricate all'occorrenza.
Quando il programma fa riferimento ad una pagina non presente in memoria (***page fault***) il sistema provvede a caricarla (***page swapping***) e il processo viene bloccato finché la pagina non è stata caricata.
La nuova pagina sarà caricata su una "sede pagina" libera - se presente- o su una occupata; in quest'ultimo caso potrebbe prima essere necessario salvare la pagina da sovrascrivere su disco (se, ad esempio, questa ha subito modifiche dopo l'ultimo caricamento).
La **rilocazione** è **Dinamica** e la memoria centrale viene gestita con l'ausilio di una **tabella di occupazione pagine**, (di tante righe quante sono le "sedi pagina") contenente, per ogni riga, l'indicazione di chi occupa la "sede pagina" corrispondente [PID e numero pagina del processo che la occupa] o 0 se libera, un bit di *variazione* (che dice se la pagina è stata modificata dopo) e un bit di *riferimento* (che dice se la pagina è stata usata - riferita – da poco tempo).
Inoltre, per ogni processo, si usa una **tabella delle pagine**, (di tante righe quante sono le pagine del processo e avente, per ogni riga, il numero della sede pagina nella quale è stata caricata la relativa pagina del programma - oppure 0 se non presente in memoria – e l'indirizzo del blocco sull'area di swap del disco che contiene l'immagine della pagina) il cui indirizzo è contenuto nel **registro base**.



Paginazione dinamica

La scelta della pagina da sovrascrivere, quando tutte le sedi pagina sono occupate, avviene tenedo conto del bit di riferimento e del bit di variazione. L'algoritmo di sostituzione generalmente utilizzato è l'**LRU** (*Last Recently Used*) che si basa sul principio di 'località globale': se una pagina è stata usata di recente, c'è un'alta probabilità che venga nuovamente utilizzata. Quindi, quando occorre sovrascrivere una pagina, si sceglie una pagina non usata di recente. Se la pagina scelta ha subito variazioni, prima di sovrascriverla, occorre salvarla su disco (per non perdere le modifiche). Occorre inoltre aggiornare opportunamente tutte le tabelle interessate (tabella di occupazione delle pagine e tabella delle pagine dei processi coinvolti).
Le tecniche di **Segmentazione dinamica** e di **Segmentazione e Paginazione dinamica** sono simili a quella della paginazione dinamica, con le opportune variazioni.
Windows e Linux, per la gestione della memoria, utilizzano la paginazione dinamica.

Gestione delle periferiche

Occorre assegnare i dispositivi ai processi secondo un'opportuna politica di scheduling, controllare l'esecuzione delle operazioni sui dispositivi e fornire un'interfaccia uniforme agli stessi.
Le operazioni di I/O vengono svolte da processori specializzati - processori di I/O - che interagiscono con il sistema attraverso buffer di comunicazione e segnali di interruzione.
Le periferiche sono viste in modo "astratto", indipendentemente dalla loro struttura HW. Tale astrazione viene realizzata dai **driver**, dei programmi che 'pilotano' la periferica e che fanno da interfaccia tra la periferica e i programmi che vi possono accedere richiamando il driver relativo.

Gestione del disco

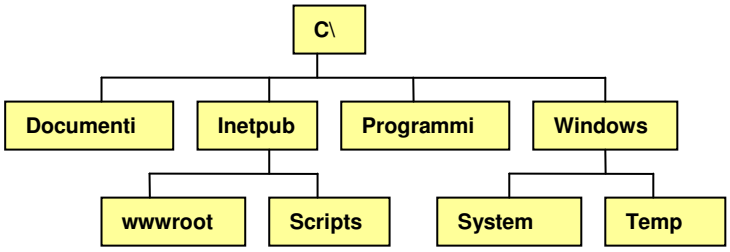
Il disco fisso (HD) viene usato in modo condiviso da più processi, ma l'accesso ad esso è molto lento rispetto alla velocità del processore. La tecnica di scheduling del disco, che deve minimizzare il tempo di accesso, può essere:

- **FCFS** (*First Come First Served*), in base all'ordine di arrivo delle richieste
- **SSTF** (*Shortest Seek Time First*), serve la richiesta che si riferisce alla traccia più vicina
- **SCAN** o dell'**ascensore**, evoluzione della SSTF, serve la richiesta che si riferisce alla traccia più vicina ma nella direzione della testina, che si muove 'avanti' e 'indietro'

File System

Il File System è il modulo del SO che si occupa della gestione delle informazioni memorizzate su disco: identifica e cataloga i file presenti, gestisce le operazioni sui file, assegna i file ai processi che lo richiedono, stabilisce i meccanismi di protezione, gestisce l'allocazione delle aree disco.
In definitiva esso realizza l'indipendenza dai dispositivi fisici facendo corrispondere ad un modello logico - File System Logico - una struttura fisica - File System Fisico - nascosto all'utente.

File System Logico



I genere il FS prevede una **struttura gerarchica** ad albero che parte dalla radice principale [C:] e ha i file organizzati in directory (o cartelle) gerarchiche, ciascuna delle quali può contenere altri file ed altre directory. La directory stessa è un file speciale costituito da un record descrittivo per ogni file o directory in essa "contenuti". Il FS fornisce le

operazioni di gestione - creazione, cancellazione, … - di file e directory. Si occupa dell'apertura dei file, della loro protezione, della condivisione di file e directory gestendo gli accessi contemporanei per evitare incongruenze, delle organizzazioni dei file e dei metodi di accesso (sequenziale, diretto, …), del recovery (ad esempio "salva/ ripristina configurazione di sistema" di Windows), etc..

File System Fisico

I file vengono memorizzati su disco come sequenze di blocchi o record fisici (un blocco è l'intersezione di faccia, traccia e settore). Ogni dispositivo ha associato un proprio descrittore del sistema di archiviazione (etichetta del disco, inizio tabella e mappa dei descrittori, inizio FAT o mappa dei blocchi, …), la tabella dei descrittori dei file, la FAT o la mappa dei blocchi.

Per l'allocazione dello spazio disco per i file esistono diverse tecniche:

- **Gestione contigua** (ogni file viene memorizzato in una parte contigua del disco; esiste una mappa delle aree libere del disco)
- **Blocchi concatenati** (i blocchi dei file sono collegati tra loro da una lista, in ogni blocco è presente il puntatore al successivo; l'accesso può essere solo sequenziale; i blocchi liberi sono legati a lista – "lista libera")
- **Tabella di puntatori – FAT** (utilizza un vettore di tanti elementi quanti sono i blocchi del disco, ogni elemento contiene la posizione dell'elemento/blocco successivo o 0 se è l'ultimo o -1 se il relativo blocco è vuoto; il puntatore al primo blocco del file si trova nel descrittore del file presente nella tabella dei descrittori; Windows usa questa tecnica)
- **Mappa dei blocchi - Block map** (ogni file ha associata una tabella di puntatori ai suoi blocchi; tale tabella è memorizzata nel descrittore del file e viene caricata in memoria all'apertura del file; è la tecnica che usa Linux)

L'elenco dei file presenti su disco è mantenuto in un descrittore di file o **indirizzario** o **directory**. Per ogni file sono specificati nome, tipo, lunghezza, informazioni per l'accesso (indirizzo iniziale o mappa dei blocchi), tipo di protezione, data di creazione/modifica. L'indirizzo iniziale del file è un elemento della **FAT** (associato al relativo blocco) che a sua volta punta all'elemento successivo (blocco), fino all'ultimo che ha valore "0". I blocchi liberi sono contrassegnati dal valore "– 1".

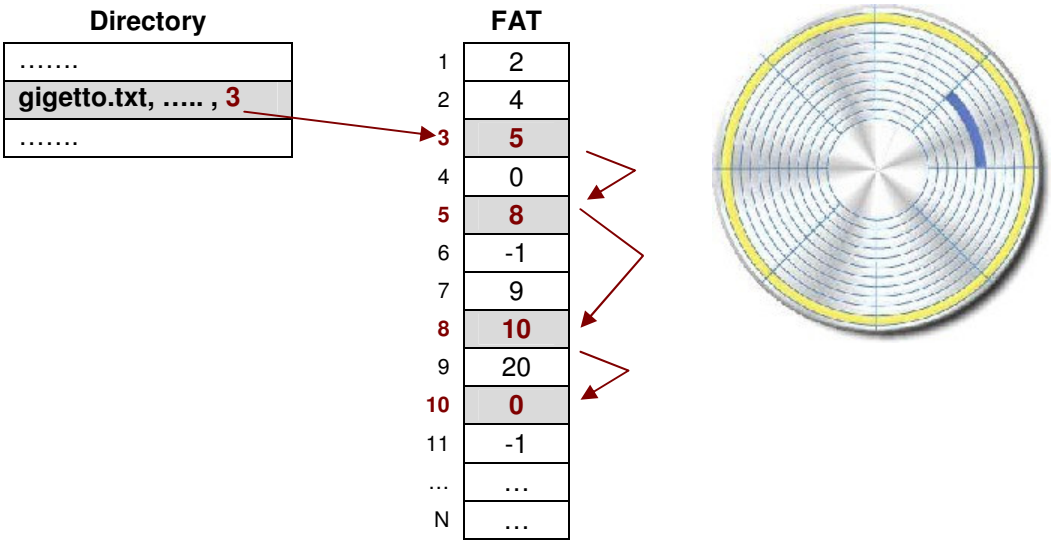


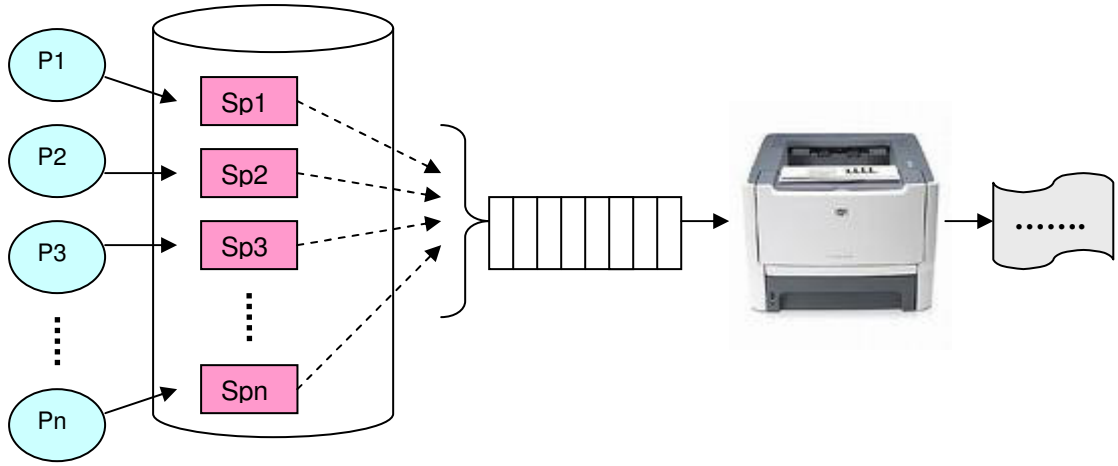
Tabella dei puntatori - FAT (il file "gigetto.txt" occupa i blocchi 3, 5, 8 e 10 del disco)

Esempio di FAT (Floppy Disk)

Virtualizzazione delle periferiche (SPOOL)

I dispositivi **virtuali** sono dispositivi fisicamente "non esistenti" simulati con l'ausilio di altri dispositivi fisici, come la Memoria Virtuale precedentemente esaminata.
Altro esempio di virtualizzazione delle periferiche è lo **SPOOL** (*Symultaneous Peripheral Operations On Line*), una tecnica che consente la gestione multipla di periferiche seriali (che possono essere concesse ad un processo alla volta) quali la stampante.

Le stampanti virtuali vengono simulate con l'ausilio del disco e dell'unica stampante di sistema presente. Ad ogni processo che usa la stampante viene associato, su disco, un **file di spool** nel quale vengono memorizzate le informazioni da stampare. Quando il processo dichiara che la stampa è terminata (o quando il processo termina), il relativo file di spool viene chiuso ed inviato alla stampante. Se la stampante è libera esso viene stampato, altrimenti la sua richiesta di stampa viene accodata nella coda di stampa ad attendere che la stampante abbia stampato i file di spool le cui richieste di stampa sono in coda prima di essa.



SPOOL - virtualizzazione della stampante di sistema