

Un Sistema Operativo controlla l'uso delle risorse di un sistema di un computer come CPU, memoria e dispositivi di I/O per soddisfare le richieste di elaborazione dei suoi utenti.

Gli obiettivi principali di un SO sono:

- Convenienza per l'utente: gli utenti si aspettano qualità dai servizi offerti
- Uso efficiente: gli amministratori si aspettano che il SO sfrutti al meglio le risorse hardware a disposizione in modo da avere buone prestazioni nell'esecuzione dei programmi
- Assenza di interferenze: gli utenti si aspettano la garanzia che altri utenti non siano in grado di interferire con le proprie attività.
- Gestione dei programmi: il SO inizializza i programmi, organizza l'uso della CPU, e li termina quando hanno completato la loro esecuzione. Visto che vengono eseguiti più programmi in contemporanea, il SO esegue una funzione chiamata scheduling che serve per gestire l'esecuzione dei programmi.
- Gestione delle risorse: il SO alloca le risorse come la memoria e i dispositivi di I/O quando un programma li richiede. Al termine dell'esecuzione del programma, il SO dealloca queste risorse per assegnarle ad altri programmi.
- Sicurezza e protezione: il SO non dovrebbe permettere a nessun utente di usare in modo illegale programmi o risorse, o di interferire con il loro funzionamento.

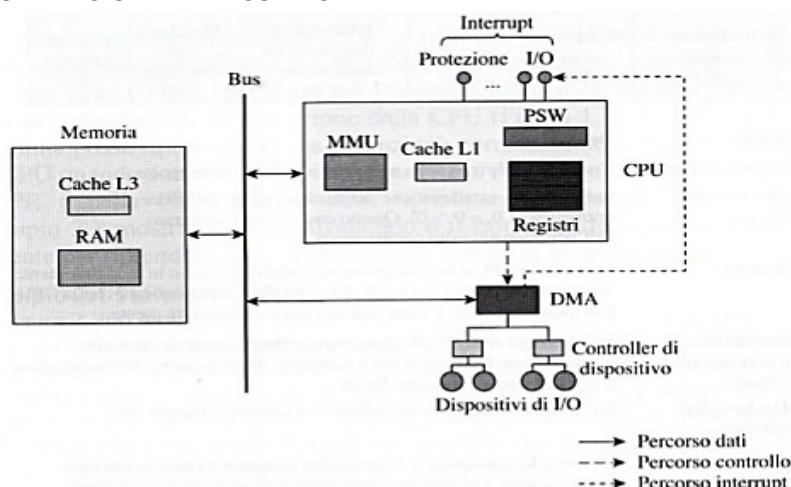
Un progettista di SO ha una sua visione astratta del sistema operativo stesso:

- Interfaccia utente: questa accetta comandi per eseguire programmi e usa le risorse e i servizi forniti dal sistema operativo. Può essere un'interfaccia a linea di comando che mostra all'utente un prompt dei comandi oppure può essere un'interfaccia grafica (GUI).
- Routine non di sistema: queste implementano i comandi utente relativi all'esecuzione dei programmi e all'uso delle risorse del computer; sono richiamate dall'interfaccia utente.
- Kernel: questo è il cuore del SO. Controlla il computer e fornisce un insieme di funzioni e servizi per utilizzare la CPU, la memoria e le altre risorse del computer. Le funzioni e i servizi del kernel sono richiamati dalle routine non-kernel e dai programmi utente.

LE ROUTINE CHE ESEGUE COMUNEMENTE IL S.O. SONO QUESTE:

Task	Eseguito
Costruire una lista di risorse	Durante la fase di boot
Memorizzare le informazioni per la sicurezza	Mentre vengono registrati nuovi utenti
Verificare l'identità di un utente	Al momento del login
Inizializzare l'esecuzione dei programmi	Quando richiesto dall'utente
Memorizzare le informazioni per l'autorizzazione	Quando un utente specifica che i suoi collaboratori possono avere accesso a i suoi programmi o ai suoi dati
Eseguire l'allocazione delle risorse	Quando richiesto dagli utenti o dai programmi
Preservare lo stato corrente delle risorse	Durante l'allocazione e la deallocazione delle risorse
Preservare lo stato corrente dei programmi ed effettuare lo scheduling	In maniera continuativa durante l'esecuzione del SO

ESEMPIO GRAFICO DI COME IL S.O. VEDE IL COMPUTER:



LO STATO DELLA CPU (CAMPI IMPORTANTI DEL PSW)

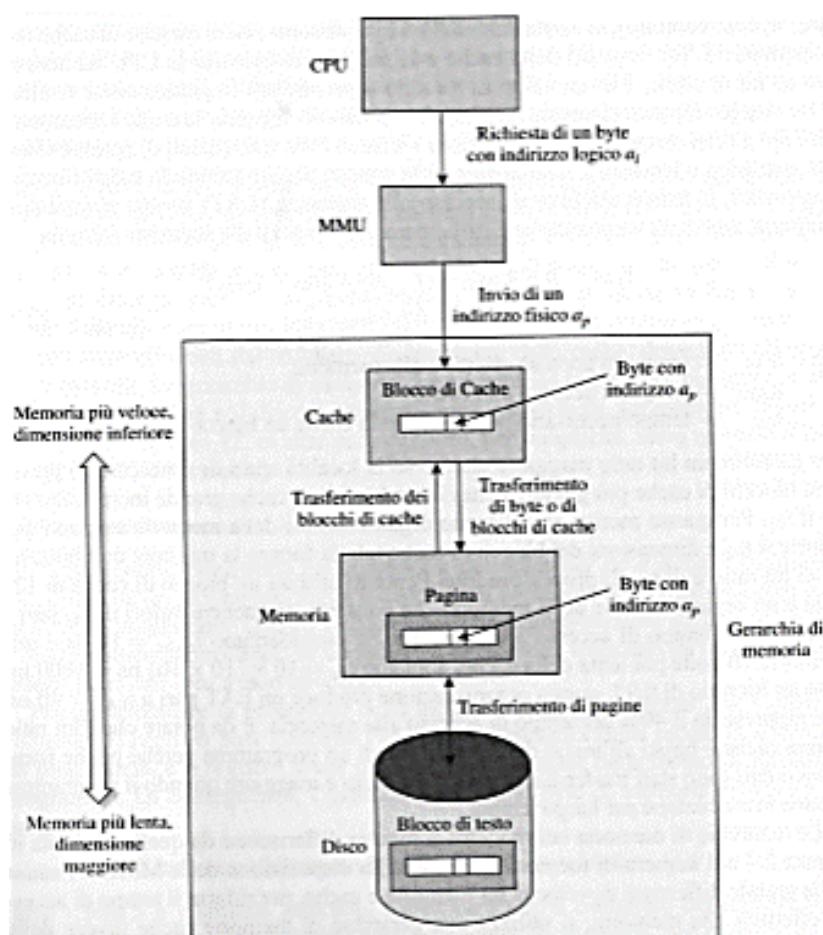
Program counter (PC)	Condition code (CC)	Modalità (M)	Informazioni di protezione della memoria (MPI)	Maschera degli interrupt (IM)	Codice interrupt (IC)
----------------------	---------------------	--------------	--	-------------------------------	-----------------------

Campo	Descrizione
Program counter	Contiene l'indirizzo della prossima istruzione da eseguire.
Condition code (flag)	Indica alcune caratteristiche del risultato di un'istruzione aritmetica (<0 , $=0$, >0). Questo codice viene utilizzato nell'istruzione di salto condizionato.
Modalità	Indica se la CPU è in esecuzione in modalità kernel o in modalità utente. Si assume un campo di un solo bit con valore 0 per indicare che la CPU è in modalità kernel e 1 per indicare che è in modalità utente.
Informazioni di protezione della memoria	Informazioni relative alla protezione della memoria del processo in esecuzione. Questo campo è composto di sottocampi che contengono il registro base e il registro limite.
Maschera degli interrupt	Indica quali interrupt sono abilitati e quali sono "mascherati".
Codice interrupt	Describe la condizione o l'evento che ha causato l'ultimo interrupt. Questo codice è utilizzato da una routine di servizio dell'interrupt.

Sia i registri GPR, General Purpose Register (visibile agli utenti) che la PSW, Program Status Word (controllato dalla CPU) contengono tutte le informazioni necessarie per conoscere cosa sta facendo la CPU. Queste informazioni costituiscono lo stato della CPU.

Memory Management Unit (MMU) si occupa di effettuare la traduzione degli indirizzi logici a fisici:

- Indirizzo Logico è l'indirizzo usato dalla CPU per fare riferimento ad un dato o ad un'istruzione
- In indirizzo Fisico è l'indirizzo in memoria dove risiede il dato o l'istruzione richiesta dalla CPU.



La CPU accede solo alla memoria più veloce: se il dato (o l'istruzione) di cui necessita è presente nella memoria più veloce allora viene usato direttamente, altrimenti il dato richiesto viene copiato dalla memoria più lenta in quella più veloce e successivamente utilizzato. Il dato rimane nella memoria più veloce finché non viene rimosso per fare spazio ad altri dati. Questa organizzazione ha, quindi, lo scopo di velocizzare gli accessi ai dati.

La memoria cache contiene alcune istruzioni e valori di dati cui la CPU ha avuto accesso più di recente. L'hardware della memoria carica sempre un blocco di memoria di dimensioni standard in un'area della cache chiamata cache block.

In questo modo, l'accesso a un byte vicino a un byte caricato di recente, può essere effettuato senza accedere nuovamente alla memoria.

Sono diversi gli schemi usati per scrivere un byte in memoria, un esempio, il write-through prevede la scrittura del byte nella cache e nella memoria allo stesso tempo. Per ogni dato o istruzione richiesti la CPU effettua una ricerca nella cache:

- Si verifica un hit se i byte richiesti sono presenti nella cache.
- Si verifica una miss se i byte richiesti non sono presenti, e in questo caso devono essere caricati in memoria i byte richiesti andandoli a cercare nella memoria nel livello precedente e portandoli al livello successivo.

Esistono diversi tipi di cache: L1 che si trova sulla CPU affiancata a sua volta da un'altra L2, si può trovare anche una L3 ma è esterna ad essa e solitamente controllata dalla MMU.

La protezione della memoria (è il campo del PSW relativo all'informazione di protezione della memoria MPI che contiene i registri base e limite) rende necessario prevenire che un programma legga o cancelli il contenuto della memoria utilizzata da un altro programma.

I campi costituenti la MPI sono:

- Un registro base, che contiene l'indirizzo di partenza della memoria allocata a un programma.
- Un registro limite che contiene la dimensione della memoria allocata al programma.

Si calcola l'ultimo byte disponibile con la formula:

$$\text{Indirizzo dell'ultimo byte} = <\text{valore registro base programma X}> + <\text{valore registro limite programma X}> - 1.$$

Quindi la CPU controlla tramite questi due registri che non si vada ad eccedere la memoria riservata ad altri programmi. Se dovesse succedere l'hardware genera un interrupt per segnalare una violazione della memoria.

I modi per effettuare le operazioni di I/O chiamate da un processo sono i seguenti:

Modalità di I/O	Descrizione
I/O programmato	Il trasferimento dati tra la periferica di I/O e la memoria avviene attraverso la CPU. La CPU non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O.
Interrupt di I/O	La CPU è libera di eseguire altre istruzioni dopo aver eseguito l'istruzione di I/O. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte. Questa sequenza di operazioni viene ripetuta finché tutti i byte sono trasferiti.
I/O basato sul direct memory access (DMA)	Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus. La CPU non è coinvolta nel trasferimento dei dati. Il controller DMA genera un interrupt quando il trasferimento di tutti i byte è stato completato.

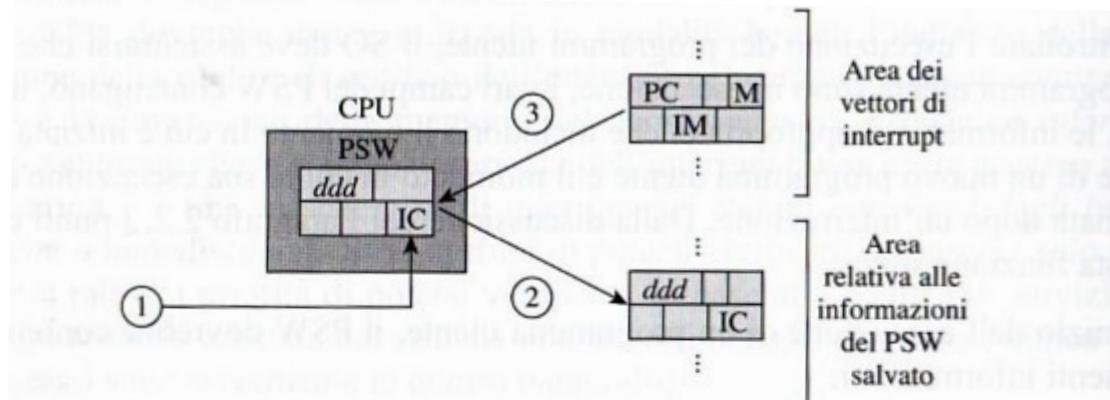
- I/O Programmato: operazione eseguita attraverso la CPU, risulta essere inefficiente.
- I/O Interrupt: solo quando i dati sono pronti al trasferimento, allora chiama la CPU, meglio del precedente.
- I/O DMA: consulta il device DMA, Direct Memory Access, dedicato solo all'esecuzione delle operazioni di I/O.

Un evento è una situazione che richiede l'attenzione del sistema operativo. Ad ogni evento è associato un interrupt il cui scopo è quello di segnalare al SO il verificarsi dell'evento a cui sono associate azioni appropriate per gestirlo. Esistono diverse "Classi" di interrupt:

- I/O interrupt: dovuto alla gestione di operazioni I/O.
- Timer Interrupt: causato da un timer
 - o Che segnala un errore, tipo scaduta connessione
 - o O programmato, come la chiamata sleep().
- Program Interrupt: causato dai programmi:

- Tramite errori nel codice. (involontario)
- Tramite System Call. (volontario)

Quando si verifica un interrupt di una certa classe di eventi, l'hardware imposta un codice interrupt nel campo IC del PSW per indicare quale specifico interrupt. Il campo maschera degli interrupt IM del PSW indica quali interrupt sono consentiti in un dato momento. Dopo l'esecuzione di ogni istruzione, la CPU controlla se sono stati generati interrupt. In caso affermativo, la CPU esegue il gestore dell'interrupt (interrupt handler):



Passo	Descrizione
1. Impostare il codice interrupt	L'hardware per la gestione dell'interrupt crea un codice che descrive la causa dell'interrupt. Questo codice viene memorizzato nel campo interrupt code (IC) del PSW.
2. Salvare il PSW	Il PSW viene copiato nell'area relativa alle informazioni del PSW salvato. In alcuni computer, questa azione salva anche i registri general purpose.
3. Caricamento del vettore di interrupt	Si accede al vettore di interrupt corrispondente alla classe di interrupt. L'informazione contenuta nel vettore di interrupt viene caricata nel campo corrispondente del PSW. Questa azione commuta la CPU per l'esecuzione dell'appropriata routine di servizio dell'interrupt del kernel.

Con il termine interrupt annidati si intende il verificarsi di interrupt contemporaneamente. Si risolve con il kernel che imposta la maschera degli interrupt in ogni vettore degli interrupt per mascherare solo gli interrupt con priorità più bassa in modo tale da poter servire gli interrupt più critici in maniera annidata con l'aggiunta di uno schema di sincronizzazione per assicurare che solo una routine di gestione degli interrupt possa accedere ai dati in ogni istante. Tali kernel sono chiamati kernel prelazionabili.

Una chiamata di sistema (in inglese system call) è il meccanismo usato da un programma a livello utente per richiedere un servizio a livello kernel del sistema operativo. Una system call è implementata attraverso il metodo degli interrupt, dunque la si può definire anche come una richiesta che un programma fa al kernel attraverso un interrupt software.

Come si instaura un programma/processo nella CPU:

1. All'inizio dell'esecuzione di un programma utente, il PSW dovrebbe contenere le seguenti informazioni:
 - a. il campo program counter (PC) dovrebbe contenere l'indirizzo della prima istruzione del programma;
 - b. il campo modalità (M) dovrebbe contenere un 1, cioè che la CPU opera in modalità utente;
 - c. il campo informazioni di protezione della memoria (MPI) dovrebbe contenere informazioni circa l'indirizzo base e la dimensione dell'area di memoria dedicata al programma;
 - d. il campo maschera degli interrupt (IM) dovrebbe essere impostato in modo da abilitare tutti gli interrupt.
2. Quando l'esecuzione di un programma utente viene interrotta, lo stato della CPU (contenuto del PSW e dei GPR) dovrebbe essere salvato.

3. Quando deve essere ripristinata l'esecuzione del programma interrotto, lo stato salvato della CPU dovrebbe essere caricato nel PSW e nei GPR.

Un ambiente di elaborazione si compone di un computer, delle sue interfacce con altri sistemi e dei servizi forniti dal suo sistema operativo agli utenti e ai loro programmi. Nel corso della storia, sia gli ambienti di elaborazione che i sistemi operativi si sono evoluti.

Riguardando gli ambienti di elaborazione (a partire dagli anni 60'):

1. AMBIENTI DI ELABORAZIONE NON INTERATTIVI

L'elaborazione viene effettuata dal SO e i risultati vengono restituiti all'utente, che non può interagire con l'elaborazione. L'obiettivo del SO è l'uso efficiente delle risorse. Le elaborazioni utilizzate negli ambienti non interattivi sono programmi o job dove:

- un programma è un insieme di funzioni o moduli.
- un job è una sequenza di programmi che insieme raggiungono l'obiettivo desiderato;
- un job viene eseguito solo se i programmi precedenti del job sono stati eseguiti con successo.

2. AMBIENTI DI ELABORAZIONE INTERATTIVI

In questo tipo l'utente può interagire con l'elaborazione mentre è in esecuzione. L'obiettivo del SO è di ridurre il tempo medio richiesto per implementare l'interazione tra un utente e la sua elaborazione. Non ha senso parlare di job in quanto è l'utente a scegliere cosa avviare.

3. AMBIENTI REAL-TIME CON TIME-SHARING

Il SO deve utilizzare tecniche speciali per assicurare che le elaborazioni siano completate rispettando dei vincoli temporali.

4. AMBIENTI DISTRIBUITI ED EMBEDDED

Dipendentemente dalle richieste e l'uso di determinati ambienti scegiamo:

- In un ambiente distribuito le risorse presenti possono essere utilizzate da diversi sistemi attraverso una rete.
- In un ambiente embedded il computer è parte di uno specifico sistema hardware, come un elettrodomestico ed esegue elaborazioni volte a controllare il sistema stesso.

5. AMBIENTI DI ELABORAZIONE MODERNI

Nei moderni ambienti di elaborazione c'è la necessità di supportare diverse applicazioni, dunque il SO deve adottare complesse strategie per gestire i programmi e le risorse.

Riguardando i sistemi operativi (a partire dagli anni 60'):

1. SISTEMI DI ELABORAZIONE BATCH

L'obiettivo principale è l'uso efficiente della CPU.

Questo sistema opera elaborando un job alla volta, eseguendo i programmi uno dopo l'altro. In questo modo un solo programma è in esecuzione in un dato momento. Un batch è una sequenza di job utente assemblati per essere elaborati dal sistema operativo. Un operatore assemblava il batch organizzando alcuni job utente in sequenza, delimitando l'inizio e la fine del batch con schede speciali. Il SO eseguiva uno dopo l'altro i vari job del batch. In questo modo l'operatore doveva intervenire solo all'inizio e alla fine del batch. I sistemi operativi per l'elaborazione batch si focalizzano sul processo di automatizzazione di una collezione di programmi, in modo da ridurre i tempi idle della CPU.

2. SISTEMI MULTIPROGRAMMATI

L'obiettivo è sia l'uso efficiente della CPU che dei dispositivi di I/O.

Questo sistema gestisce diversi programmi in uno stato di parziale completamento per ogni istante di tempo ed attraverso le priorità del programma concede o meno l'utilizzo della CPU ai vari programmi. Poiché diversi programmi sono in memoria contemporaneamente, le istruzioni, i dati e le operazioni di I/O di un programma dovrebbero essere protette dall'interferenza di altri programmi. In questa fase appare il DMA, il concetto di protezione della memoria e modalità utente/kernel della CPU. Il SO mantiene un certo numero di programmi in memoria questo numero è detto grado di multiprogrammazione. I processi in memoria si differenziano a seconda del tipo di richiesta, ossia CPU-bound e I/O-bound(che non sfrutta la CPU e fa solo operazioni I/O). E per ottimizzare operazioni di multiprogrammazione bisogna utilizzare un mix appropriato di questi programmi. La priorità è quel criterio mediante il quale lo scheduler decide quale richiesta debba

essere schedulata quando molte richieste sono in attesa di essere servite (la maggiore l'hanno i programmi I/O-bound).

3. SISTEMI TIME-SHARING

L'obiettivo principale è l'ottimizzazione della velocità di risposta alle richieste fatte dai processi.

Questo obiettivo è raggiunto dando un'equa opportunità di esecuzione a ogni processo attraverso due metodi: il SO serve tutti i processi a turno (scheduling round-robin) ed evita che un processo utilizzi per troppo tempo la CPU (time-slicing). L'obiettivo è ottenuto condividendo il tempo di CPU tra i processi in modo tale che ogni processo che ha fatto richiesta ottenga l'uso della CPU senza attendere troppo. Questo obiettivo è raggiunto utilizzando lo scheduling round-robin con time-slicing.

4. SISTEMI REAL-TIME

L'obiettivo principale è il supporto di applicazioni real-time.

Cioè quelle applicazioni dove è necessario ottenere una risposta dal SO in un tempo prefissato. Non è importante l'intervallo di tempo in cui il SO deve reagire; l'importante è che risponda entro un tempo massimo predeterminato. In altre parole, il sistema deve essere prevedibile. Sono stati sviluppati due tipi di sistemi real-time:

- Un sistema hard real-time è solitamente dedicato all'elaborazione di applicazioni real-time e richiede che i suoi vincoli temporali siano rispettati in maniera garantita.
- Un sistema soft real-time può tollerare che in modo occasionale non vengano soddisfatti i suoi vincoli temporali.

Un SO real-time usa due tecniche:

- la fault tolerance, indice massimo di tempo affinché un processo termini la sua esecuzione.
- la graceful degradation indice che rappresentano l'affidabilità di un processo.

5. SISTEMI DISTRIBUITI

L'obiettivo principale è quello di consentire ad un utente di avere accesso alle risorse di altri computer. Sempre in modo conveniente ed efficace. Per migliorare la convenienza, il SO non richiede all'utente di conoscere dove si trovano le risorse (trasparenza); per migliorare l'efficienza, il SO può eseguire parti di un'elaborazione su computer differenti allo stesso tempo. Il SO esegue le sue funzioni di controllo in diversi computer tra quelli collegati alla rete. Ciò consente l'uso efficiente delle risorse di tutti i computer consentendo ai programmi di condividerle attraverso la rete, l'aumento della velocità di esecuzione di un programma eseguendo le sue parti su differenti computer allo stesso tempo e di fornire affidabilità attraverso la ridondanza delle risorse e dei servizi.

6. SISTEMI OPERATIVI MODERNI

In pratica un moderno SO controlla un diverso ambiente di elaborazione che ha elementi di tutti i classici ambienti di elaborazione visti fin qui (batch, time-sharing, real-time, distribuiti), e deve pertanto utilizzare tecniche differenti per differenti applicazioni. Utilizza una strategia adattiva che seleziona le tecniche più appropriate per ogni applicazione in base alla sua natura.

Nei moderni SO si parla di:

- Portabilità: si riferisce alla facilità con cui il SO può essere implementato su un computer che ha una differente architettura. Il problema della portabilità del SO è affrontato separando le parti dipendenti dall'architettura da quelle indipendenti. Infatti, ci sarebbe un'elevata portabilità se il codice di SO dipendente dall'architettura del sistema fosse di dimensione ridotta.
- Espandibilità: si riferisce alla facilità con cui le sue funzionalità possono essere migliorate per adattarle a un nuovo ambiente di elaborazione. L'espandibilità di un SO è necessaria per incorporare nuovo hardware in un computer e per fornire nuove funzionalità in risposta a nuove aspettative degli utenti.

In generale Il Sistema Operativo esegue principalmente delle operazioni per conciliare questi 2 aspetti:

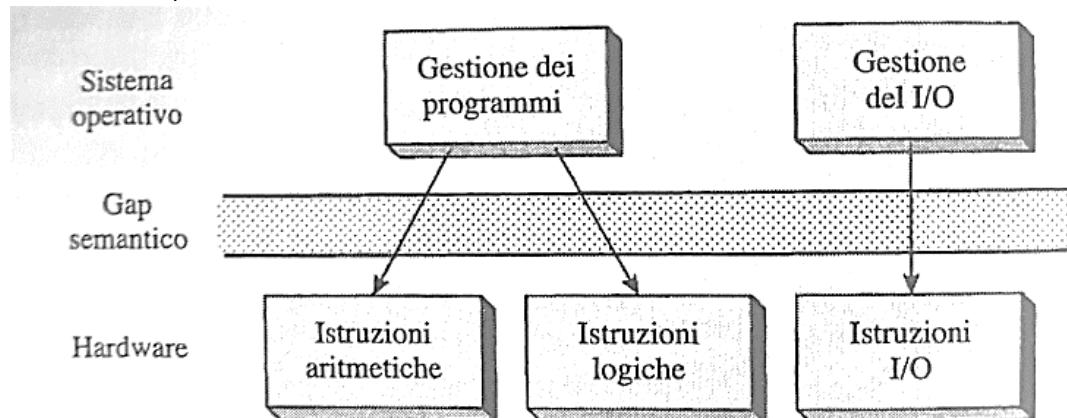
1. All'accensione di un computer, la procedura di boot analizza la sua configurazione: tipo di CPU, quantità di RAM, dispositivi di I/O e altri dettagli dell'hardware. Successivamente carica una parte del SO in memoria, inizializza le sue strutture dati con le informazioni ottenute e gli passa il controllo del sistema.
2. Le funzioni di un SO sono implementate da gestori di eventi e sono attivate dalle procedure di servizio degli interrupt. Queste funzioni riguardano principalmente la gestione dei processi, la gestione della memoria, la gestione dell'I/O, la gestione dei file e l'implementazione della sicurezza e protezione.

3. Nel determinare come un SO debba svolgere una sua funzione si devono considerare la politica e il meccanismo dello stesso. In pratica, la politica decide cosa dovrebbe essere fatto, mentre un meccanismo determina come dovrebbe essere fatto (e in effetti lo fa). Dunque, la politica decide quale modulo va richiamato e in quali circostanze. Il meccanismo è implementato nel modulo ed esegue un'azione specifica.

Si evidenziano come i SO nel corso della storia si sono evoluti nei termini di portabilità ed espandibilità:

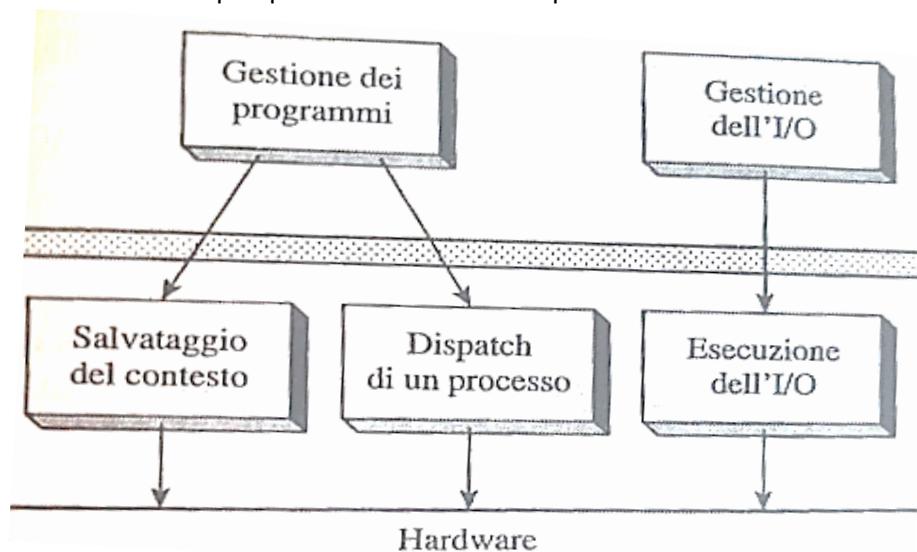
1. SISTEMI OPERATIVI CON STRUTTURA MONOLITICA

Secondo cui il SO formava un singolo strato software tra l'utente e la macchina (hardware). L'interfaccia utente consisteva in un interprete dei comandi. Questo tipo di sistemi aveva una portabilità molto limitata poiché il codice dipendente dall'architettura era presente in gran parte nel SO. Questo rendeva complicate e dispendiose (anche in termini economici) le fasi di test e debug a causa del gap semantico (l'assenza di corrispondenza tra la natura delle operazioni necessarie all'applicazione e la natura delle operazioni fornite dall'hardware).



2. SISTEMI OPERATIVI STRUTTURATI A LIVELLI

Questa progettazione vede il SO come una gerarchia di livelli, in cui ogni livello fornisce un insieme di servizi al livello superiore ed esso stesso usa i servizi messi a disposizione dal livello inferiore. La progettazione a livelli causa diversi problemi. Il primo è legato al numero di livelli e al fatto che ognuno di essi può "comunicare" solo coi livelli adiacenti, quindi una azione richiesta da un processo utente deve "scalare" i vari livelli partendo da quello in cima fino ad arrivare a quello in fondo. L'ultimo problema riguarda la stratificazione delle funzionalità del SO. Questa si verifica poiché ogni funzionalità deve essere divisa in parti che appartengano a differenti livelli del SO. Inoltre, la stratificazione crea problemi anche per l'inserimento di nuove funzioni nel SO che può portare alla modifica di più livelli.



3. SISTEMI OPERATIVI BASATI SU KERNEL

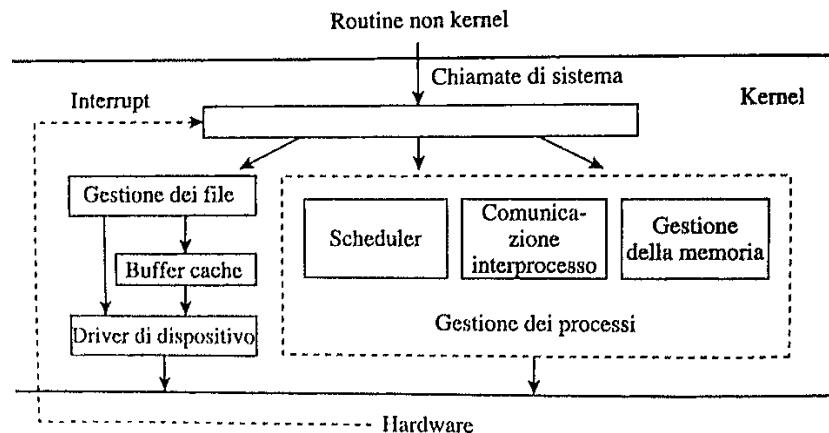
Il kernel è il cuore di un SO e fornisce un insieme di istruzioni e servizi per supportare differenti funzioni. Il resto del SO è organizzato come un insieme di routine non kernel, che implementano operazioni sui processi e risorse di interesse per l'utente, e un'interfaccia grafica. Il funzionamento del kernel è guidato dagli

interrupt, di fatti prende il controllo quando un interrupt gli notifica l'occorrenza di un evento o quando un interrupt viene generato per servire una system call. La portabilità si ottiene inserendo nel kernel le parti del codice del SO dipendenti dall'architettura, mantenendo al di fuori del kernel le parti di codice indipendenti dall'architettura. I SO basati su kernel presentano una ridotta espandibilità poiché l'aggiunta di nuove funzionalità può richiedere cambiamenti nelle funzioni e nei servizi offerti dal kernel.

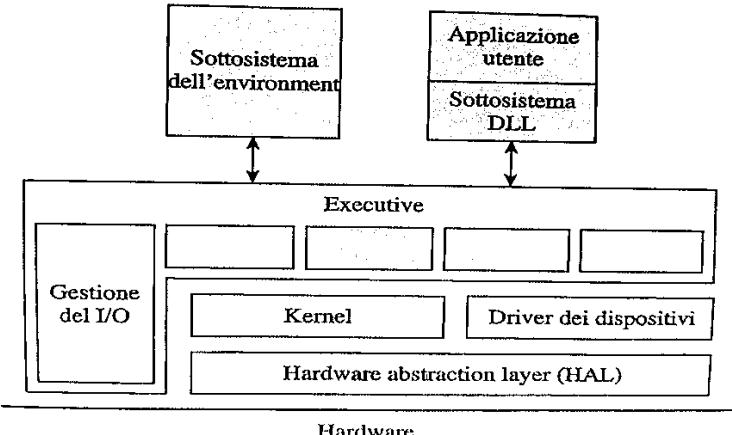
4. SISTEMI OPERATIVI BASATI SU MICRKERNEL

Tuttavia, in pratica, anche i kernel contengono del codice indipendente dall'architettura. Questo fa sì che la dimensione del kernel sia elevata allontanando l'obiettivo della portabilità. Inoltre, per incorporare nuove funzionalità spesso è necessario effettuare modifiche al kernel, e ciò porta a poca espandibilità.

ESEMPIO GRAFICO KERNEL DEL SISTEMA OPERATIVO UNIX



ESEMPIO GRAFICO KERNEL DEL SISTEMA OPERATIVO WINDOWS



Un programma consiste di un insieme di funzioni.

Le funzioni di un programma possono essere processi separati.

Quindi un processo è un programma in esecuzione che utilizza un insieme di risorse.

Un'applicazione può essere sviluppata per avere molti processi che operano concorrentemente e interagiscono tra di loro per ottenere un risultato comune. È il kernel che alloca le risorse ai processi e li schedula per l'utilizzo della CPU. Gestire i processi significa:

1. Crearli,
2. Soddisfare le richieste di risorse,
3. Schedularli per l'uso della CPU,
4. Sincronizzarli per controllare la loro interazione,
5. Evitare deadlock in modo che non siano in attesa l'un l'altro indefinitamente,
6. Terminarli quando hanno concluso l'esecuzione.

In Unix esistono tre tipi di processi:

- Un processo utente
esegue operazioni per l'utente e dipende dal terminale di controllo. Quando un utente avvia un programma, il kernel crea il processo primario, che a sua volta può creare processi figli.

- Un processo daemon
esegue operazioni di sistema e non dipende dal terminale di controllo. Viene eseguito in background, talvolta rimanendo in esecuzione durante tutto il funzionamento del sistema.
- Un processo kernel
esegue il codice del kernel e vengono eseguiti anch'essi in background. Sono creati all'avvio del sistema.

Anche un thread è un programma in esecuzione ma usa le risorse di un processo, quindi somiglia a un processo in tutti gli aspetti. La gestione dei thread genera meno overhead rispetto alla gestione dei processi, anche quanto riguarda la comunicazioni interprocesso. E' possibile che molti thread vengano eseguiti nell'ambito dello stesso processo. Un thread differisce da un processo nel fatto che

1. A esso non sono allocate risorse e che non può richiederle. Ma i passaggi di stato ad esclusione degli stati swapped sono identici.
2. Comunicazione veloce, visto che i thread (diversamente dai processi) condividono lo spazio di indirizzamento del processo genitore, possono comunicare tra loro attraverso dati condivisi anziché mediante messaggi.
3. La progettazione semplificata, infatti, l'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente

Ci sono diversi tipi di thread:

- Thread di livello kernel

Implementato dal kernel, dunque la creazione, la terminazione e il recupero dello stato di un thread kernel sono effettuati mediante system call. L'overhead dovuto alla commutazione, nel caso di thread di livello kernel, risulta 10 volte più veloce rispetto alla commutazione tra processi.

- VANTAGGI: un thread di livello kernel è come un processo eccetto che ha una quantità inferiore di informazioni di stato. Questa similarità conviene ai programmatore visto che la programmazione dei thread non differisce molto dalla programmazione dei processi.
- SVANTAGGI: la commutazione dei thread è effettuata dal kernel, dunque viene generato overhead anche se il thread interrotto e il thread selezionato appartengono allo stesso processo.

- I thread di livello utente

Implementati da una libreria di thread, che viene linkata al codice del processo. In questo tipo di thread il kernel non viene coinvolto ed è la libreria stessa che gestisce l'alternanza dell'esecuzione dei thread nel processo.

- VANTAGGI: la sincronizzazione e la schedulazione dei thread sono implementate dalla libreria. Questa organizzazione evita l'overhead della system call per la sincronizzazione dei thread, per cui l'overhead dovuto alla commutazione dei thread potrebbe essere 10 volte più veloce rispetto ai thread di livello kernel.
- SVANTAGGI: usando thread di questo tipo, il kernel non conosce la differenza tra thread e processo, per cui se un thread si bloccasse su una system call, il kernel bloccherebbe il processo genitore.

- I thread ibridi possono essere ottenuti da differenti combinazioni caratterizzate da ridotto overhead di commutazione dei thread di livello utente ed elevata concorrenza e parallelismo dei thread di livello kernel. La libreria di thread crea thread utente in un processo e, a ogni thread utente, associa un thread control block (TCB). Il kernel crea i thread kernel in un processo e, a ogni thread kernel, associa un kernel thread control block (KTCB).

Soltamente, un processo crea uno o più figli in modo tale da delegare ad ognuno di essi una parte del suo lavoro; questa tecnica prende il nome di multitasking e presenta tre benefici:

1. Speedup dell'elaborazione

Diminuzione del tempo di esecuzione dell'applicazione grazie alla creazione di processi figli. Se il processo primario non creasse processi figli, eseguirebbe le operazioni sequenzialmente; invece creando i processi figli, questi eseguono concorrentemente le operazioni.

2. Priorità per le funzioni critiche

Molti SO consentono a un processo genitore di assegnare priorità ai processi figli. Un'applicazione real-time può assegnare una priorità alta a un processo figlio che deve eseguire una funzione critica in modo tale da soddisfare i suoi requisiti di risposta.

3. Proteggere un processo genitore dagli errori

Il kernel può terminare un processo figlio in caso di errore, ma il padre resta protetto e può avviare un'azione di recupero.

L'esecuzione di un programma può essere velocizzata utilizzando sia il parallelismo che la concorrenza.

- Il parallelismo si riferisce alla caratteristica di verificarsi allo stesso tempo, dunque due processi sono eseguiti in parallelo se sono eseguiti nello stesso tempo. Questa tecnica di programmazione è realizzata usando più CPU.
- La concorrenza è un'illusione di parallelismo, infatti c'è solo l'illusione che i processi sono eseguiti nello stesso tempo. In realtà, il singolo processore, dotato di un'alta velocità di elaborazione, dà all'utente solo la sensazione di eseguire più processi nello stesso tempo, ma, in effetti, esegue i processi uno per volta.

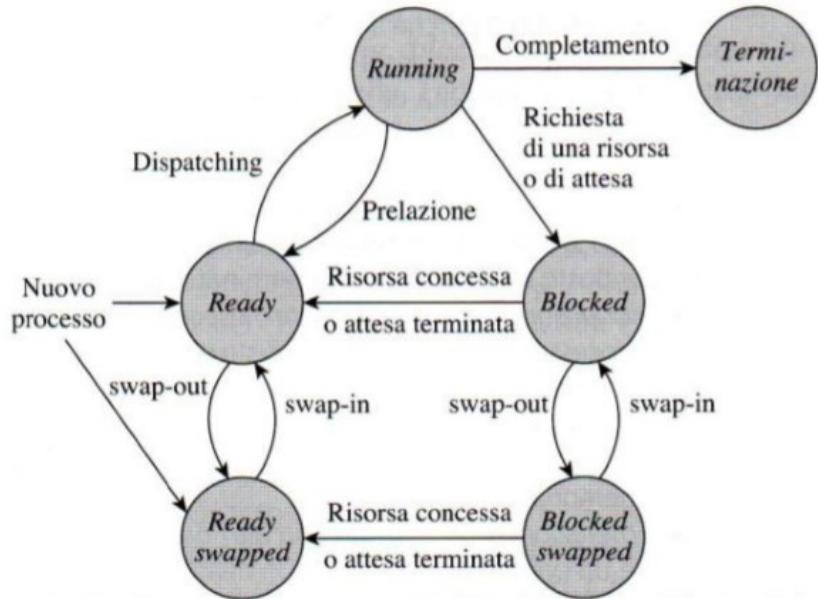
Nello specifico, un processo/thread comprende sei componenti:

- Id: identificativo univoco assegnato dal SO
- Codice: codice del programma
- Dati: dati usati durante l'esecuzione del programma, inclusi i dati contenuti nei file
- Stack: contiene i parametri e gli indirizzi di ritorno delle funzioni chiamate durante l'esecuzione
- Risorse: risorse allocate dal SO
- Stato della CPU: composto dal contenuto del PSW e dei registri GPR della CPU inoltre contiene una serie di informazioni molto importanti come la prossima istruzione da eseguire e il contenuto del campo CC (condition code).

Un processo è identificato anche dallo stato in cui si trova in particolare, quando un processo è in:

- BLOCKED
quando è in attesa che una risorsa da lui richiesta venga allocata o quando è in attesa del verificarsi di un particolare evento.
- BLOCKED SWAPPED
se l'utente specifica che il processo non deve essere considerato schedulabile per un certo periodo di tempo (è comunque swappato sul disco).
- READY
quando può essere schedulato, cioè quando la richiesta è soddisfatta o quando si è verificato l'evento che stava attendendo.
- READY SWAPPED
quando viene swappato sul disco e rimane in attesa. Prima che possa riprendere l'esecuzione deve essere riportato in memoria.
- RUNNING
quando la CPU sta eseguendo le istruzioni del processo stesso.
- TERMINATED
quando viene portata a termine la sua esecuzione o se, per qualche motivo, viene terminato dal kernel.

I passaggi di stato tra uno stato di un processo ad un altro sono:

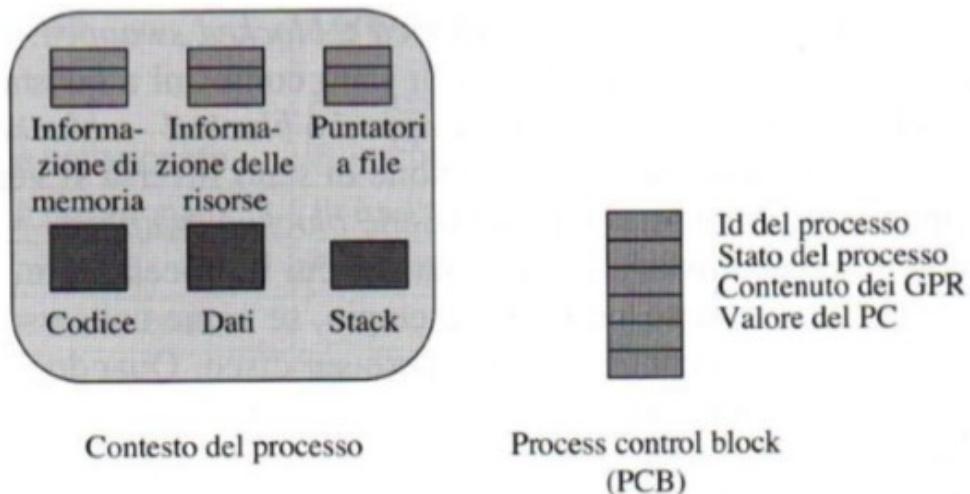


La commutazione di stato di un processo può avvenire in questo sensi:

- (NEW) -> READY: un nuovo processo entra nello stato ready dopo che le risorse richieste sono state allocate. In questo stato ci sono tutti i processi che si trovano in memoria centrale. N.B. per andare in esecuzione, il processo deve essere caricato in memoria centrale.
- (NEW) -> READY SWAPPED: un nuovo processo entra nello stato ready swapped quando è pronto per eseguito ma non c'è spazio in memoria centrale. Viene salvato sul disco (nell'area di swap), quindi non è ancora selezionabile per prendere il controllo della CPU.
- READY SWAPPED -> READY: un processo che si trova nello stato di ready swapped, prima di poter ottenere l'uso della CPU, deve prima essere 'swappato' in memoria centrale mediante un'operazione di swap-in.
- READY -> READY SWAPPED: se il sistema ha l'esigenza di liberare la memoria per l'esecuzione di altri processi, può swappare alcuni processi sul disco mediante un'operazione di swap-out.
- READY -> RUNNING: un processo può andare nello stato running solo se in precedenza si trovava nello stato ready e ci va non appena viene completato il dispatching. In pratica un processo va in esecuzione non appena gli viene passato il controllo della CPU. Dallo stato running può raggiungere tutti gli altri stati.
- RUNNING -> TERMINAZIONE: un processo passa dallo stato running a quello di terminazione quando viene portata a termine l'esecuzione del programma. Le cause che portano alla terminazione di un programma sono molteplici (es: auto-terminazione, terminazione richiesta dal processo padre, eccesso di utilizzo di una risorsa, condizioni anomale durante l'esecuzione, interazione non corretta con altri processi). Quando un processo è terminato, vengono liberate tutte le risorse che gli erano state assegnate.
- RUNNING -> READY: un processo passa dallo stato running a quello ready quando viene prelazionato poiché il kernel decide di schedulare un altro processo (es: un processo a priorità più alta va nello stato ready oppure la time-slice del processo si esaurisce).
- RUNNING -> BLOCKED: un processo passa dallo stato running a quello blocked quando effettua una system call per richiedere l'uso di una risorsa o quando rimane in attesa di un evento. Un processo bloccato si trova in memoria, ma non necessariamente in memoria centrale (ad esempio in memoria swap).
- BLOCKED -> BLOCKED SWAPPED: analoga situazione a quella in alto (ready -> ready swapped).
- BLOCKED SWAPPED -> READY SWAPPED: questa transizione si verifica se viene soddisfatta la richiesta per cui il processo è in attesa o se si verifica l'evento. Ciò non toglie che il processo si trova sempre sul disco nell'area di swap e non in memoria centrale.
- BLOCKED -> READY: quando la richiesta del processo viene soddisfatta o quando si verifica l'evento, il processo passa dallo stato blocked a quello ready.

Tutte le informazioni che è necessario salvare e ripristinare durante la commutazione (avviene il cambio di un processo) prendono il nome di informazione di stato di un processo. L'overhead dovuto alla commutazione dipende dalla dimensione dell'informazione di stato del processo. In particolare, si utilizza il Process Control Block (PCB).

Process Control Block o PCB è una struttura dati che contiene informazioni sul processo ad esso correlato. Il blocco controllo processo è anche noto come blocco controllo attività, immissione della tabella processi, ecc. Da non dimenticare che il PCB è posizionato all'inizio dello stack kernel per tenere a "sicuro" le informazioni dei processi.



In particolare, il contesto del processo si compone delle seguenti parti:

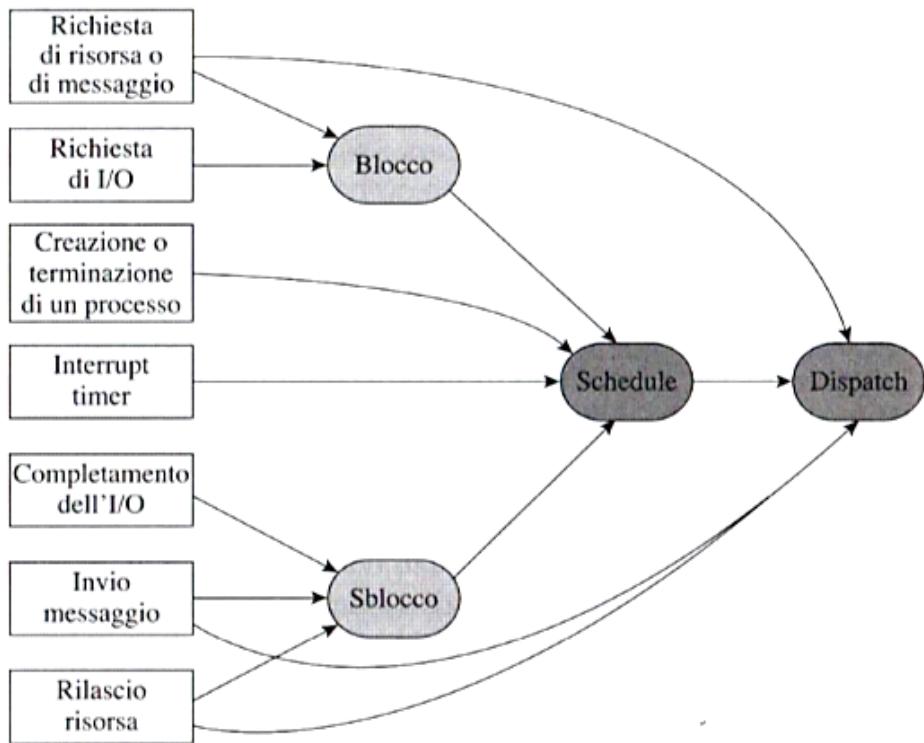
Campo del PCB	Contenuto
Id del processo	L'id univoco assegnato al processo al momento della creazione.
Id del genitore e dei figli	Questi id sono usati per la sincronizzazione dei processi, tipicamente per consentire a un processo di verificare se un processo figlio ha terminato la sua esecuzione.
Priorità	La priorità è generalmente un valore numerico. Al momento della creazione, al processo viene assegnata una priorità. Il kernel può cambiare la priorità dinamicamente in base alla natura del processo (CPU-bound o I/O-bound), al tempo di attività e alle risorse utilizzate (solitamente il tempo di CPU).
Stato del processo	Lo stato corrente del processo.
PSW	Questa è un'istantanea, ovvero un'immagine del PSW eseguita l'ultima volta che il processo è stato bloccato o prelazionato. Il caricamento di questa immagine nel PSW ripristina l'esecuzione del processo (vedi Figura 2.2 per i campi del PSW).
GPR	Il contenuto dei registri general purpose salvati l'ultima volta che il processo è stato bloccato o prelazionato.
Informazione sugli eventi	Per un processo nello stato <i>blocked</i> , questo campo contiene l'informazione relativa all'evento per il quale il processo è in attesa.
Informazioni sui segnali	Informazione relativa ai gestori dei segnali (vedi Paragrafo 5.2.6).
Puntatore al PCB	Questo campo è utilizzato per creare una lista di PCB, necessaria per la schedulazione.

Quando si verifica un evento, il kernel deve trovare il processo il cui stato è influenzato dall'evento. L'operazione è effettuata mediante una ricerca nei campi delle informazioni relative agli eventi dei PCB ma sarebbe costoso. Per cui i SO usa un campo appositamente dedicato: Event Control Block (ECB).

Un ECB contiene tre campi:

- Descrizione dell'evento, che descrive un evento
- ID del processo, che contiene l'ID del processo in attesa dell'evento
- Puntatore all'ECB, che è necessario per inserire l'ECB di un processo nella lista appropriata, in quanto il kernel può mantenere più liste separate di ECB per ogni classe di evento.

La Gestione di un evento che controlla lo stato di un processo è vista così usando una ECB:



UNIX utilizza due strutture dati per memorizzare i dati di controllo relativi ai processi:

1. Struttura processo

memorizza i dati relativi alla schedulazione. In particolare, contiene ID processo, stato del processo, priorità, relazioni con altri processi, descrittore dell'evento per il quale un processo bloccato è in attesa, maschera per la gestione dei segnali, informazioni relative alla gestione della memoria

2. U-area

contiene i dati relativi all'allocazione delle risorse e alla gestione dei segnali. La user-area contiene un PCB per conservare lo stato di un processo bloccato, il puntatore alla struttura proc, gestore dei segnali, file aperti, directory corrente.

Queste due strutture mantengono l'informazione analoga alla struttura PCB vista precedentemente

Col termine processi concorrenti ci si riferisce a processi il cui comportamento è influenzato dalla contemporanea presenza di altri processi. I processi inter comunicanti sono processi concorrenti che condividono dati o che coordinano le loro attività. I processi che non interagiscono tra loro sono detti processi indipendenti.

La sincronizzazione per l'accesso ai dati serve per garantire la consistenza dei dati condivisi. Si tratta, in pratica, di coordinare i processi per implementare la mutua esclusione. Indica il meccanismo di sincronizzazione con il quale si impedisce che più processi accedano contemporaneamente agli stessi dati in memoria.

1. La mutua esclusione viene implementata usando le sezioni critiche, queste sono sezioni del codice che possono essere eseguite da un solo processo per volta, come se fosse un blocco atomico. Un'operazione atomica (detta anche operazione indivisibile) è il mezzo che assicura l'esecuzione di una sequenza di istruzioni senza essere prelazionati.

Affinché sia possibile la mutua esclusione occorrono tre condizioni:

1. Mutua esclusione: un solo processo alla volta può accedere alla sezione critica.
2. Progresso: l'uso di una sezione critica non può essere "riservato" ad un processo; in pratica, nessun processo fuori dalla sezione critica può impedire ad un altro processo di entrare.
3. Attesa limitata: nessun processo deve attendere indefinitivamente prima di entrare nella sezione critica.

Ci sono vari modi per soddisfare i requisiti della mutua esclusione:

1. Ciclare – Approccio software

Un processo (P0 o P1) che vuole eseguire la sua sezione critica controlla la variabile globale condivisa turno; tale variabile indica quale processo potrà entrare nella sezione critica; se un processo non entra continua a ciclare fin quando la variabile non gli dà disponibilità di accesso.

while (qualche processo è nella sezione critica su $\{d_s\}$ o
sta eseguendo un'operazione indivisibile utilizzando $\{d_s\}$)
{ non fare niente }

Sezione critica oppure
operazione indivisibile
tramite $\{d_s\}$

Quest'ultima situazione prende il nome di busy waiting (o attesa attiva) poiché si mantiene la CPU durante l'esecuzione di un processo anche se il processo non fa nulla.

2. Bloccare – Approccio hardware

Per evitare le attese attive, un processo in attesa di entrare in una sezione critica deve andare nello stato blocked invece di continuare a ciclare inutilmente. impostato a ready solo quando gli è consentito di entrare in sezione critica. La sincronizzazione tra processi viene implementata utilizzando istruzioni macchina speciali fornite dall'architettura, unite all'uso di variabili condivise, chiamate variabili di lock.

if (qualche processo è in una sezione critica su $\{d_s\}$ o
sta eseguendo un'operazione indivisibile usando $\{d_s\}$)
effettua una chiamata di sistema per bloccarsi;

Sezione critica oppure
operazione indivisibile
tramite $\{d_s\}$

I SO implementano le sezioni critiche e le operazioni atomiche attraverso istruzioni indivisibili fornite dai computer, insieme a variabili condivise chiamate variabili di lock.

Come detto, una variabile di lock è una variabile a due stati – aperto e chiuso – utilizzata dai processi per accedere alle sezioni critiche.

Quando un processo vuole eseguire una sezione critica, legge il contenuto della variabile di lock e:

1. se il lock è aperto (valore 0),
 - a. il processo impone il valore del lock a chiuso (valore 1)
 - b. ed esegue la sezione critica,
 - c. dopodiché, al termine, lo impone di nuovo ad aperto.
2. se il lock è chiuso (1),
 - a. il processo attende che diventi aperto (0).

Per evitare race condition nell'impostazione della variabile di lock, viene utilizzata un'operazione indivisibile per la lettura e la chiusura.

Una soluzione viene proposta con l'algoritmo di Peterson:

Consideriamo il processo P0. Se vuole entrare in sezione critica impone flag[0]=true. Dopodiché cerca di favorire l'altro processo impostando turn=1. A questo punto P0 controlla se l'altro processo entra in sezione critica, cioè se il suo flag è impostato a true e se turn=1. Se queste condizioni si verificano, allora P0 continua a ciclare (attesa attiva). Altrimenti entra in sezione critica. Se entrambi i processi, P0 e P1, vogliono entrare in sezione critica, il valore di turn determina quale processo può entrare nella propria sezione critica. All'uscita della sezione critica, il processo P0 impone flag[0]=false per permettere all'altro processo di poter accedere in sezione critica. Un ragionamento analogo è valido nel caso in cui si consideri il funzionamento di P1. Possiamo vedere la variabile flag[x] come voglio_entrare[x] e la variabile turn=y come finisci_presto=y. Viene considerato pure "processo gentile".

Algoritmo 6.4 Algoritmo di Peterson

```
var      flag : array [0..1] of boolean;
turn : 0..1;
begin
    flag[0] := false;
    flag[1] := false;
Parbegin
repeat
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1
        do { niente };
    { Sezione critica }
    flag[0] = false;
    { Resto del ciclo }
    forever;
Parend;
end.
```

Processo P_0

```
repeat
    flag[1] := true;
    turn := 0;
    while flag[0] and turn = 0
        do { niente };
    { Sezione critica }
    flag[1] = false;
    { Resto del ciclo }
    forever;
```

Processo P_1

L'algoritmo di Peterson, risolve il problema della mutua esclusione ma in un modo più semplice

La variabile $turn$ indica a chi spetta entrare nella sezione critica, mentre l'array $flag$ indica se un processo è pronto ad entrare nella propria sezione critica (cioè indica l'intenzione di entrare).

Utilizza, invece che variabili di $flag$, un array di $flag$ contenente un $flag$ per ogni processo (qui, visto che ci sono 2 processi, conterrà due $flag$). Il valore di ciascun $flag$ è di tipo booleano, *true* o *false*.

Le soluzioni che si basano sull'approccio basato su linguaggio di programmazioni usano i meccanismi di IPC (InterProcess Communication) che sono essenzialmente:

- SCAMBIO DI MESSAGGI

Meccanismi di comunicazione stile I/O per il trasferimento di informazioni tra due o più processi.

- SEMAFORI

Sono variabili su cui sono definite operazioni indivisibili.

Un semaforo è una particolare struttura di sincronizzazione. Consiste in una variabile intera condivisa a valori non negativi che può essere soggetta solo alle seguenti operazioni:

- o Inizializzazione (specificata come parte della sua dichiarazione)
- o Le operazioni indivisibili *wait* e *signal* che si distinguono per le loro operazioni:
 - Quando un processo effettua una *wait* su un semaforo

1. controlla se il valore del semaforo è > 0.
2. in caso affermativo, decrementa il valore del semaforo e consente al processo di proseguire la sua esecuzione.
3. altrimenti, blocca il processo sul semaforo.
 - Quando un processo effettua una *signal* su un semaforo

1. controlla se ci sono processi bloccati sul semaforo.
2. in caso affermativo, l'operazione attiva un processo bloccato sul semaforo;
3. altrimenti incrementa il valore del semaforo di un'unità.

```
procedure wait (S)
begin
    if S > 0
        then S := S-1;
        else blocca il processo su S;
end;

procedure signal (S)
begin
    if qualche processo è bloccato su S
        then attiva un processo bloccato;
        else S := S+1;
end;
```

Uso dei semafori nei sistemi concorrenti serve per implementare:

- o La mutua esclusione è utile per implementare le sezioni critiche. Ogni processo effettua:
 - una *wait* sul semaforo prima di entrare nella propria sezione critica
 - una *signal* al termine della sezione critica.

In particolare, si utilizza un semaforo binario (detto anche mutex) che permette l'operazione indivisibile sulla sezione critica, essendo impostato ad 1 ogni operazione signal è riserva ad un solo processo per volta.

- La concorrenza limitata è importante quando una risorsa può essere condivisa da al più $c \geq 1$ processi
 - Può essere creato instanziando il valore del semaforo (chiamati contatori) pari a c . Il valore del semaforo rappresenta:
1. O il numero di accessi consentiti (risorse libere) se ≥ 0 .
 2. O il numero di processi in attesa se < 0 (in valore assoluto).
 - La segnalazione è utile nel controllo della sincronizzazione. Viene usata quando un processo P_i vuole effettuare un'operazione (a_i) solo dopo che un processo P_j ha effettuato un'altra operazione (a_j).
 - Viene implementata utilizzando un semaforo inizializzato a 0.
 - P_i effettua una wait sul semaforo prima di effettuare l'operazione a_i .
 - P_j effettua una signal sul semaforo dopo aver effettuato l'operazione a_j .

- MONITOR

Un monitor è un costrutto di sincronizzazione che può essere utilizzato da due o più processi per rendere mutuamente esclusivo l'accesso a risorse condivise in maniera semplice. Esso è un modulo software che contiene:

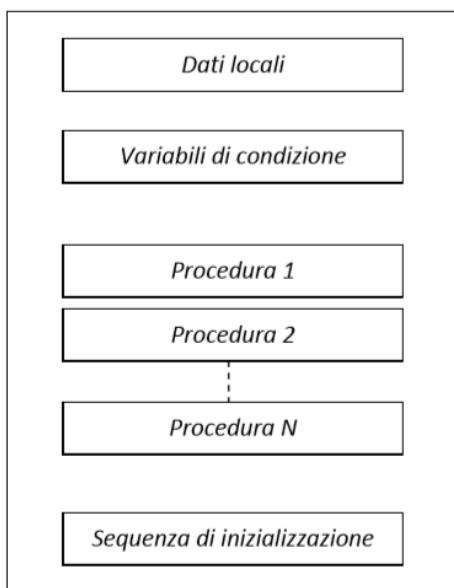
- una o più funzioni e procedure, una sequenza di inizializzazione e variabili locali.
- può contenere le dichiarazioni di speciali dati di sincronizzazione chiamati variabili di condizione su cui possono essere effettuate solo le operazioni wait e signal.

Le caratteristiche principali di un monitor sono le seguenti:

- le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne
- un processo entra nel monitor chiamando una delle sue procedure
- solo un processo alla volta può essere in esecuzione all'interno del monitor; ogni altro processo che ha chiamato il monitor è sospeso, nell'attesa che questo diventi disponibile

Le chiamate delle operazioni sono servite in ordine FIFO per soddisfare la proprietà di attesa limitata. In pratica, la coda di processi entranti è di tipo FIFO.

Esempio di struttura di un monitor (sx) e del corpo di un monitor (dx)



```
monitor assegnazione_risorse
{
    .
    boolean occupato;
    condition x;

    void acquisizione(int tempo) {
        if (occupato)
            x.wait(tempo);
        occupato = true;
    }

    void rilascio() {
        occupato = false;
        x.signal();
    }

    void inizializzazione() {
        occupato = false;
    }
}
```

All'interno del monitor sono implementate variabili di condizione che sono un modo per bloccare i processi quando non possono proseguire. Una variabile di condizione è una variabile con l'attributo condizione ed è associata a una condizione nel monitor. Le uniche due operazioni eseguibili su una variabile di condizione sono wait e signal che differiscono dai normali semafori per:

- la wait sospende l'esecuzione del processo chiamante sulla condizione c ; il monitor diventa disponibile per gli altri processi
- la signal riattiva un processo sospeso sulla condizione c ; se i processi sospesi sono molti, ne sceglie uno; se non ce ne sono, non fa niente

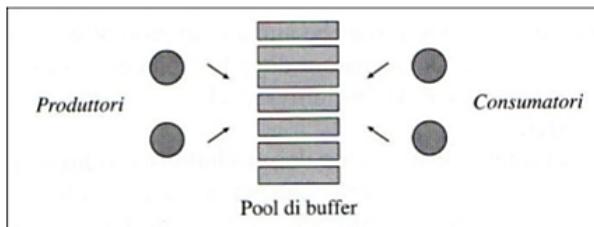
- o la wait e la signal non sono contatori

Inoltre:

- o Ad ogni semaforo è associata una coda di processi in attesa sul semaforo.
- o Ad ogni variabile di tipo condition è associata una coda su cui attendono i processi sospesi
- o L'operazione wait su un semaforo è sospensiva solo se il semaforo non è positivo
- o L'operazione wait su un monitor è immediatamente sospensiva .

Problemi classici di sincronizzazione dei processi: Sono illustrati svariati casi la cui soluzione può essere data tramite l'uso di semafori e monitor. In particolare, esistono 4 tipi differenti di "esercizi" che vanno a definire quale delle possibili soluzioni sono valide o meno:

1. Produttore consumatore:



Questo problema è anche noto come *problema del buffer a capienza limitata*.

Esso si compone di un numero non specificato di *produttori* e *consumatori*, e di un insieme finito di *buffer*, ciascuno dei quali è in grado di contenere un elemento di informazione.

Un *buffer* è *pieno* quando un produttore scrive un nuovo elemento al suo interno.

Un *buffer* è *vuoto* quando un consumatore estrae un elemento contenuto in esso.

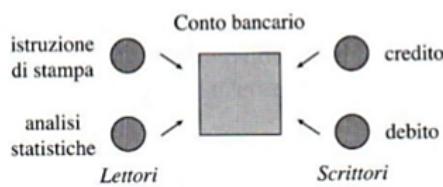
Un *processo produttore* produce un elemento di informazione alla volta e lo inserisce in un buffer vuoto.

Un *processo consumatore* estrae l'informazione, un elemento alla volta, da un buffer pieno.

Una soluzione al problema produttori-consumatori deve soddisfare queste condizioni:

1. un produttore non deve sovrascrivere un buffer pieno
2. un consumatore non deve consumare un buffer vuoto
3. i produttori e i consumatori devono accedere ai buffer in maniera mutuamente esclusiva

2. Lettore Scrittore:



Esso si compone di un numero non specificato di *lettori* e *scrittori*. Entrambi agiscono su dati differenti.

Un *processo lettore* può esclusivamente leggere i dati.

Un *processo scrittore* può modificare o aggiornare i dati.

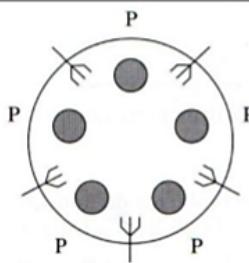
Si utilizzeranno i termini *leggere* e *scrivere* per gli accessi ai dati condivisi eseguiti, rispettivamente, dai processi lettori e scrittori.

Una soluzione al problema lettori-scrittori deve soddisfare queste condizioni:

1. molti lettori possono effettuare la lettura in maniera concorrente
2. solo uno scrittore per volta può eseguire la scrittura
3. la lettura è proibita quando uno scrittore sta eseguendo una scrittura

I punti 1-3 consentono o a uno scrittore di eseguire la scrittura o a più lettori di eseguire letture concorrenti.

3. Filosofi a Cena:



Ci sono cinque filosofi seduti attorno a un tavolo. Ogni filosofo ha davanti a sé un piatto, mentre tra ogni coppia di filosofi c'è una forchetta. I filosofi alternano momenti durante i quali meditare e momenti durante i quali mangiare.

Per mangiare, un filosofo deve prendere, una alla volta, le due forchette che si trovano alla sua destra e alla sua sinistra. Durante la meditazione, invece, un filosofo deve tenere le forchette sul tavolo.

Risulta evidente che il numero di forchette impedisce a tutti i filosofi di mangiare contemporaneamente quindi una corretta programmazione concorrente deve essere in grado di far mangiare alternativamente tutti i filosofi evitando che qualcuno in particolare soffra di starvation ed evitando che si verifichino stalli in fase di "acquisizione delle forchette". In pratica:

- la condizione di *correttezza* è mantenuta se un filosofo affamato non attende indefinitamente quando decide di mangiare
- i *deadlock* sono evitati quando i processi non si bloccano rimanendo in attesa l'uno dell'altro
- i *livelock* sono evitati quando i processi non ritardano l'esecuzione a favore degli altri in modo indefinito

4. Barbiere che dorme:

Un barbiere possiede un negozio con una sola sedia da lavoro e un certo numero limitato di posti per attendere. Se non ci sono clienti il barbiere dorme; altrimenti, all'arrivo del primo cliente il barbiere si sveglia ed inizia a servirlo. Se dovessero sopraggiungere clienti durante il periodo di attività del barbiere, essi si metterebbero in attesa sui posti disponibili. Al termine dei posti di attesa, un ulteriore cliente viene scartato. Una corretta programmazione concorrente deve far "dormire" il barbiere in assenza di clienti, attivare il barbiere sul primo cliente al suo arrivo e mettere in coda tutti i successivi clienti tenendoli inattivi.

Lo scheduler sfrutta una politica di scheduling che decide a quale processo debba essere assegnata la CPU in un certo momento. Per questo motivo, le decisioni dello scheduler influenzano sia il servizio utente che le prestazioni del sistema.

Esistono principalmente le tre tecniche che usa lo scheduler per ottenere la combinazione ottimale tra servizio utente e prestazioni del sistema:

1. Assegnazione delle priorità
2. Riordino delle richieste
3. Variazione della time-slice

La parte del sistema operativo che decide quale processo eseguire viene chiamata scheduler e l'algoritmo che usa è chiamato algoritmo di schedulazione. Esistono dei concetti che devono essere ben chiari per poter apportare un uso efficiente dello scheduler:

CONCETTI DI SCHEDULING RELATIVI ALLA RICHIESTA:

Termine o concetto	Definizione o descrizione
Relativamente alla richiesta	
Tempo di arrivo	Istante in cui un utente invia un job o un processo.
Tempo di ammissione	Istante in cui il sistema comincia a considerare un job o un processo per lo scheduling.
Tempo di completamento	Istante in cui un job o un processo è terminato.
Deadline	Istante entro il quale un job o un processo deve essere terminato per rispettare il requisito di risposta di un'applicazione real-time.
Tempo di servizio	Il totale del tempo di CPU e quello di I/O richiesto da un job, processo o sottorichiesta per completare la sua operazione.
Preelazione	Deallocazione forzata della CPU da un job o da un processo.
Priorità	Una regola discriminante usata per selezionare un job o un processo quando molti job o processi attendono l'elaborazione.

CONCETTI DI SCHEDULING RELATIVI AL SERVIZIO PER L'UTENTE:

Relativamente al servizio per l'utente: richieste individuali

Superamento della deadline (deadline overrun)	La quantità di tempo per la quale il tempo di completamento di un job o un processo supera la sua deadline. Il deadline overrun può essere sia positivo sia negativo.
Condivisione equa	Una condivisione specifica del tempo di CPU che dovrebbe essere dedicato all'esecuzione di un processo o di un gruppo di processi.
Rapporto di risposta	Il rapporto $\frac{\text{tempo di attesa} + \text{tempo di servizio di un job o processo}}{\text{tempo di servizio del job o processo}}$
Tempo di risposta (rt)	Il tempo tra la sottomissione di una sottorichiesta per l'elaborazione e il tempo in cui il suo risultato diventa disponibile. Questo concetto è applicabile ai processi interattivi.
Tempo impiegato per il completamento o tempo di turnaround (ta)	Il tempo che intercorre tra la sottomissione di un job o processo e il suo completamento da parte del sistema. Questo concetto è significativo solo per job non interattivi o processi.
Turnaround pesato (w)	Rapporto del tempo di turnaround di un job o processo e il suo tempo di servizio.

Relativamente al servizio per l'utente: servizio medio

Tempo di risposta medio (\bar{rt})	La media dei tempi di risposta di tutte le sottorichieste elaborate dal sistema.
Tempo medio di turnaround (\bar{ta})	La media dei tempi di turnaround di tutti i job o processi elaborati dal sistema.

CONCETTI DI SCHEDULING RELATIVI AL SISTEMA:

Relativamente alle prestazioni

Durata della schedulazione	Il tempo necessario per completare un insieme specifico di job o processi.
Throughput	Il numero medio di job, processi o sottorichieste completate da un sistema nell'unità di tempo.

Visto che un SO deve fornire una giusta combinazione tra prestazioni del sistema e servizio utente, occorrono diversi scheduler (cioè diversi algoritmi) ognuno dei quali può utilizzare una combinazione di diverse politiche di scheduling. Infatti, in un moderno SO, possono essere impiegati fino a tre scheduler:

1. Scheduler a lungo termine

decide quale processo entra nella coda dei processi ready tra quelli che la richiedono, perciò controlla il numero di processi in memoria. Inoltre, può ritardare l'ammissione di una richiesta per due motivi:

- I. o non riesce ad allocare risorse sufficienti.
- II. o l'ammissione della richiesta porterebbe ad un calo delle prestazioni del sistema.

2. Scheduler a medio termine

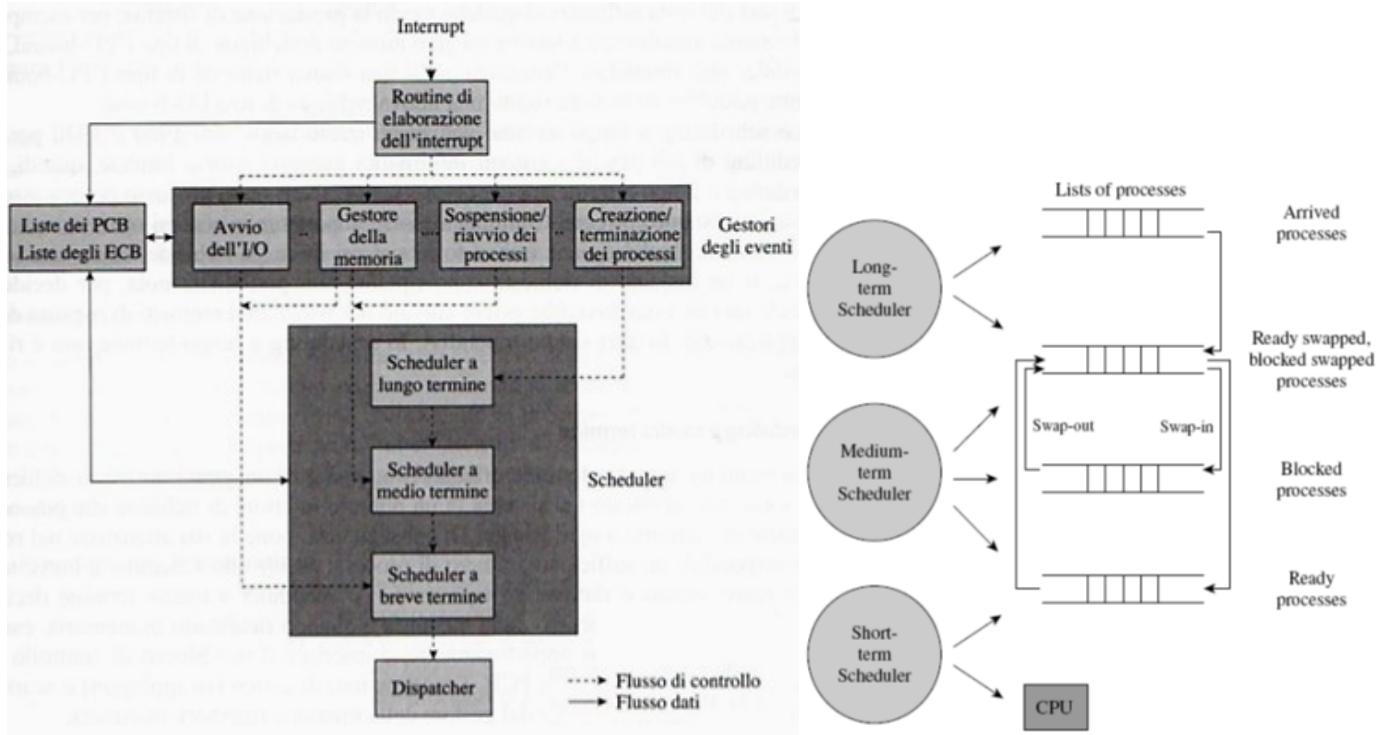
gestisce la permanenza in memoria dei processi non in esecuzione. In pratica decide quali processi possono risiedere in memoria e quali sul disco. Le operazioni di carico (swap in) e scarico (swap out) sono comunque effettuate dal gestore della memoria.

3. Scheduler a breve termine

è detto anche dispatcher o scheduler. Decide quale processo nello stato ready può andare in esecuzione. Inoltre, può anche decidere il tempo di utilizzo della CPU per il processo selezionato. I momenti in cui può intervenire lo scheduler a breve termine sono su eventi che possono causare il passaggio ad un altro processo (interrupt di clock, interrupt I/O, system call, segnalazioni tra processi). Sono principalmente quattro:

- I. Quando un processo passa dallo stato running allo stato blocked per selezionare quale altro processo ready andrà in esecuzione. (preemptive) si deve scegliere un altro processo ready.

- II. Quando un processo passa dallo stato blocked allo stato ready per determinare con quali altri processi ready compete. (non preemptive) si può scegliere un altro processo ready, oppure si può lasciare o mandare in esecuzione il processo che ha cambiato stato.
- III. Quando un processo passa dallo stato running allo stato ready (es. per una interruzione) per determinare se il processo tornerà in esecuzione subito oppure no. (non preemptive) si può scegliere un altro processo ready, oppure si può lasciare o mandare in esecuzione il processo che ha cambiato stato.
- IV. Quando un processo termina per selezionare quale processo verrà eseguito. (preemptive) si deve scegliere un altro processo ready.



In particolare, le politiche di scheduling si dividono in preemptive e non:

- La politica di scheduling preemptive (con prelazione) dà la possibilità di interrompere (anche più volte) il processo correntemente in esecuzione a favore di un altro processo. Il processo interrotto viene riportato all'interno della lista dei processi ready in modo tale che prima o poi possa essere rielezionato dallo scheduler. Essa seleziona un processo e lo lascia in esecuzione per una certa quantità di tempo massima. Se il processo, alla fine del tempo concessogli, è ancora in esecuzione, allora viene sospeso e lo scheduler seleziona un altro processo a cui attribuire l'uso della CPU. Una politica del genere richiede un interrupt timer alla fine dell'intervallo di tempo per restituire il controllo della CPU allo scheduler. Questa caratteristica genera un maggior overhead rispetto allo scheduling non preemptive.
- La politica di scheduling non preemptive (senza prelazione) non dà la possibilità di interrompere un processo in esecuzione. Essa seleziona un processo e lo lascia in esecuzione finché non si blocca (per attendere un evento o richiedere una risorsa) oppure finché non completa le sue operazioni rilasciando volontariamente la CPU. Lo scheduler ha soltanto la funzione di riordinare le richieste per migliorare il servizio utente o le prestazioni del sistema.

Si dividono anche per uso della priorità:

- Le politiche che usano la priorità, selezionano i processi a cui attribuire l'uso della CPU proprio in base alla priorità acquisita da ciascuno di essi. Queste politiche sono necessarie nei sistemi real-time e nei sistemi interattivi.
- Le politiche che non usano la priorità, considerano i processi "equivalenti", cioè tutto sullo stesso piano senza privilegiare l'uno rispetto ad un altro. Queste politiche sono basate su strategie di ordinamento First Come First Served.

E si differenziano anche se siano statiche e/o dinamiche:

- Nelle politiche statiche ogni processo conserva nel tempo i suoi diritti di accesso alla CPU. In pratica le decisioni di schedulazione sono prese prima che il sistema inizi l'esecuzione.
- Nelle politiche dinamiche ogni processo modifica i propri diritti di accesso alla CPU in base al comportamento passato o estrapolando quello futuro. In pratica le decisioni di schedulazione sono prese durante l'esecuzione dei processi.

Esistono tre ambienti differenti, per i quali esistono determinati algoritmi di schedulazione:

- Batch
non ci sono processi impazienti in attesa di una risposta veloce. Quindi sono accettabili sia algoritmi non preemptive, sia quelli preemptive con un periodo lungo per ogni processo. Questo approccio riduce gli scambi tra processi, migliorando le prestazioni. Per i sistemi batch sono importanti il throughput, il tempo di turnaround e l'uso della CPU. Politiche usate: FCFS, SJF (SNPF e SRTF).
- Interattivo
il prerilascio è essenziale per evitare che un processo si impossessi della CPU. Per i sistemi interattivi è importante il tempo di risposta. Politiche usate: Round-Robin, Priorità, (Code Multiple e HRRN)
- Real-time
il prerilascio non è sempre necessario perché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e normalmente fanno il loro lavoro e si bloccano in fretta. (Si analizza successivamente, con appositi algoritmi date dalle deadline, ossia, la durata massima del processo nella CPU). (EDF e SRM)

Algoritmi di scheduling:

- Scheduling First Come, First Served (FCFS) (non preemptive)
- Scheduling Shortest Job First (SJF) nella versione non preemptive (SNPF) (non preemptive)
- Scheduling Highest Response Ratio Next (HRRN) (non preemptive)
- Scheduling Shortest Job First (SJF) nella versione preemptive (SRTF) (preemptive)
- Scheduling Round-Robin con Time-Slicing (RR) (preemptive)
- Scheduling Least Completed Next (LCN) (preemptive)

Scheduling First Come First Served (FCFS)

TIPO: lo scheduling FCFS è un algoritmo senza prelazione, senza priorità e statico.

FUNZIONAMENTO: i processi sono schedulati nell'ordine in cui giungono al sistema, cioè il primo processo ad essere eseguito è quello che per primo ha richiesto la CPU. I processi successivi vengono schedulati con lo stesso criterio non appena il processo in esecuzione completa le sue operazioni. In pratica, i processi ready sono organizzati come una coda FIFO e i processi che richiedono la CPU vengono inseriti alla fine di questa coda.

CONCLUSIONI: questo tipo di algoritmo è semplice da implementare ma solitamente è poco efficiente, almeno considerando il tempo medio d'attesa. E' un algoritmo che privilegia i processi CPU bound, infatti questo algoritmo presenta il problema effetto convoglio: i processi che richiedono meno tempo di CPU (I/O bound) devono attendere che i processi che richiedono molto tempo di CPU (CPU bound) liberino la CPU.

Scheduling Shortest Job First (SJF) – Shortest Next Process First (SNPF)

Gli algoritmi SJF possono essere sia non preemptive (SNPF) che preemptive (SRTF). Qui tratteremo lo SNPF.

TIPO: lo scheduling SNPF è un algoritmo senza prelazione, con o senza priorità e dinamico.

FUNZIONAMENTO: seleziona il processo in attesa che userà la CPU per minor tempo. Se due processi hanno lo stesso tempo di esecuzione, verrà applicato lo scheduling FCFS. In pratica, le richieste brevi tendono a ricevere prima l'uso della CPU. In pratica, è come se il CPU-BURST fosse la priorità.

CONCLUSIONI: l'algoritmo SJF eleva il throughput (numero di processi portati a termine in un dato tempo) Presenta, però, due problematiche. Questo algoritmo privilegia i processi I/O bound. La prima è che è necessario conoscere in anticipo i tempi di esecuzione (tempo di servizio) dei vari processi. Visto che il SO non conosce a priori i tempi di servizio, occorre effettuare una stima dei CPU burst. Questa può essere effettuata conoscendo la 'storia passata' in modo tale da poter fare una stima del 'futuro'. Occorre comunque sapere che la media esponenziale si adatta meglio della media aritmetica per effettuare la stima. La seconda è che possiede un potenziale problema di starvation, in cui è possibile che un processo rimanga in attesa troppo tempo prima di essere completato se vengono aggiunti continuamente piccoli processi alla coda dei processi pronti.

Scheduling Highest Response Ratio Next (HRRN)

TIPO: lo scheduling HRRN è un algoritmo senza prelazione, con priorità e dinamico.

FUNZIONAMENTO: questo algoritmo viene utilizzato per prevenire l'aging, ossia l'attesa eccessiva dei processi molto lunghi scavalcati da quelli più brevi, che avviene negli algoritmi SJF. Questa politica calcola i rapporti di risposta di tutti i processi nel sistema secondo la formula:

$$\text{Rapporto di risposta: } (W+S)/S \text{ dove } W \text{ è il tempo di attesa ed } S \text{ è il tempo di servizio.}$$

Questa politica schedula il processo con il rapporto di risposta maggiore.

CONCLUSIONI: l'obiettivo è quello di dare la possibilità ai processi con un CPU-BURST elevato di essere schedulati prima dei processi con CPU-BURST più piccolo. Il rapporto di risposta dei processi brevi si incrementa più rapidamente di quello di un processo lungo; ma, comunque, il rapporto di risposta di un processo lungo, col passare del tempo, può diventare sufficientemente grande da consentire al processo di essere schedulato. Mediante questa politica si evita la starvation descritto nell'algoritmo precedente (SJF).

Scheduling Shortest Job First (SJF) – Shortest Remaining Time First (SRTF)

TIPO: Lo scheduling SRTF non è altro che la versione preemptive (con prelazione) dello scheduling SJF.

FUNZIONAMENTO: si differenzia dallo SNPF per il fatto che, quando viene sottomesso un nuovo processo la cui durata è minore del tempo necessario al processo in esecuzione per concludere le proprie operazioni, lo scheduler provvede ad effettuare un context switch per assegnare l'uso della CPU al nuovo processo. In caso di uguaglianza viene selezionato il processo che non è stato elaborato per il periodo di tempo più lungo.

CONCLUSIONI: un nuovo processo con CPU burst minore del tempo rimasto all'attuale processo in esecuzione prelaziona quest'ultimo. In pratica favorisce i processi più brevi rispetto a quelli più lunghi. Poiché è analoga alla politica SNPF, i processi lunghi possono andare incontro a starvation.

Scheduling Round-Robin con Time-Slicing (RR)

TIPO: lo scheduling RR è un algoritmo con prelazione, con o senza priorità e statico o dinamico.

FUNZIONAMENTO: ad ogni processo viene assegnato un intervallo di tempo, chiamato quanto (time slice), durante il quale al processo è assegnato l'uso della CPU. Per scandire i quanti, alla fine di ognuno di essi viene generato un timer interrupt. Se alla fine del quanto, il processo non ha terminato le sue operazioni, allora viene prelazionato e inserito di nuovo nella coda, e la CPU viene assegnata ad un altro processo. Se prima della fine del quanto, il processo si blocca o termina le sue operazioni, allora viene selezionato un altro processo a cui assegnare la CPU.

CONCLUSIONI: questo algoritmo privilegia i processi CPU bound perché utilizza l'intero quanto assegnato. Il round robin è facile da implementare: lo scheduler mantiene una coda di processi in stato ready e seleziona semplicemente il primo processo in coda e quando scade il quanto il processo viene messo in fondo alla lista. L'unica questione riguardo a questo algoritmo è la durata del quanto: assegnare un quanto troppo breve provoca troppi context switch e peggiora l'efficienza della CPU (overhead troppo elevato), ma assegnarlo troppo alto può provocare tempi di risposta lunghi per richieste interattive brevi. Se nella coda dei processi pronti esistono n processi e il quanto di tempo è pari a q, ciascun processo ottiene un 1/n-esimo del tempo di elaborazione della CPU, in frazioni di, al massimo, q unità di tempo.

Scheduling Least Completed Next (LCN)

TIPO: lo scheduling LCN è un algoritmo con prelazione, con o senza priorità e statico o dinamico.

FUNZIONAMENTO: la politica LCN seleziona il processo che ha utilizzato il minimo tempo di CPU. In caso di uguaglianza lo scheduler seleziona il processo che non è stato elaborato per il periodo di tempo più lungo.

CONCLUSIONI: tutti i processi operano approssimativamente eguali progressi in termini di tempo di CPU utilizzato, cioè questa politica garantisce che processi brevi finiranno prima di processi lunghi (I/O bound). Ha il noto svantaggio di generare la starvation dei processi lunghi. Inoltre, trascura i processi esistenti quando arrivano nuovi processi nel sistema. Per questo motivo anche i processi non troppo lunghi soffrono di starvation o di tempi lunghi di completamento.

Strutture dati e meccanismi di scheduling:

1. Prioritario

USO: lista doppiamente linkata di PCB

IN PRATICA: viene mantenuta una lista separata di processi ready per ogni valore di priorità; questa lista è organizzata come una coda di PCB, ognuno dei quali punta al PCB del successivo processo in coda.

FUNZIONAMENTO: La testa della coda contiene due puntatori: uno che punta al PCB del primo processo nella coda, l'altro che punta alla testa della coda con priorità inferiore. Lo scheduler parte dalla testa della coda con priorità maggiore e man mano scorre le teste delle code con ordine decrescente di priorità.

Seleziona il primo processo nella prima coda non vuota che trova. Le priorità possono essere assegnate dinamicamente o staticamente.

2. Multilivello

USO: code multiple

IN PRATICA: esso combina lo scheduling basato su priorità e quello round-robin per fornire una buona combinazione tra prestazione del sistema e tempi di risposta. Lo scheduling multilivello è un algoritmo con prelazione, con priorità e statico.

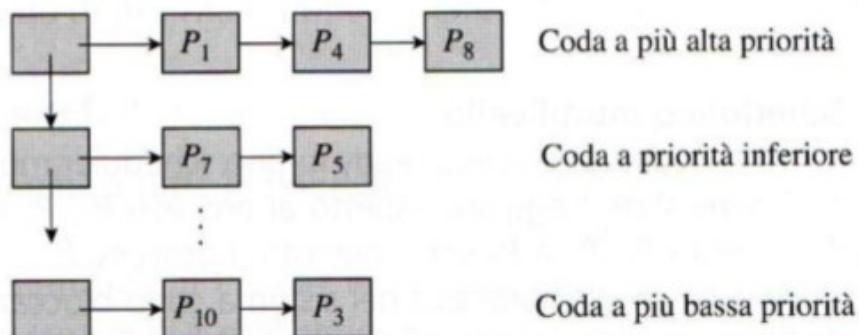
FUNZIONAMENTO: Uno scheduler multilivello utilizza varie code di processi pronti, e ad ogni classe è associata una priorità. Alla coda con priorità più alta viene assegnato un quanto più piccolo, mentre alle code con priorità via via minore viene assegnato un quanto sempre più grande. La coda con priorità più bassa avrà il quanto più grande. Un processo può essere prelazionato se arriva un processo con priorità più alta; in questo caso, il processo interrotto viene aggiunto alla fine della coda a cui apparteneva.

3. Multilivello Adattivo

USO: code multiple con feedback

IN PRATICA: è un normale scheduler multilivello ma si utilizzano le proprietà dinamiche

FUNZIONAMENTO: simile a multilivello ma permette allo scheduling di variare la priorità di un processo in modo tale da evitare la starvation dei processi. Lo scheduler analizza il comportamento passato di un processo per modificare, eventualmente, la sua priorità. Quindi, l'uso delle priorità dinamiche consente ad un processo di muoversi tra le varie code presenti.



4. Fair Share

IN PRATICA: tecnica a parte.

FUNZIONAMENTO: Tutte le politiche di scheduling viste finora si occupano di fornire servizi equi ai processi piuttosto che agli utenti o alle loro applicazioni. Se le applicazioni creano un diverso numero di processi, un'applicazione che impiega più processi è probabile che riceva maggiore attenzione dalla CPU rispetto ad un'applicazione che utilizza meno processi. La nozione di fair share risponde proprio a questa esigenza. Una fair share assicura un equo utilizzo della CPU da parte degli utenti o delle applicazioni. In pratica, gli utenti o le applicazioni che usano un numero più alto di processi ricevono meno risorse.

5. Lotteria

IN PRATICA: tecnica a parte.

FUNZIONAMENTO: Lo scheduling a lotteria è una tecnica utile per la condivisione di una risorsa in maniera probabilisticamente equa. I 'biglietti' sono distribuiti a tutti i processi che condividono una risorsa (ad esempio il tempo di CPU). Quando lo scheduler deve selezionare un processo a cui attribuire la CPU, viene scelto a caso un biglietto e il processo che lo possiede ottiene la risorsa. Per migliorare le prestazioni, ai processi più importanti possono essere assegnati biglietti extra, per aumentare la loro probabilità di vincere. Inoltre, i processi possono scambiarsi i biglietti se lo desiderano per avvantaggiare i processi con la quale cooperano per raggiungere un obiettivo comune.

I Sistemi real-time sono classificati in:

- Hard real-time: garantiscono che i compiti critici sono completati in un intervallo limitato; cioè rispetta le deadline in maniera garantita.
- Soft real-time: sono meno restrittivi; cioè che pur avendo scadenze da rispettare, se alcuni processi non lo fanno occasionalmente, sono tollerati; cioè rispetta le deadline in maniera probabilistica.

Gli eventi a cui un sistema real-time può dover reagire possono essere ulteriormente classificati in:

- periodici: che si verificano ad intervalli regolari
- aperiodici: che si verificano in modo imprevedibile

Gli algoritmi di schedulazione real-time possono essere inoltre:

- statici: cioè che prendono le decisioni di scheduling prima che il sistema inizi l'esecuzione. La schedulazione statica funziona solo quando sono disponibili in anticipo le informazioni complete circa il lavoro da fare e le scadenze da rispettare;
- dinamici: cioè che le prendono durante l'esecuzione. Mentre gli algoritmi di scheduling dinamica non hanno queste restrizioni.

Nello scheduling real-time il tempo gioca un ruolo essenziale perché si devono rispettare delle scadenze chiamate deadline. I processi di un'applicazione real-time interagiscono tra loro per assicurare che le loro azioni vengono eseguite nel giusto ordine. Per gestire le dipendenze tra i processi viene usato un grafo di precedenza tra processi. La schedulabilità è una sequenza di decisioni di scheduling che consentono ai processi di un'applicazione di agire secondo le precedenze e rispettare il requisito di risposta dell'applicazione.

Approccio	Descrizione
Scheduling statico	Uno schedule viene preparato <i>prima</i> che l'esecuzione dell'applicazione real-time cominci, tenendo conto delle interazioni tra processi, le periodicità, i vincoli sulle risorse e le scadenze.
Scheduling basato su priorità	L'applicazione real-time viene analizzata per attribuire appropriate <i>priorità</i> ai suoi processi. Durante l'esecuzione dell'applicazione viene adottato lo scheduling convenzionale basato su priorità.
Scheduling dinamico	Lo scheduling è eseguito quando <i>proviene</i> una richiesta di creazione di un processo. La creazione di un processo avviene solo se il requisito di risposta del processo può essere soddisfatto in maniera garantita.

Per ogni processo possono essere specificati due tipi di deadline:

- la scadenza d'inizio, cioè il minimo istante entro cui le operazioni del processo devono cominciare
- la scadenza di fine, cioè il tempo entro il quale le operazioni del processo devono terminare

Esistono vari modi per determinare le scadenze che prendono in considerazione diversi fattori: le precedenze del processo, la possibilità di eseguire operazioni di I/O per i processi, la disponibilità delle risorse, ecc.

Tra i vari algoritmi dedicati ai sistemi real-time ricordiamo:

1. Scheduling Earliest Deadline First (EFS)

TIPO: L'algoritmo EFS è un algoritmo non prelazionabile e statico. Adatto ai sistemi real-time soft.

FUNZIONAMENTO: Questa politica seleziona sempre il processo con la scadenza più breve. La scadenza ipotizzata di un processo viene calcolata in base ad una formula e questo processo non viene mai prelazionato poiché non si verificherà mai che superi la deadline settata dallo scheduler, per questo non ci sono prelazioni.

CONCLUSIONI: Presenta dei vantaggi tipo semplicità e rapidità poiché il carico dello scheduler è diminuito a causa della sua natura non preemptive. Inoltre, è una buona politica di scheduler statico in quanto i fattori che scandiscono i processi sono stabiliti a priori. L'unico svantaggio è che non si possono prevedere i processi che non rispettano la scadenza calcolata.

2. Scheduling Rate Monotonic (RM)

TIPO: L'algoritmo RMPO è un algoritmo di scheduling preemptive che assegna a ciascun task una priorità. Adatto ai sistemi real-time hard

FUNZIONAMENTO: Gestisce una serie di processi che vengono eseguiti periodicamente (ex. il processo x viene eseguito ogni 10ms sempre <1sec e occupa la CPU per 3ms, quindi la frazione del tempo che la usa è 0.3 ossia 10/3). Una volta determinato il rate al quale processo si deve ripetere, ossia il numero di ripetizioni

per secondo. Questo rate è proprio quello che va a scandire la priorità del processo in questione (un processo che ha periodo inferiore ha priorità più alta). E li esegue concorrentemente in base a questa.

CONCLUSIONI: Esegue così i processi periodici in modo concorrente e coinciso, sfruttando i tempi "morti" della CPU e incastrando i processi in modo ottimale massimizzando il lavoro.

UN DEADLOCK è quella situazione di stallo in cui un insieme di due o più processi attendono indefinitamente degli eventi (che potrebbero non verificarsi mai), ciascuno dei quali può essere generato solo da altri processi dell'insieme. Un deadlock si verifica quando due o più processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa. Un deadlock pregiudica il servizio utente, il throughput e l'efficienza delle risorse.

In un SO si possono verificare vari tipi di deadlock, come:

- il deadlock di sincronizzazione, dove un processo rimane in attesa di un'operazione di un altro processo che, però, non la effettua.
- il deadlock di comunicazione, dove un processo rimane in attesa di un messaggio da parte di un altro processo che, però, non lo invia.
- il deadlock generato da risorse dove un processo rimane in attesa di una risorsa, bloccata da un altro processo. In particolare, il kernel gestisce una tabella delle risorse per tener traccia dello stato di allocazione di una risorsa: quando un processo richiede una risorsa, se risulta già allocata, allora il processo va nello stato blocked altrimenti se risulta libera, gli viene allocata e il processo va nello stato ready.

Un deadlock di risorsa si verifica quando si accertano quattro condizioni Contemporaneamente:

- Condizioni di mutua esclusione: Almeno una delle risorse del sistema deve essere non condivisibile (ossia deve essere usata da un processo alla volta oppure essere libera)
- Condizione di possesso e attesa: (hold and wait) Un processo continua a tenere la risorsa allocata (che potrebbe servire ad altri processi) ma al tempo stesso, esso stesso, è in attesa di altre risorse per poter completare le sue operazioni
- Condizioni di assenza di prelazione: Una risorsa allocata a un processo non può essere rimossa in modo forzato da questo processo per poter essere assegnata ad un altro processo
- Condizione di attesa circolare: Esiste nel sistema una catena circolare dei processi, ognuno dei quali aspetta il rilascio di una risorsa da parte del processo che lo segue; ad esempio, il processo Pi aspetta Pj, Pj aspetta Pk e Pk aspetta Pi.

Più in generale la richiesta si divide in tre eventi:

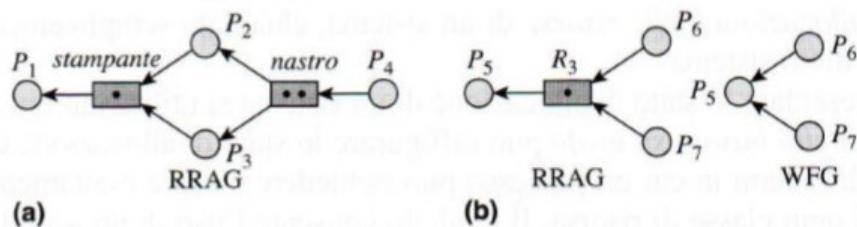
Evento	Descrizione
Richiesta	Un processo richiede una risorsa tramite una chiamata di sistema. Se la risorsa è libera, il kernel la alloca al processo immediatamente; altrimenti, cambia lo stato del processo a <i>blocked</i> .
Allocazione	Il processo diventa <i>holder</i> della risorsa a esso allocata. Le informazioni sullo stato della risorsa vengono aggiornate e lo stato del processo diventa <i>ready</i> .
Rilascio	Un processo rilascia una risorsa tramite una chiamata di sistema. Se vari processi sono bloccati sull'evento allocazione della risorsa, il kernel usa alcune regole, come l'allocazione FCFS, per decidere a quale processo allocare la risorsa.

Per gestire tutte le richieste da parte dei processi di risorse e se questi andranno a formare un deadlock, il SO risolve questo problema secondo due differenti strutture dati immagazzinando difatti lo stato di allocazione delle risorse di un sistema:

- **MODELLO BASATO SU GRAFI** si divide in due tipi:
 - Un grafo di richiesta e allocazione risorse (RRAG) contiene due tipi di nodi: processi, rappresentati da un cerchio e risorsa, rappresentati da rettangoli, ove all'interno di questi ci sono dei pallini che rappresentano quante risorse di quel tipo ci sono. Contiene due tipi di archi: allocazione, vanno da risorsa a processo e richiesta che vanno da processo a risorsa, indicando che il processo è bloccato per la richiesta.
 - Un grafo di attesa (WFG) rappresenta lo stato di allocazione in modo più conciso di un RRAG se ogni classe di risorsa nel sistema contiene solo un'unità di risorsa (un pallino). Il WFG contiene i nodi solo di un tipo, detti, nodi processo. Un arco nel WFG rappresenta il fatto che il processo Pi è bloccato su una

richiesta per una risorsa attualmente allocata al processo P_j (cioè il processo P_i sta attendendo che il processo P_j rilasci una risorsa).

Indipendentemente dal grafo considerato se non contiene cicli, nessun processo del sistema subisce uno stallo; mentre se un grafo contiene un ciclo può verificarsi un deadlock (dipende dal numero di istanze della risorsa).



La figura mostra un grafo RRAG e un grafo WFG equivalenti quando ogni classe di risorsa contiene soltanto una unità di risorsa. Come si può vedere il grafo RRAG richiede un arco in più rispetto al grafo WFG.

- MODELLO BASATO SU MATRICE si divide in due tipi:
 - Matrice delle risorse allocate indica quante unità di risorsa di ogni classe di risorse sono allocate ad ogni processo del sistema.
 - Matrice delle risorse richieste tiene conto delle richieste in attesa e, in pratica, indica quante unità di risorsa di ogni classe di risorsa sono state richieste da ogni processo del sistema

Se un sistema contiene n processi con r classi di risorsa, ognuna di queste matrici è una matrice $n \times r$. In alcuni casi vengono utilizzate anche tabelle ausiliarie per rappresentare informazioni aggiuntive.

Stampante Nastro		Stampante Nastro		Stampante Nastro	
P_i	0	1	P_i	1	0
P_j	1	0	P_j	0	1
P_k	0	1	P_k	0	0
Risorse allocate		Risorse richieste		Risorse totali	1
				Risorse libere	0

Per trattare adeguatamente le situazioni di stallo si possono impiegare tre diversi approcci Gestionali dei deadlock:

1. INDIVIDUAMENTO E RISOLUZIONE DEI DEADLOCK

Questo approccio permette che si abbia il deadlock, per poi provare a scoprire quando ciò accade ed infine, provare a risolvere la situazione di stallo dopo che si è verificata. Quindi serve un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo ed una tecnica per permettere al sistema di ricominciare a funzionare.

- a. Tramite matrici: per determinare se c'è un deadlock oppure no nel sistema, occorre provare a costruire sequenze di eventi in base al quale tutti i processi blocked possono avere le risorse che hanno richiesto. Se si riesce con successo a costruire tale sequenza allora nel sistema non c'è presenza di deadlock; altrimenti ci sono processi in condizione di stallo.

Esempio 8.6 – Individuazione dei deadlock

Si consideri il seguente stato di allocazione di un sistema con 10 unità di una classe di risorse R_1 e tre processi P_1, P_2 e P_3 :

R_1	R_1	Risorse totali	R_1
P_1 4	P_1 6		10
P_2 4	P_2 2		
P_3 2	P_3 0		
Risorse allocate	Risorse richieste	Risorse libere	0

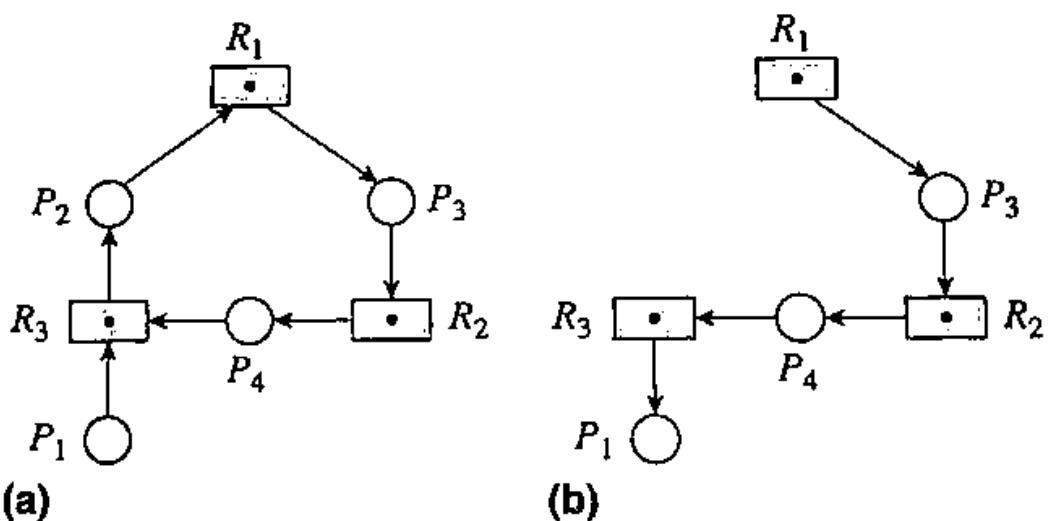
Il processo P_3 è in stato *running* perché non è bloccato da una richiesta di risorse. Tutti i processi nel sistema possono terminare nel seguente modo: il processo P_3 termina e rilascia 2 unità della risorsa a esso allocata. Queste unità possono essere allocate a P_2 . Quando questo termina, 6 unità della risorsa possono essere allocate a P_1 . Così, non c'è alcun processo *bloccato* quando termina la simulazione, quindi non c'è deadlock nel sistema.

Se le richieste dai processi P_1 e P_2 fossero per 6 e 3 unità, rispettivamente, nessuno di loro potrebbe terminare anche dopo che il processo P_3 ha rilasciato 2 unità di risorsa. Questi processi sarebbero in stato *blocked* al termine della simulazione, e quindi sarebbero in deadlock allo stato attuale del sistema.

- b. Tramite grafi: tra le condizioni necessarie per un deadlock c'è anche l'attesa circolare, che si manifesta in un ciclo nei modelli basati sui grafi, che rappresenta in alcuni sistemi una condizione più che sufficiente che si verifichi un deadlock (in particolare sui WFG). Sui RRAG un ciclo non è una condizione sufficiente per un deadlock nei sistemi a istanza multipla perché le classi di risorsa possono contenere diverse unità di risorsa. Affinché un processo sia in deadlock è necessario che tutti i processi che posseggono unità di una risorsa da esso richiesta, siano anch'essi in deadlock.

RISOLUZIONE

Supponiamo che siano stati individuati dei deadlock nel sistema. A questo punto è necessaria qualche tecnica che permetta al sistema di ricominciare a funzionare. Una delle tecniche più utilizzate è la tecnica della risoluzione del deadlock mediante prelazione. In pratica, se vi sono più processi in stato di deadlock, la risoluzione del deadlock mediante prelazione comporta l'interruzione dei deadlock per assicurare l'avanzamento di alcuni di questi processi. Ciò può essere realizzato interrompendo uno o più processi e allocando le loro risorse a qualche altro processo in modo tale che possa completare le sue operazioni. Ogni processo interrotto è detto vittima della risoluzione del deadlock. La scelta delle vittime viene fatta tramite criteri quali la priorità dei processi, le risorse già utilizzate, ecc.



In questo caso il processo P_2 , che causava il deadlock nella fig. a, è stato fatto vittima in fig. b

2. PREVENZIONE DEI DEADLOCK

In questo approccio si cerca di evitare i deadlock annullando una o più condizioni necessarie. In pratica, il kernel può usare una politica di allocazione delle risorse che assicuri che uno dei delle quattro condizioni necessarie non possa verificarsi:

- a. Per evitare la mutua esclusione: In un sistema non esisterebbero deadlock se tutte le risorse potessero essere rese condivisibili. Infatti, in questo modo, in un RRAG ci sarebbero solo archi di allocazione, quindi non si avrebbero mai attese circolari. Un problema è che alcune risorse sono non condivisibili, ma si può superare tale problema creando dispositivi virtuali, come ad esempio una stampante virtuale da allocare a vari processi. Un altro problema è che questo approccio non può funzionare per risorse software come file condivisi, che dovrebbero essere modificati in maniera mutuamente esclusiva per evitare race condition. (Quindi quasi Impossibile)
- b. Per evitare la non prelazione: Se le risorse sono rese prelazionabili, il kernel può assicurare che i processi abbiano tutte le risorse di cui necessitano, il che evita percorsi circolari nei RRAG. L'unico problema è che i dispositivi sequenziali di I/O non possono essere prelazionati. (Non praticabile)
- c. Per evitare hold and wait: Per evitare la condizione relativa al possesso e attesa, è necessario che un processo che abbia acquisito delle risorse non possa effettuare richieste di risorse, oppure che un processo bloccato su una richiesta di risorsa non possa poter impegnare altre risorse. In un RRAG, i percorsi che coinvolgono più di un processo non possono comparire, quindi non possono esistere percorsi circolari. Una semplice politica per l'implementazione di questo approccio è consentire a un processo di effettuare solo una richiesta di risorse durante la propria esecuzione in cui chiede tutte le risorse di cui necessita. (Plausibile)
- d. Per evitare l'attesa circolare: Un'attesa circolare può dipendere dalla condizione hold-and-wait, che a sua volta è conseguenza delle condizioni di mutua esclusione e assenza di prelazione; quindi, non si verifica se non si verifica nessuna di queste condizioni. Le attese circolari possono essere evitate separatamente non consentendo ad alcuni processi di attendere determinate risorse.
- e. La politica che sembra più efficace è l'allocazione globale e il ranking delle risorse, ossia:
 - I. Allocazione globale è la più semplice, espressa nell'evitare l'hold and wait questo approccio consente ad un processo di effettuare solo una richiesta di risorse durante la propria esecuzione in cui chiede tutte le risorse di cui necessita, anche quelle che vengono utilizzate dopo istanti successivi.
 - II. Ranking delle risorse funziona in modo differente, ogni classe di risorse possiede un numero identificativo chiamato rank. Quando un processo effettua una richiesta di una risorsa il kernel la concede se e solo se il rank della risorsa richiesta è maggiore del rank massimo delle risorse attualmente in uso dal processo, se sì, gli viene allocata, altrimenti va in blocked, se volesse avere la risorsa a tutti i costi basta rilasciare la/e risorse allocategli che hanno rango maggiore della risorsa richiesta. Funziona meglio se i processi fanno richieste di risorse in maniera ordinata e crescente (infattibile).

3. EVITARE I DEADLOCK

La politica di evitare i deadlock ammette una richiesta di risorsa solo se si riesce a stabilire che l'accettazione della richiesta non può condurre a un deadlock né immediatamente né successivamente. In caso contrario, la richiesta è messa in attesa finché non viene ammessa. Questa soluzione è possibile solo se il sistema è capace di mantenere delle informazioni sulle risorse disponibili al sistema e sulle risorse che ogni processo può potenzialmente richiedere. Come il kernel determina se può verificarsi successivamente un deadlock si utilizza un approccio conservativo di questo genere: ogni processo dichiara il massimo numero di unità di risorsa di ogni classe che può richiedere. Il kernel permette che un processo richieda queste unità di risorsa a stadi successivi, ossia, poche unità di risorsa alla volta, in base al massimo numero dichiarato dal processo, e utilizza una tecnica di analisi nel caso peggiore per controllare la possibilità di deadlock successivi. Questo approccio è conservativo in quanto un processo può terminare il suo funzionamento senza richiedere il massimo numero dichiarato di unità. Uno degli algoritmi che adotta tale approccio è l'algoritmo del banchiere. Tuttavia, per la maggior parte dei sistemi è impossibile conoscere in anticipo le risorse che richiederà un processo, per cui è spesso impossibile evitare del tutto i deadlock.

Algoritmo del Banchiere

USO: strutture di tipo array del tipo

Notazione	Significato
$Risorse_richieste_{j,k}$	Numero di unità della classe di risorsa R_k attualmente richieste dal processo P_j
$Risorse_massime_{j,k}$	Massimo numero di unità della classe di risorsa R_k di cui può aver bisogno il processo P_j
$Risorse_allocate_{j,k}$	Numero di unità della classe di risorsa R_k allocate al processo P_j
$Risorse_totali_allocate_k$	Numero totale di unità allocate della classe di risorsa R_k , ossia $\sum_j Risorse_allocate_{j,k}$
$Risorse_totali_k$	Numero totale di unità della classe di risorsa R_k appartenenti al sistema

TIPO: è un algoritmo di scheduling utilizzato per prevenire i deadlock nell'allocazione delle risorse. In particolare, questo algoritmo può indicare se un sistema (in particolare un SO) si venga a trovare in uno stato sicuro (tutti i processi avranno termine) o meno (non c'è garanzia che abbiano termine) nel caso assegnasse una risorsa ad uno dei processi richiedenti.

FUNZIONAMENTO: Le tecniche per evitare i deadlock sono implementate trasferendo il sistema da uno stato sicuro a un altro stato sicuro come descritto di seguito:

1. quando un processo effettua una richiesta, si calcola il nuovo stato in cui si troverebbe il sistema se la richiesta fosse ammessa. Chiameremo questo stato, stato proiettato;
2. se lo stato proiettato è uno stato sicuro, si ammette la richiesta aggiornando le matrici $Risorse_allocate$ e $Risorse_totali$; altrimenti, si tiene la richiesta in attesa;
3. quando un processo rilascia una o più risorse o termina la propria esecuzione, si esaminano tutte le richieste in attesa e si allocano quelle che porterebbero il sistema in un nuovo stato sicuro.

Tutto ciò può essere realizzato usando un algoritmo di individuazione dei deadlock, però apportando una modifica: il completamento di un processo P , sia running che blocked, può richiedere ($Risorse_massime - Risorse_allocate$) ulteriori unità di risorse di ogni classe di risorsa. Quindi l'algoritmo controlla che per ogni classe di risorsa:

$$Risorse_totali - Risorse_totali_allocate \geq Risorse_massime(P) - Risorse_allocate(P)$$

Se questa condizione è soddisfatta, viene simulato il completamento del processo P . E poi, viene controllato se qualche altro processo può soddisfare questa condizione e così via.

Quando un processo effettua una nuova richiesta, viene inserita nella matrice $Risorse_richieste$, che memorizza le richieste in attesa di tutti i processi e viene invocato l'algoritmo con l'identificativo del processo richiedente. Quando un processo rilascia alcune risorse allocate per sé oppure termina le proprie operazioni, viene invocato l'algoritmo per ogni processo, la cui richiesta è in attesa.

L'algoritmo può essere descritto come segue:

1. dopo alcune inizializzazioni al passo 1,
2. l'algoritmo simula l'accettazione della richiesta al passo 2 calcolando lo stato proiettato.
3. Il passo 3 controlla se lo stato proiettato è fattibile, ossia se esistono sufficienti risorse libere per permettere l'accettazione della richiesta.
4. Per controllare se lo stato proiettato è uno stato sicuro, si controlla se il massimo numero di risorse di cui necessita ogni processo attivo (running o blocked) può essere soddisfatto allocando alcune delle risorse libere.
 - a. In caso affermativo, l'algoritmo simula il suo completamento cancellandolo dall'insieme dei processi active e rilasciando le risorse allocate per esso. Questa azione viene ripetuta finché non è possibile cancellare più processi dall'insieme active.
 - b. Se al termine di questo passo l'insieme active è vuoto, lo stato proiettato è uno stato sicuro, quindi l'algoritmo cancella la richiesta dalla lista delle richieste in attesa e alloca le risorse richieste. Questa azione non sarà eseguita se lo stato proiettato non è né fattibile né sicuro, per cui la richiesta rimane in attesa.

La complessità computazionale di questo algoritmo è $O(n^2m)$ dove n è il numero di processi ed m è il numero di tipi di risorse (per ogni tipo possono essere disponibili più risorse).

Algoritmo 8.2 Algoritmo del banchiere**Input**

<i>n</i>	:	numero di processi;
<i>r</i>	:	numero di classi di risorse;
<i>Blocked</i>	:	insieme di processi;
<i>Running</i>	:	insieme di processi;
<i>P_{processo_richiedente}</i>	:	Processo che effettua la nuova richiesta di risorsa;
<i>Risorse_massime</i>	:	array [1..n, 1..r] di integer;
<i>Risorse_allocate</i>	:	array [1..n, 1..r] di integer;
<i>Risorse_richieste</i>	:	array [1..n, 1..r] di integer;
<i>Risorse_totali_allocate</i>	:	array [1..r] di integer;
<i>Risorse_totali</i>	:	array [1..r] di integer;

Strutture dati

<i>Active</i>	:	insieme di processi;
<i>fattibile</i>	:	boolean;
<i>Nuova_richiesta</i>	:	array [1..r] di integer;
<i>Allocazione_simulata</i>	:	array [1..n, 1..r] di integer;
<i>Risorse_totali_allocate_simulate</i>	:	array [1..r] di integer;

1. $Active = Running \cup Blocked;$
for $k = 1..r$
 $Nuova_richiesta[k] = Risorse_richieste[processo_richiedente, k];$
2. $Allocazione_simulata := Risorse_allocate;$
for $k = 1..r$ /* Calcolare lo stato di allocazione proiettato */
 $Allocazione_simulata[processo_richiedente, k] :=$
 $Allocazione_simulata[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate_simulate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$
3. $fattibile = true;$
for $k = 1..r$ /* Controllare se lo stato di allocazione proiettato è fattibile */
if $Risorse_totali[k] < Risorse_totali_allocate_simulate[k]$ **then** $fattibile = false;$
4. **if** $fattibile = true$
then /* Controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro */
while l'insieme Active contiene un processo P_i tale che
Per ogni k , $Risorse_totali[k] - Risorse_totali_allocate_simulate[k] \geq Risorse_massime[i, k] - Allocazione_simulata[i, k]$
Rimuovi P_i da Active;
for $k = 1..r$
 $Risorse_totali_allocate_simulate[k] :=$
 $Risorse_totali_allocate_simulate[k] - Allocazione_simulata[i, k];$
5. **if** l'insieme Active è vuoto
then /* Lo stato di allocazione proiettato è uno stato di allocazione sicuro */
for $k = 1..r$ /* Cancellare la lista dalle richieste in attesa */
 $Risorse_richieste[processo_richiedente, k] := 0;$
for $k = 1..r$ /* Accettare la richiesta */
 $Risorse_allocate[processo_richiedente, k] :=$
 $Risorse_allocate[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$

Il message passing è un modo con il quale i processi interagiscono tra loro. I processi possono esistere nello stesso computer o in computer diversi connessi da una rete.

Message passing usato in diverse applicazioni:

- Paradigma client-server
- Backbone di protocolli di comunicazione di livelli più alti
- Programmi paralleli e distribuiti

Sfrutta le SystemCall:

```
send (<destination_process>, <message_length>, <message_address>);
receive (<source_process>, <message_area>);
```

```

begin
Parbegin
    var buffer : . . . ;
repeat
    { Produce in buffer }
    send (Pj, buffer);
    { Remainder of the cycle }
forever;
Parend;
end.

```

Process P_i

```

var message_area : . . . ;
repeat
    receive (Pi, message_area);
    { Consume from message_area }
    { Remainder of the cycle }
forever;

```

Process P_j

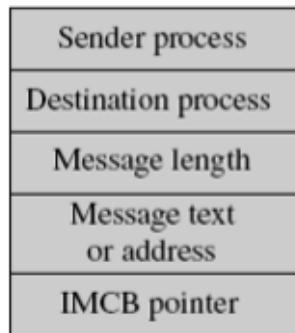
Del message passing esistono principalmente due metodologie applicanti riguardante il source_process e destination_process:

- naming diretto semplice: i processi mittente e destinatario si citano a vicenda.
- naming diretto simmetrico, entrambi i processi specificano il nome dell'altro.
- naming diretto asimmetrico, il destinatario non indica il nome del processo da cui intende ricevere un messaggio.
- naming indiretto, i processi non menzionano i nomi l'uno dell'altro.

La SystemCall receive blocca sempre il processo chiamante, mentre per la send per le metodologie utilizzate sono:

- send bloccante: blocca il processo mittente fino alla consegna del messaggio al processo destinatario, poi passarlo a ready una volta ricevuto. Lo scambio di messaggi avviene in modo sincrono, semplifica difatti la progettazione dei processi concorrenti.
- send non bloccante: permette al mittente di continuare le proprie operazioni dopo la chiamata, non lo blocca mai. Lo scambio dei messaggi avviene in maniera asincrona però Incrementa la concorrenza tra mittente e destinatario. In questo caso bisogna analizzare il processo di message passing un modo differente:

Se un processo P_i invia un messaggio a P_j usando una send non bloccante il kernel costruisce un IMCB (interprocess message control block) per memorizzare tutte le informazioni necessarie per consegnare il messaggio (a cui è stato allocato un buffer nel kernel). Quando P_j invoca receive, il kernel copia il messaggio dallo IMCB nell'area fornita da P_j. Ovviamente se ho più send provenienti da più processi avrò una lista FCFS di IMCB. In particolare, la IMCB è composta da:



Nel naming simmetrico, è usata una lista separata per ogni coppia di processi P_i-P_j che comunica.

Nel naming asimmetrico, è usata una singola lista per ogni processo. Quando un processo esegue receive, il primo IMCB nella sua lista è elaborato per consegnare il messaggio.

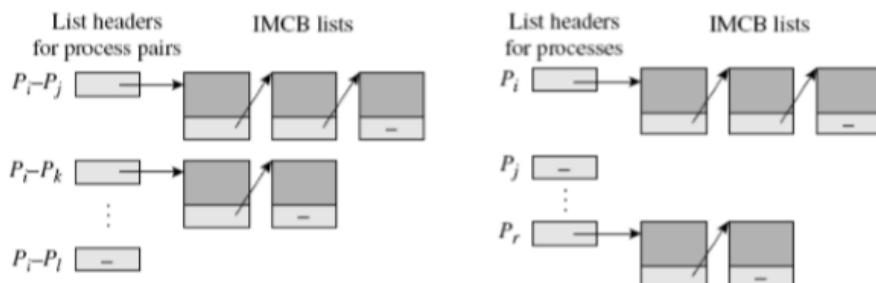
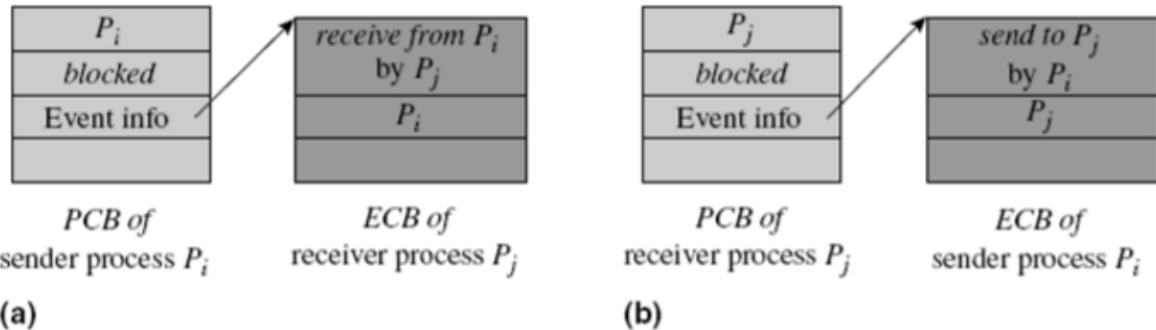


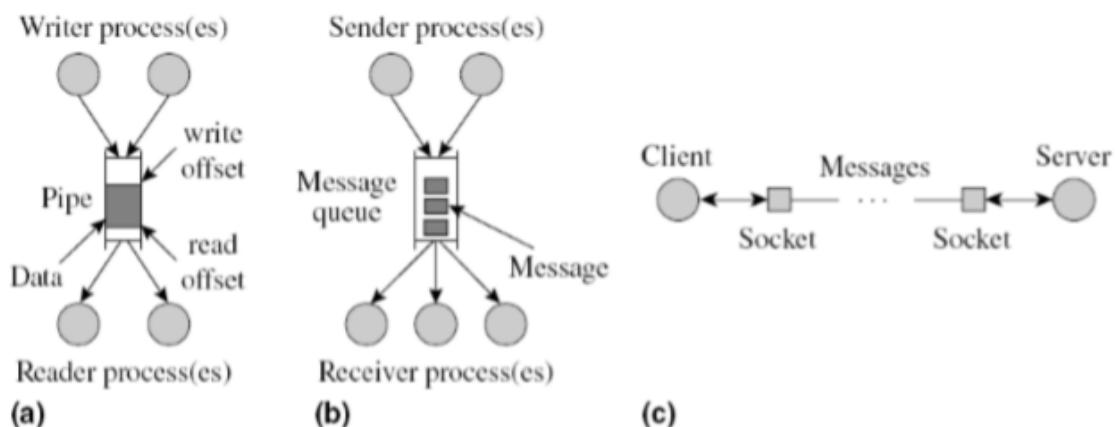
Figura mostra naming simmetrico (sx) e naming asimmetrico (dx)

Ricorda che nello scambio di messaggi da P_i a P_j il kernel usa un Event Control Block (ECB) per annotare le azioni da intraprendere quando si verifica un evento. Per esempio, nel naming simmetrico:



- a) Quando P_i invoca send il kernel controlla se esiste un ECB per tale chiamata, cioè, se P_j ha fatto una chiamata a receive ed era in attesa che P_i inviasse un messaggio allora lo invia. Altrimenti, il kernel sa che receive si verificherà in futuro, quindi crea un ECB per l'evento «ricevi da P_i » e specifica P_i come il processo coinvolto nell'evento. (P_i è eventualmente messo allo stato blocked o no) e l'indirizzo dell'ECB è messo nel campo evento del suo PCB.
- b) Se P_j fa una chiamata a receive prima che P_i invochi send viene creato un ECB per «invia a P_j da P_i ». L'id di P_j è messo nell'ECB indicando che lo stato di P_j sarà modificato quando si verifica l'evento send.

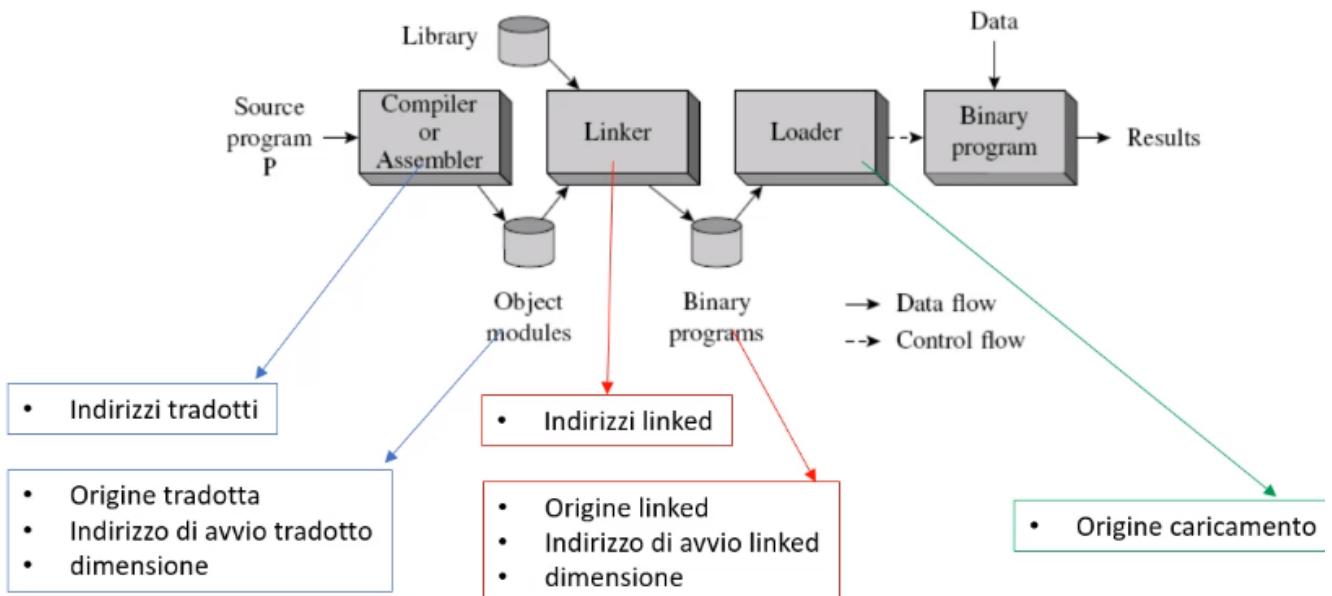
Esistono principalmente 3 supporti alla comunicazione interprocesso:



1. Pipe: Meccanismo FIFO per il trasferimento di dati (che possono essere letti una sola volta e poi cancellati) tra processi chiamati lettori e scrittori. Esistono due tipi di pipe:
 - a. Pipe con nome può essere utilizzata da processi indipendenti dall'antenato formante.
 - b. Pipe senza nome può essere usata solo da processi che appartengono allo stesso antenato.
2. Code di messaggi (message queue): Usate dai processi nel dominio del sistema Unix. I permessi di accesso indicano quali processi possono inviare o ricevere messaggi
3. Socket: un'estremità di un percorso di comunicazione. Può essere usata per impostare dei percorsi di comunicazione tra processi nel dominio Unix e all'interno di alcuni domini Internet

La memoria di un computer è condivisa da un grande numero di processi, per cui la gestione della memoria è tradizionalmente una parte molto importante di un SO.

L'aspetto principale delle prestazioni riguarda l'inserimento di più processi in memoria, in modo da migliorare sia le prestazioni del sistema che il servizio per l'utente. Per mantenere in memoria un elevato numero di processi, il kernel può decidere di mantenere in memoria anche solo una parte dello spazio di indirizzamento di ogni processo. A tal fine si utilizza la parte della gerarchia della memoria chiamata memoria virtuale che si compone della memoria RAM e dell'hard disk. Le parti dello spazio di indirizzamento di un processo non presenti in memoria vengono caricate dal disco quando necessario.



La figura mostra uno schema delle tre trasformazioni eseguite sul programma P prima di poter essere caricato in memoria per l'esecuzione:

- Compilazione: durante la compilazione, le istruzioni del codice sorgente sono tradotte in istruzioni macchina. Viene creato il modulo oggetto.
- Linking: durante la fase di linking, il codice delle librerie viene incluso nel modulo oggetto. In pratica, vengono linkati al programma le funzioni che esso utilizzerà. Viene creato il codice binario.
- Caricamento: durante il caricamento, il codice binario viene caricato in memoria per poter essere eseguito dalla CPU.

La differenza tra linker e loader è diventata sempre meno netta nei moderni SO. Tuttavia, usiamo i termini nel seguente modo: un linker collega insieme i moduli per formare un programma eseguibile. Un loader carica un programma o una parte di esso in memoria per l'esecuzione.

- Nel linking statico il linker collega tutti i moduli di un programma prima che cominci la sua esecuzione. Se più programmi usano lo stesso modulo di una libreria, ogni programma riceverà una propria copia del modulo, quindi diverse copie del modulo potranno essere presenti in memoria allo stesso tempo se i programmi che usano il modulo vengono eseguiti simultaneamente.
- Nel linking dinamico viene eseguito durante l'esecuzione di un programma binario. Il linker viene invocato quando, durante l'esecuzione, si incontra un riferimento esterno non assegnato. Il linker collega il riferimento esterno e riprende l'esecuzione del programma. Presenta molti vantaggi: i moduli non invocati durante l'esecuzione non vengono linkati; se un modulo è usato da più programmi è presente una sola volta in memoria; se si aggiorna una libreria di moduli, il programma utilizzerà automaticamente la nuova versione del modulo.

Un'operazione molto importante nella gestione della memoria è il binding, ovvero il processo tramite cui si associano gli indirizzi di memoria alle entità di un programma. Un'entità di un programma, per esempio una funzione o una variabile, ha un insieme di attributi e ogni attributo ha un valore; il binding consiste nello specificare il valore di un attributo. Per esempio, una variabile in un programma ha attributi come il nome, il tipo, la dimensione. Un binding del nome specifica il nome della variabile e un binding del tipo specifica il suo tipo. (Un programma P scritto in linguaggio L prima di essere eseguito deve subire varie trasformazioni. In particolare, deve essere compilato poi linkato poi eseguito. Ognuna di queste trasformazioni effettua il collegamento delle istruzioni e i dati del programma a un nuovo insieme di indirizzi).

Questa operazione può essere effettuata in 3 momenti diversi:

1. Durante la compilazione
Il binding viene eseguito in questo momento quando si sa dove il processo risiederà in memoria. Gli indirizzi che vengono assegnati saranno gli stessi ad ogni esecuzione del programma. Il codice generato prende il nome di codice assoluto. Se la posizione in memoria del processo dovesse cambiare, sarebbe necessaria la ricompilazione.
2. Durante il caricamento

La posizione in memoria del processo è fissa, ma è nota solo quando viene lanciata l'applicazione (non quando è compilata). Il codice generato è detto codice rilocabile. E' il loader che effettua la rilocazione.

3. Durante l'esecuzione

La posizione in memoria del processo può variare durante l'esecuzione, quindi il programma può essere spostato da una zona all'altra della memoria durante l'esecuzione.

Il momento esatto nel quale viene eseguito il binding può determinare l'efficienza e la flessibilità con cui l'entità del programma può essere utilizzata. In generale possiamo distinguere:

- binding statico è un binding eseguito prima dell'esecuzione di un programma (durante la compilazione o durante il caricamento).
- Il binding dinamico è eseguito durante l'esecuzione di un programma.

Se gli indirizzi sono generati nelle fasi di compilazione e di caricamento, allora indirizzi logici e indirizzi fisici corrispondono, per cui non è necessario effettuare la traduzione degli indirizzi. Se gli indirizzi sono generati durante l'esecuzione, allora è necessaria la traduzione degli indirizzi.

Se gli indirizzi logici non coincidono con quelli fisici. In questo caso ci si riferisce agli indirizzi logici col termine di indirizzi virtuali. Per ovviare al fatto che a volte gli indirizzi logici non hanno corrispondenza con quelli fisici, si fa uso della rilocazione per ovviare al problema. Esistono due tipi di rilocazione:

- Rilocazione Statica

viene eseguita prima che abbia inizio l'esecuzione del programma, qui il kernel gli assegna l'area di memoria abbastanza grande da contenere il programma e chiama il loader con il nome del programma e l'origine di caricamento con i dovuti parametri. Questa politica permette di risparmiare l'overhead dovuto alla traduzione degli indirizzi, ma non è prestante in quanto non consente di cambiare l'area di memoria allocata al programma.

- Rilocazione Dinamica

viene effettuata durante l'esecuzione del programma quando il kernel necessita di spostarla. Questa politica consente di cambiare l'area di memoria allocata al programma ma soffre di overhead: può essere effettuata sospendendo l'esecuzione del programma, eseguendo la rilocazione e riprendendo l'esecuzione del programma. Ma poiché il programma è stato fisicamente spostato, quindi gli indirizzi logici e quelli fisici non combaciano si fa uso dei REGISTRI DI RILOCAZIONE, ossia dei registri attivati durante la rilocazione dinamica, questi consentono la correzione della traduzione in questo modo:

Indirizzo di memoria effettivo = indirizzo di memoria utilizzato nell'istruzione corrente + contenuto del registro di rilocazione

Esempio:

Un programma X va in esecuzione, il kernel gli assegna l'area di memoria adatta ad esso, chiama il loader e lo alloca (rilocazione statica) all'indirizzo 50000 , durante l'esecuzione però viene spostato all'indirizzo 70000 (rilocazione dinamica) in questo caso il registro di rilocazione calcola l'indirizzo di memoria effettivo come: 70000-50000=20000, questo valore va aggiunto all'indirizzo dell'istruzione del programma X: un'istruzione che si trovava all'indirizzo 55234 ora si troverà all'indirizzo 75234 del programma rilocalizzato.

la maggior parte dei moderni SO consentono a diversi processi di girare contemporaneamente. In questo caso si parla di multiprogrammazione.

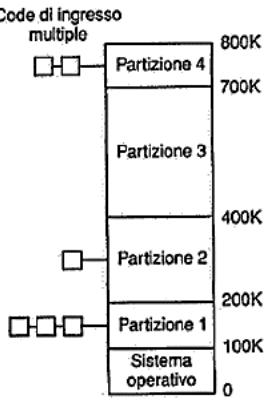
Essa consente di avere diversi processi in memoria; questo comporta che se, ad esempio, un processo è bloccato in attesa di I/O, un altro processo può utilizzare la CPU. Esistono principalmente 2 tipi di gestione della memoria per realizzare la multiprogrammazione:

- Partizionamento fisso (non più usato)

Il modo più semplice per realizzare la multiprogrammazione è quello di dividere la memoria in n partizioni fisse di diversa dimensione. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato al numero di partizioni. Dal momento che le partizioni sono fisse, tutto lo spazio di una partizione non usato dal processo viene sprecato.

In questo tipo di multiprogrammazione vengono utilizzati due approcci per le code di processi che portano all'allocazione dei processi nelle varie partizioni della memoria.

CODE SEPARATE

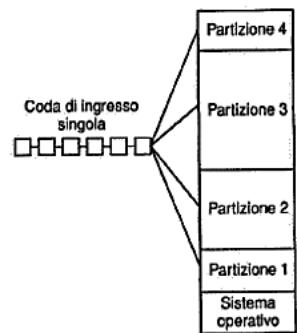


In questo approccio, ciascuna partizione ha una propria coda di ingresso. All'avvio del sistema viene effettuata la suddivisione della memoria in partizioni fisse.

Quando arriva un processo, viene messo nella coda di ingresso della partizione più piccola che lo può contenere. Dal momento che le partizioni sono fisse, in questo schema, tutto lo spazio di una partizione non usato dal processo viene sprecato.

Lo svantaggio dell'organizzare i processi in ingresso in code separate diventa evidente nel caso in cui la coda per una partizione grande sia vuota ma quella per una partizione piccola sia piena, come nel caso delle partizioni 1 e 3 dell'immagine a sinistra: qui i piccoli processi devono aspettare per essere inseriti in memoria, sebbene la maggior parte della stessa sia libera.

CODA UNICA



In questo approccio, tutte le partizioni hanno una singola coda di ingresso "comune". La partizione della memoria viene sempre effettuata all'avvio del sistema.

Ogni qualvolta una partizione diventa libera, vi viene caricato il processo più vicino alla testa della coda che può entrare nella partizione, ed esso è quindi mandato in esecuzione.

Dal momento che non è desiderabile sprecare una partizione molto grande per far girare un processo piccolo, un'altra strategia è quella di cercare in tutta la

coda in ingresso ogni volta che si libera una partizione, e scegliere il job più grande che può entrarci. Questo algoritmo dà poca priorità ai processi piccoli, anche se di solito è preferibile dare a quest'ultimi una priorità alta.

Una possibile soluzione è quella di avere sempre a disposizione una partizione piccola che permette ai processi più piccoli di girare senza dover allocare per loro una partizione grande.

- Partizionamento dinamico

Con il partizionamento dinamico, il numero, la locazione e la dimensione delle partizioni variano dinamicamente; quando un processo è caricato in memoria centrale, gli viene allocata tanta memoria quanta ne richiede. La flessibilità delle partizioni variabili migliora l'utilizzo della memoria, cioè non si hanno partizioni troppo piccole o troppo grandi come nel partizionamento statico.

Il problema delle partizioni variabili è che complicano rispetto alle partizioni fisse la procedura per tenere traccia dell'allocazione e la deallocazione della memoria.

ALLOCAZIONE DELLA MEMORIA DI UN PROCESSO si intende all'allocazione dello stesso processo in memoria e anche tenendo conto ai dati e richieste di memoria che vuole. In particolare, intendiamo in un processo l'area stack e l'area heap che vengono utilizzate durante l'uso di funzioni (stack) malloc, calloc o new (heap) che gestiscono una quantità di dati dinamici controllati dal programma in un'area di memoria definita durante l'esecuzione. Questi dati sfruttati dai programmi vengono definiti PCD controllati dalla struttura heap.

- L'area stack

eseguito come LIFO serve durante l'esecuzione del programma per coadiuvare le chiamate a funzione (che se gestite interamente nell'intero blocco main sarebbe molto complesso ed esageratamente esaustivo). Quando viene chiamata una funzione viene caricato lo stack frame che viene inserito nello stack contenente gli indirizzi o i valori dei parametri e l'indirizzo di ritorno. Durante l'esecuzione della funzione il supporto run-time crea i dati locali della funzione all'interno dello stack frame e al termine dell'esecuzione la funzione ritorna all'indirizzo specificato.

- L'area heap

viene utilizzate dal programma consentendolo di allocare e deallocare la memoria in ordine casuale. Si fa uso dell'heap per gestire i dati PCD di un processo. L'allocatore heap deve riutilizzare e sfruttare le aree di memoria libere per soddisfare le future richieste di memoria.

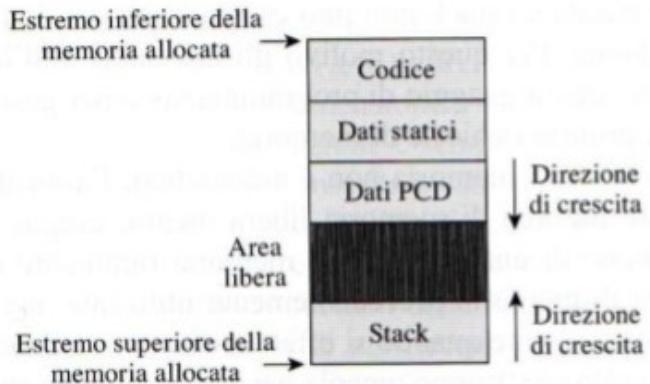
Quando si crea un processo o quando lo si carica (swap-in) in memoria, bisogna determinare quanta memoria deve essere allocata al processo stesso. Se si utilizza un'allocazione statica, prima che il programma venga eseguito, si

ipotizza la dimensione massima che può raggiungere il processo e la si alloca. Questo approccio può causare sprechi di memoria.

Nei moderni SO si utilizza un'allocazione dinamica. Viene allocata tanta memoria quanta realmente ne occorre al processo. Però, durante l'esecuzione di un processo, le strutture dati dello stesso possono crescere.

Le componenti da tener presente per la quantità di memoria da allocare a un processo sono:

- codice e dati statici del programma
- segmento heap (di dati PCD, dati dinamici controllati da programma)
- segmento stack (gestore delle chiamate a funzioni)



NOTA: l'heap e lo stack quando necessitano di più spazio si espandono verso l'area libera.

Per frammentazione intendiamo l'inutilizzo degli spazi di memoria non utilizzata. Si differenziano:

- frammentazione esterna: abbiamo un blocco e ne allochiamo solo una parte per fare una richiesta e il resto lo lasciamo libero, in questo caso avremmo una piccola parte della memoria che resterà inutilizzata in quanto non esiste processo così piccolo.
- frammentazione interna: quando andiamo a destinare un blocco di un processo maggiore rispetto a quella richiesta (solitamente si verifica quando ci sono schemi di allocazione che per minimizzare l'overhead di allocazione vanno ad allocare a ciascuna richiesta blocchi di dimensione fissa).

Quando la memoria viene assegnata dinamicamente, il SO deve gestirla. In generale, ci sono due modi per tenere traccia dell'utilizzo della memoria: i bitmap (mappe di bit) e le liste libere.

- Tecniche di allocazione mediante Bitmap (non più usate)

Con una mappa di bit, la memoria viene divisa in unità di allocazione che possono essere lunghe solo poche word o arrivare a qualche kilobyte; a ciascuna delle unità di allocazione viene associato un bit della mappa, che vale 0 se l'unità è libera e 1 se è occupata. Le dimensioni delle unità di allocazione rappresentano una scelta di progettazione importante. Più piccola è l'unità di allocazione più grande è la mappa di bit; tuttavia anche con un'unità di allocazione di soli 4 byte, 32 bit in memoria richiedono solo 1 bit nella mappa di bit. Una mappa di bit di 32n bit userà n bit nella mappa di bit, così la mappa di bit prenderà solo 1/33 della memoria. Se si sceglie un'unità di allocazione grande, allora la mappa di bit è piccola, ma viene sprecata una quantità significativa di memoria nell'ultima unità quando la dimensione dei processi non è un multiplo esatto dell'unità di allocazione. Il problema principale, quando si utilizza l'allocazione contigua, sta nel fatto che quando si decide di caricare un processo di k unità il gestore della memoria deve esaminare la mappa di bit per trovare una sequenza consecutiva di k zeri consecutivi. Questa è un'operazione lenta per cui le bitmap sono poco utilizzate.

- Tecniche di allocazione mediante Free-list

Quando un processo completa la propria esecuzione o rilascia la memoria allocata, il kernel riutilizza la memoria per soddisfare le richieste di altri processi. Quando si fa uso del partizionamento della memoria, il SO conserva una tabella nella quale sono indicate le partizioni di memoria disponibili e quelle occupate. In pratica, il SO usa questa tabella per tener traccia dello stato della memoria.

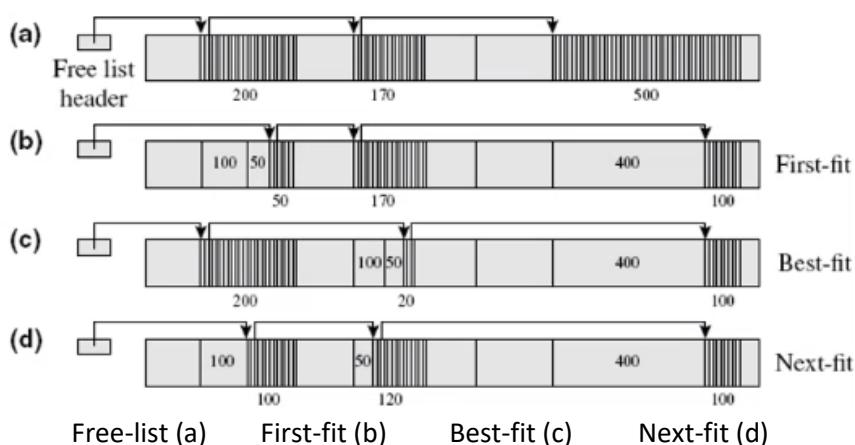
Questa gestione della memoria è chiamata gestione della memoria con liste concatenate ove inizialmente tutta la memoria è a disposizione dei processi utente; si tratta di un grande blocco di memoria disponibile, un buco. Ma col passare del tempo, vengono eseguiti molti processi, dunque, solitamente, in memoria sono sparsi un insieme di buchi di diverse dimensioni.

Il kernel utilizza tre funzioni per garantire il riuso efficace della memoria.

1. Esso mantiene una free list contenente le informazioni relative a ogni area di memoria libera. Quando un processo libera parti di memoria, le informazioni relative alla memoria liberata vengono inserite nella free list. Quando un processo termina, ogni area di memoria a esso allocata e le informazioni relative vengono inserite nella free list.
2. Seleziona un'area di memoria per l'allocazione. Quando viene effettuata una nuova richiesta di memoria, il kernel seleziona l'area di memoria più adatta per soddisfare la richiesta.
3. Unisce le aree di memoria libere. Due o più aree di memoria libere contigue possono essere unite per formare un'unica area libera più grande. Le aree da unire sono rimosse dalla free list e viene inserita al loro posto l'area più grande creata.

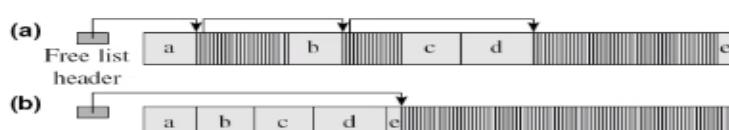
Come detto, al tempo t esistono in memoria n buchi di differenti dimensioni. Quando un processo è in attesa di memoria, esistono quattro tecniche per scegliere un buco libero tra quelli disponibili:

1. First-fit
questo è l'algoritmo più semplice nel quale si assegna al processo il primo buco abbastanza grande da contenerlo a partire dall'inizio della lista. Quindi le dimensione dell'area di memoria (buco) sono maggiori delle dimensioni richieste dal processo; questo algoritmo divide l'area di memoria in due parti, parte viene assegnata al processo, la restante viene reinserita nella free list.
2. Next-fit
questo algoritmo è un caso particolare di First-fit. La differenza è che in questo algoritmo si assegna al processo il primo buco abbastanza grande da contenerlo a partire dal punto in cui era terminata la ricerca precedente.
3. Best-fit
questo algoritmo assegna al processo il più piccolo buco capace di contenerlo. Si può facilmente intuire che bisogna ricercare il buco scorrendo completamente la free list. Allocchiamo di quel blocco/buco solo la quantità richiesta.
4. Worst-fit (non trattato)
questo algoritmo assegna il buco più grande al processo. Anche qui si deve esaminare l'intera lista. Lo scopo di questo algoritmo è quello di produrre buchi più grandi, cercando di evitare la frammentazione esterna.

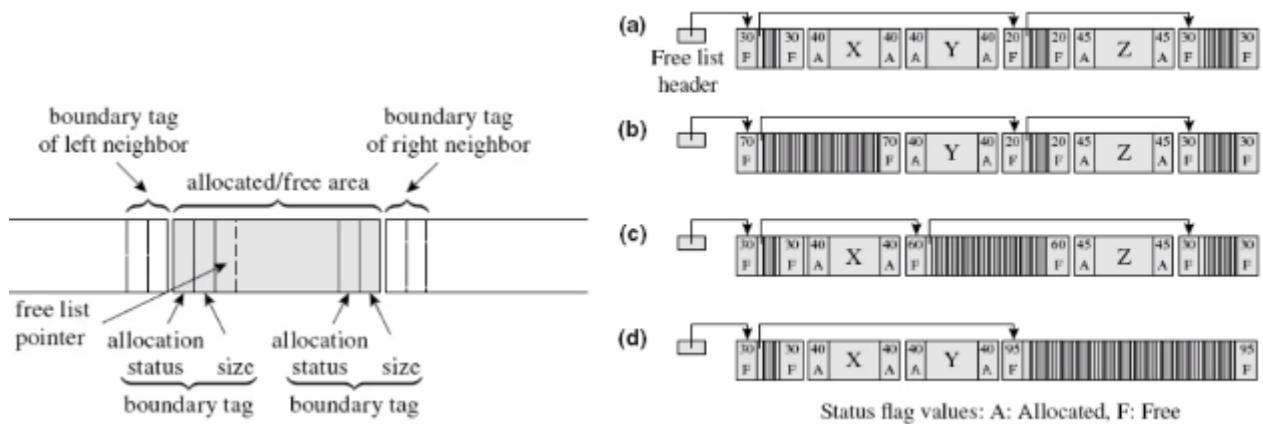


Come è possibile limitare la frammentazione della memoria entrano 5 strategie:

1. Unione delle aree di memoria libere contigue
è molto costosa in quanto devo scorrere le aree di memoria contigue per verificare. Risulta essere molto inefficiente per l'overhead.
2. Compattazione della memoria
controlla le aree di memoria occupate e le contengo in un unico blocco di processi attiva (a sx) e contengo tutta la memoria libera (a dx). Risulta essere una strategia inefficiente per l'overhead.



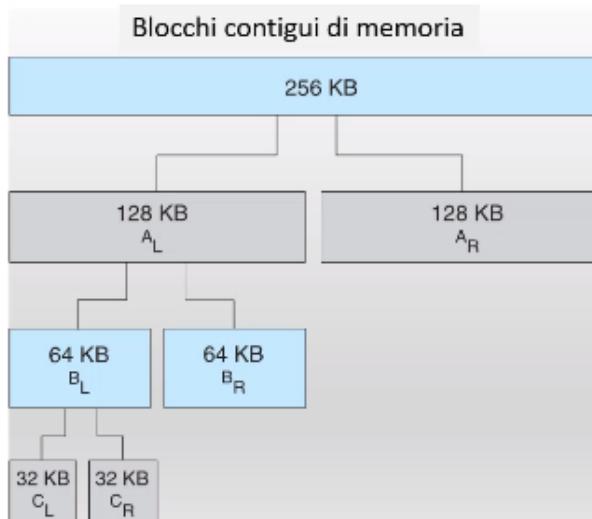
3. Boundary tag, ciascuna area di memoria ha posti agli estremi i descrittori di pochi byte che identifica f = free (libera) ed a = allocated (allocata) dell'area delimitata. Per gestire la frammentazione con questo prototipo quando le aree sono appena liberate controlla le aree quelle adiacenti (controlla sia a sx che a dx) liberi e otteniamo un blocco più grande con tag f. Esempio:



4. Allocatori di potenze di 2

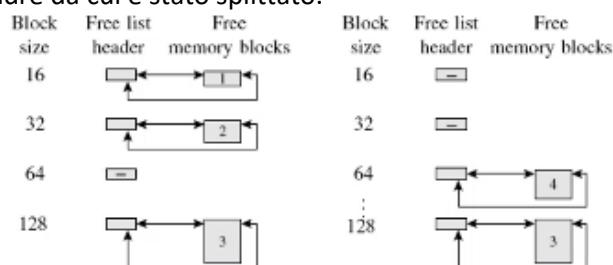
suddivide la memoria di blocchi a potenza di 2 e manteniamo free-list separate per ogni blocco spartito.

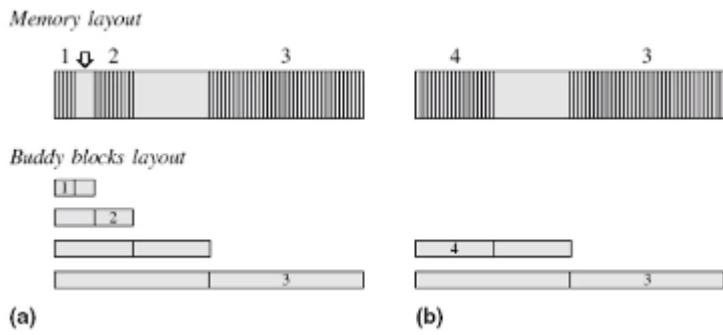
FUNZIONAMENTO: si comincia con generare dei blocchi di dimensioni scelte e ne mantengo una free-list separata i nuovi blocchi che possono essere creati dinamicamente se per esempio l'allocatore non ha più blocchi di quella dimensione per una richiesta e lo creo. Alloco un intero blocco per una richiesta, non faccio nessuna suddivisione e chiaramente non faccio nessuno sforzo della fusione di un blocchi continui, poiché lo inserisco nella free-list che gestisce tutti i blocchi di dimensione k.



5. Buddy system e allocatori potenza del 2 eseguono l'allocazione di memoria in blocchi dimensioni predefinite. Questa caratteristica porta alla frammentazione interna poiché parte della memoria in ogni blocco allocato può essere sprecata. Tuttavia, consente all'allocatore di mantenere free list separate per blocchi di dimensioni differenti. Questa organizzazione evita ricerche costose nella free list e porta ad allocationi e deallocationi veloci.

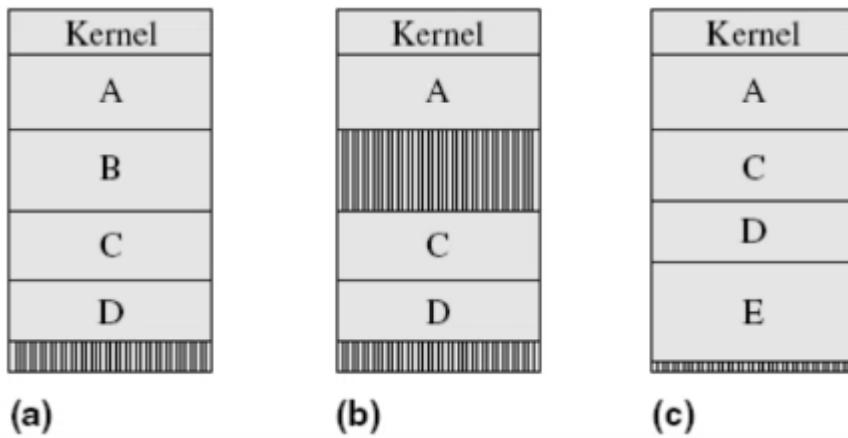
FUNZIONAMENTO: supponiamo di avere un blocco a potenza di 2 (2^p) e lo suddivido in 2 parti fino alle dimensioni opportune. Quando arriva una richiesta di memoria, divido il blocco di memoria in 2 fino a che non rispetta le dimensioni richieste dal processo e lo alloca al processo richiedente, il suo blocco fratello è libero e viene definito blocco buddy. Quando dealloco, controllo se il blocco fratello è libero e li unisco per ottenere la dimensione del blocco padre da cui è stato spartito.





I tipi di allocazioni, oltre a differenziarsi per strategie, si possono anche suddividere in due macroaree:

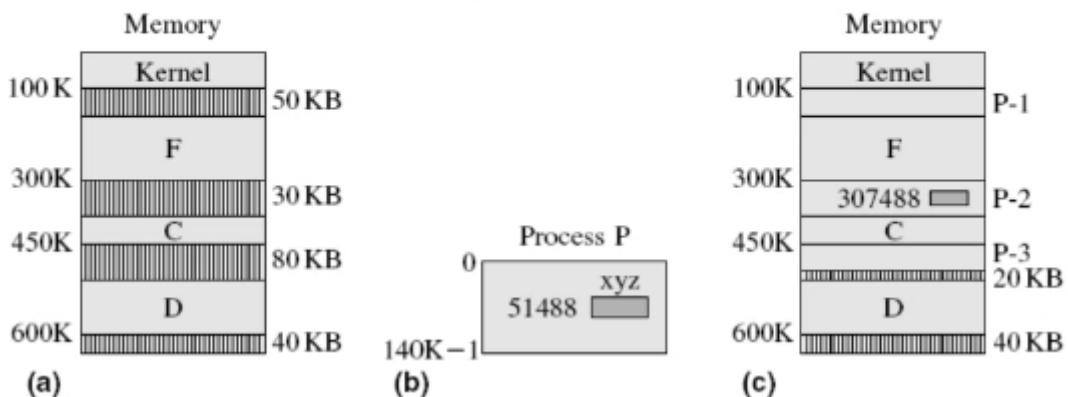
- **Tipo di allocazione di memoria Contigua**
ciascun processo è contenuto in una singola sezione contigua della memoria. Poiché concediamo blocchi di memoria di dimensioni adatte alla richiesta e con tecniche di compattazione con rilocazione dinamica (è necessario avere un registro di compattazione). (Strategie viste prima)



- Tipo di allocazione di memoria Non Contigua

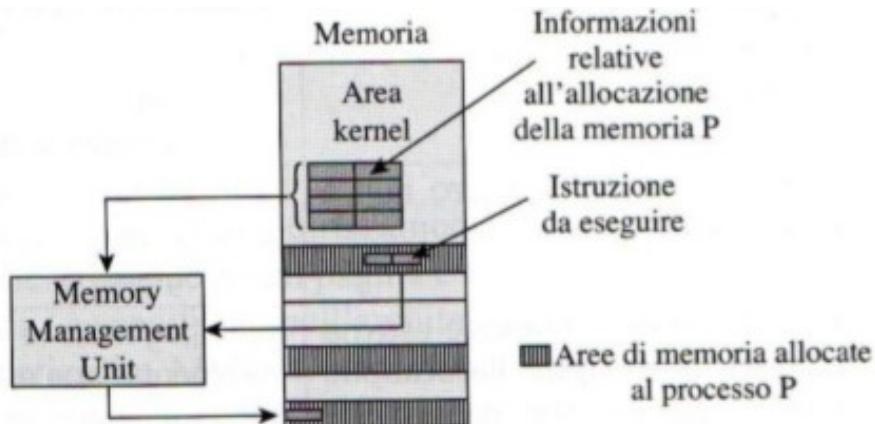
le varie parti di un processo sono contenute in più aree di memoria, anche sparpagliate. Questo modello di allocazione della memoria consente al kernel di riutilizzare le aree di memoria libere più piccole rispetto alla dimensione di un processo, per cui si può ridurre la frammentazione esterna. (Questo tipo di allocazione richiede l'uso di una MMU in hardware).

Process component	Size	Memory start address
P-1	50 KB	100K
P-2	30 KB	300K
P-3	60 KB	450K



In questo caso sussiste l'indirizzo logico della variabile xyz è 51488: il primo blocco P-1 di 50 KB (51200 byte) e inizia all'indirizzo 100K (102400 byte) non la conterrà sicuramente, quindi sarà nei primi $51488 - 51200 = 288$ byte a partire da P-2 che inizia al blocco 300K (307200 byte) e ha una dimensione di 30KB (30720 byte) quindi P-2 va da 307200 byte a $307200 + 30720 = 337920$ byte. L'offset della variabile xyz era di 288 byte allora si troverà nella posizione di memoria $307200 + 288 = 307488$. Eccome si traducono gli indirizzi in questo caso.

Il kernel memorizza le informazioni relative alle aree di memoria allocate al processo P in una tabella e le rende disponibili alla MMU. La CPU invia gli indirizzi logici di ogni dato o istruzione utilizzati nel processo alla MMU. La MMU utilizza le informazioni relative all'allocazione della memoria, memorizzate nella tabella, per calcolare i corrispondenti indirizzi fisici. Questo indirizzo è chiamato indirizzo di memoria effettivo del dato o dell'istruzione. La procedura per calcolare tale indirizzo a partire da un indirizzo logico è chiamata traduzione dell'indirizzo.



La formula che usa la MMU per calcolare l'indirizzo logico e quello fisico è:

$$\text{indirizzo effettivo in memoria (comp}_i, \text{ byte}_i\text{)} = \text{inizio dell'indirizzo dell'area di memoria allocata a comp}_i + \text{offset di byte}_i$$

Ove per comp_i sta per la componente del processo (nel caso di prima P-2) e byte_i è l'offset (come prima 288).

Con la protezione della memoria genererà un interrupt della comp_i .

Esistono fondamentalmente 3 approcci per implementare l'allocazione non contigua della memoria:

- Paginazione

ogni processo viene diviso in parti di dimensione fissata, chiamate pagine di dimensioni standard s byte, dove s è potenza di 2. Si considera che la memoria di un computer sia composta da frame di pagina, dove un frame di pagina è un'area di memoria che ha la stessa dimensione di una pagina. Questi frame sono numerati da 0 a #frame-1 dove #frame è il numero dei frame di pagina di memoria.

Il kernel tiene conto di due tipi di strutture:

- Una coda dei frame liberi, che se un processo fa una richiesta di allocazione della pagina la consulta e se esiste alloca la pagina e associa il frame libero trovato
- *Una tabella delle pagine* (ogni tabella ha una dimensione di 32 o 64 bit) consente per ciascun processo di “mappare” le pagine virtuali sui frame corrispondenti, cioè per indicare in quale frame di memoria si trovano le varie pagine (Questa tabella facilita l'implementazione della traduzione degli indirizzi, del caricamento su richiesta e delle operazioni di sostituzione delle pagine).
 - Ciascuna tabella contiene un elemento (riga) per ogni pagina allocata al processo.
 - I campi (colonne) sono molteplici e contengono svariate informazioni.
 - Tra le informazioni più importanti occorre ricordare:
 - il bit di validità, che indica se la pagina è presente o meno in memoria (0 non presente – 1 presente)
 - il numero frame, che indica il frame di memoria occupato dalla pagina
 - il campo modificato, che indica se la pagina è stata modificata o meno, cioè se è dirty. Nella memoria virtuale quando una pagina viene modificata (cioè è dirty) deve essere copiata sul disco (page-out).

ATTENZIONE considerare i campi della struttura sottostante come “l'ingresso” della pagina, inoltre per ogni pagina sono memorizzati un certo numero di byte per memorizzare questi attributi.

Informazioni varie					
Bit di validità	Frame #	Prot info	Ref info	Modificato	Altre info

Campo	Descrizione
Bit di validità	Indica se la pagina descritta dall'elemento attualmente è presente in memoria. Questo bit è anche chiamato bit di presenza.
Frame #	Indica quale frame di memoria è occupato dalla pagina.
Prot info	Indica le modalità di utilizzo del contenuto della pagina, se in scrittura, lettura o esecuzione.
Ref info	Fornisce informazioni riguardanti i riferimenti fatti alla pagina mentre era in memoria.
Modificato	Indica se la pagina è stata modificata mentre era in memoria, ovvero se è dirty. Questo campo è un singolo bit, detto dirty bit.
Altre info	Altre informazioni utili relative alla pagina, per esempio, la sua posizione nello spazio di swap.

Ogni indirizzo logico usato in un processo è rappresentato da una coppia (p_i, b_i) dove p_i è un numero di pagina e b_i è il numero di byte nella pagina p_i (il suo valore è $0 \leq b_i \leq s$). La MMU consulta la tabella delle pagine che restituisce l'indirizzo del frame corrispondente.

La formula di traduzione dell'indirizzo fisico è:

$$\text{indirizzo effettivo in memoria (comp}_i, \text{byte}_i\text{)} = \text{numero di frame} * \text{dimensione della pagina (in byte)} + \text{scostamento (in byte)}.$$

Usiamo la seguente notazione per descrivere come viene eseguita la traduzione dell'indirizzo logico in fisico:

s dimensione di una pagina;

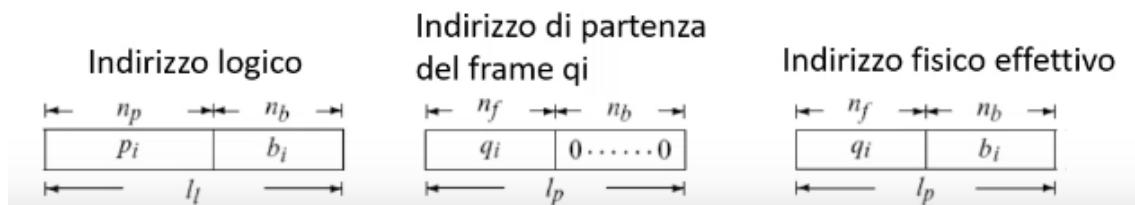
l_l lunghezza di un indirizzo logico (ovvero, numero di bit in esso contenuti);

l_p lunghezza dell'indirizzo fisico;

n_b numero di bit utilizzati per rappresentare il byte in un indirizzo logico; viene scelto in modo tale che $s = 2^{n_b}$.

n_p numero di bit utilizzati per rappresentare il numero di pagina in un indirizzo logico;

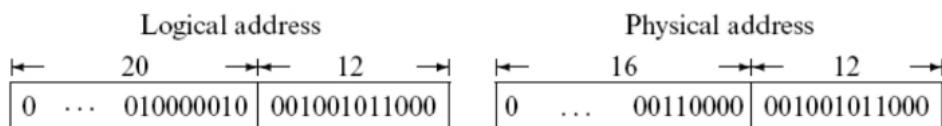
n_f numero di bit utilizzati per rappresentare il numero di frame in un indirizzo fisico.



Quindi è un processo di concatenazione.

ESEMPIO Traduzione:

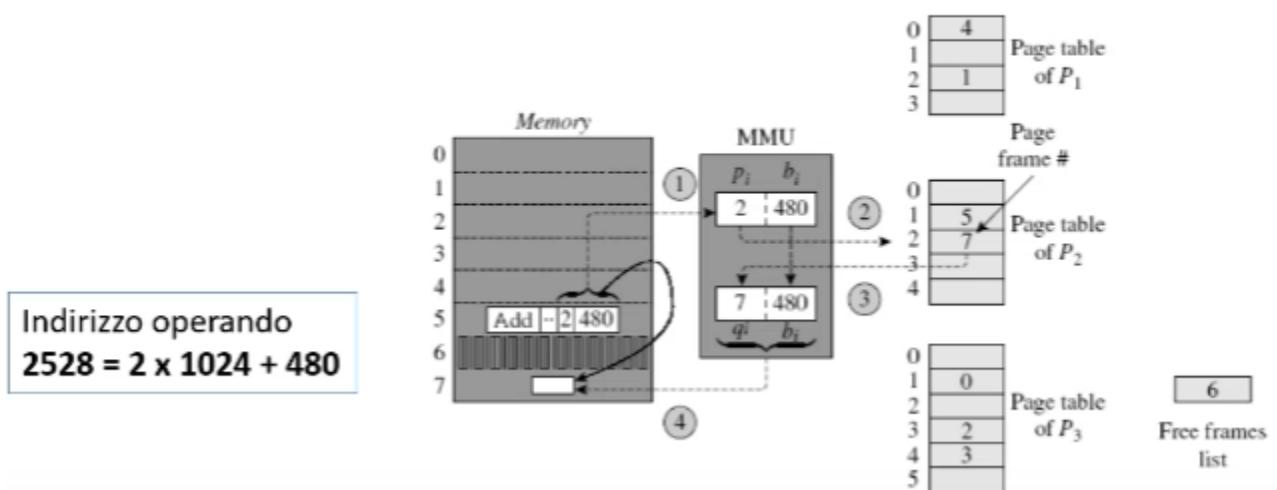
- Indirizzi logici a 32 bit
- Dimensione pagina 4KB
 - 12 bit sono adeguati per indirizzare i byte in una pagina
 - $2^{12} = 4\text{KB}$
- Per una memoria di dimensioni di 256 MB, $l_p = 28$
- Se alla pagina 130 è allocato il frame 48,
 - $p_i = 130$ e $q_i = 48$
 - Se $b_i = 600$, gli indirizzi logici e fisici sono:



In pratica, se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n unità di indirizzamento, allora $m-n$ bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano lo scostamento di pagina.

ESEMPIO ritrovo dato con paging:

Effective memory address of logical address (p_i, b_i)
 = start address of the page frame containing page $p_i + b_i$



Supponga paginazione in cui si assume che la dimensione di pagina sia 1 KB, ossia 1024 byte.

La memoria contiene 8 frame numerati da 0 a 7.

Per ciascuno dei tre processi P_1 , P_2 e P_3 esiste una tabella delle pagine, nella quale sono contenute le informazioni sull'allocazione di memoria. In particolare, ogni elemento di una tabella contiene il numero del frame in cui una pagina risiede. Come si può vedere dalle tabelle, i processi P_1 , P_2 e P_3 hanno alcune delle loro pagine in memoria:

- P_1 ha le sue pagine 0 e 2 nei frame 4 e 1 della memoria ... le altre pagine staranno in qualsiasi altro frame
- P_2 ha le sue pagine 1 e 2 nei frame 5 e 7 della memoria ... le altre pagine staranno in qualsiasi altro frame
- P_3 ha le sue pagine 1, 3 e 4 nei frame 0, 2 e 3 della memoria ... le altre pagine staranno in qualsiasi altro frame

Un'ulteriore tabella, chiamata tabella dei frame liberi contiene la lista dei frame liberi in memoria.

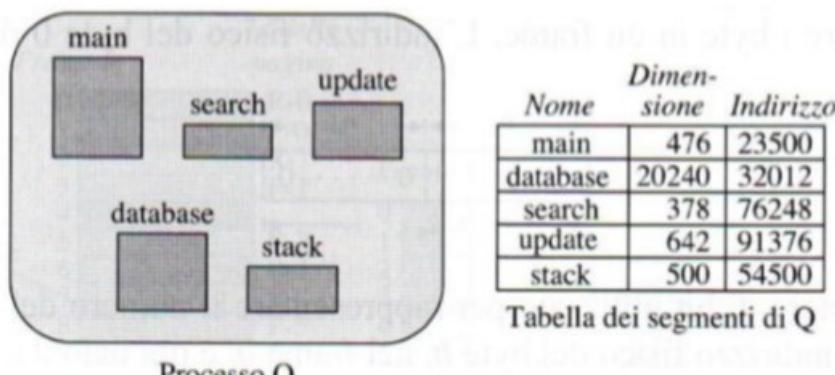
Passi per la traduzione degli indirizzi:

si sta traducendo l'indirizzo logico della pagina 2 del processo P_2 nell'effettivo indirizzo di memoria (questa operazione è compiuta dal MMU, che fa uso della tabella delle pagine di P_2 per effettuare quest'operazione). Adesso la MMU deve eseguire questi passaggi:

1. Vede l'indirizzo dell'operando 2528 come la coppia (2, 480) visto che $2528 = 2($ numero della pagina in cui risiede il dato) $\times 1024$ (page size) + 480 (offset)
2. Accede all'elemento relativo alla pagina. nella tabella delle pagine di P_2
3. Questo elemento contiene il numero di frame 7, quindi la MMU calcola l'indirizzo effettivo $7 \times 1024 + 480$
4. Utilizza questo indirizzo per accedere alla memoria e, in effetti, accede al byte 480 del frame 7.

- Nella segmentazione

un programmatore identifica le componenti logiche (come una funzione o una struttura dati.) di un processo chiamate segmenti di dimensioni variabili. Per minimizzare la frammentazione esterna usa tecniche di riuso.



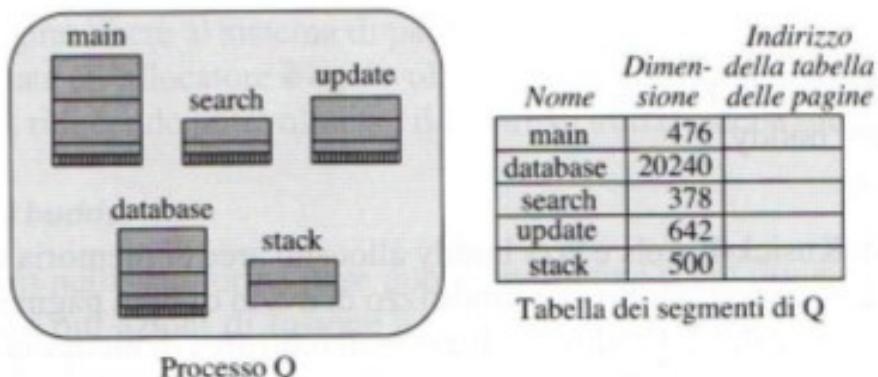
Fondamentalmente il kernel mantiene in memoria una tavola dei segmenti ove memorizza l'indirizzo di partenza e la dimensione. il calcolo dell'indirizzo di memoria effettivo per l'indirizzo logico (s_i, b_i) consiste nel sommare b_i all'indirizzo di avvio del segmento s_i . Elimina la frammentazione interna.

ESEMPIO:

- Se in **update** c'è l'istruzione **get_sample** al byte 232
- $(update, get_sample)$ ha indirizzo di memoria effettivo: $91376 + 232 = 91608$

• Approccio Irido Segmentazione con Paginazione

prende il meglio dei due, dove ciascun segmento viene paginato. Cioè facilita la condivisione del codice, dei dati e dei moduli di programma tra i processi senza incorrere nella frammentazione esterna; tuttavia, si verifica la frammentazione interna così come si verifica nella paginazione.



Secondo tale approccio, ogni segmento di un programma viene paginato separatamente; di conseguenza viene allocato a ogni segmento un numero intero di pagine.

Per ogni segmento viene creata una tabella delle pagine e l'indirizzo di questa tabella viene memorizzato nell'elemento relativo al segmento nella tabella dei segmenti. Come si può vedere nella figura, ogni segmento viene paginato separatamente, per cui si verifica frammentazione interna nell'ultima pagina di ogni segmento. Ogni elemento della tabella contiene l'indirizzo della tabella delle pagine e la dimensione del segmento (utile per la protezione della memoria).

Per ogni segmento è costruita una tabella delle pagine. L'indirizzo della tabella delle pagine è mantenuto nell'entrata del segmento della tabella dei segmenti.

La traduzione di un indirizzo logico (s_i, b_i) avviene in due fasi.

1. Nella prima fase, si cerca l'elemento di s_i nella tabella dei segmenti e viene estratto l'indirizzo della sua tabella delle pagine. Il numero del byte b_i è ora diviso in una coppia (ps_i, bp_i), dove:
 - ps_i è il numero di pagina nel segmento s_i
 - bp_i è il numero del byte nella pagina p_i
2. Nella seconda fase, s_i completa il calcolo dell'indirizzo effettivo così come nella paginazione, ovvero si ottiene il numero del frame di ps_i , e bp_i viene concatenato ad esso per ottenere l'indirizzo effettivo.

La seguente tabella mette a confronto la paginazione con la segmentazione:

Oggetto	Confronto
Concetto	Una pagina è una porzione a misura fissa di uno spazio di indirizzamento di un processo identificato dall'hardware della memoria virtuale. Un segmento è un'entità logica all'interno di un programma, ossia una funzione, una struttura dati o un oggetto. I segmenti sono identificati dal programmatore.
Dimensione delle componenti	Tutte le pagine sono della stessa dimensione. I segmenti possono essere di dimensioni differenti.
Frammentazione esterna	Non si verifica durante la paginazione perché la memoria è divisa in page frame la cui dimensione è uguale alla dimensione delle pagine. Si verifica nella segmentazione perché un'area libera di memoria può essere piccola per ospitare un segmento.
Frammentazione interna	Si verifica nell'ultima pagina di un processo durante la paginazione. Non si verifica nella segmentazione perché un segmento è allocato in un'area di memoria la cui dimensione è uguale alla dimensione del segmento.
Condivisione	La condivisione (sharing) di pagine è realizzabile subordinatamente ai vincoli sulla condivisione delle pagine di codice descritti successivamente nel Paragrafo 12.6. La condivisione di segmenti è possibile liberamente.

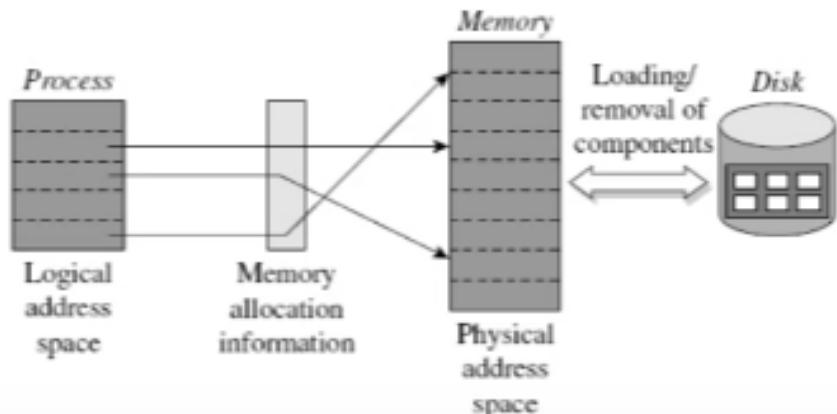
Funzione	Allocazione contigua	Allocazione non contigua
Allocazione della memoria	Il kernel alloca una singola area di memoria a un processo.	Il kernel alloca diverse aree di memoria a un processo – ogni area contiene una componente del processo.
Traduzione dell'indirizzo	La traduzione dell'indirizzo non è necessaria.	La traduzione dell'indirizzo è eseguita dalla MMU durante l'esecuzione del programma.
Frammentazione della memoria	Si verifica frammentazione esterna quando viene usata l'allocazione first-fit, best-fit o next-fit. Si verifica frammentazione interna se l'allocazione della memoria viene eseguita in blocchi di poche dimensioni standard.	Nella paginazione non si verifica la frammentazione esterna ma può verificarsi la frammentazione interna. Nella segmentazione si verifica la frammentazione esterna, ma non si verifica la frammentazione interna.
Swapping	Se il computer non ha un registro di rilocazione, un processo swap-pato deve essere rimesso nell'area originariamente allocata a esso.	Le componenti di un processo swap-pato possono essere caricate in una qualunque parte della memoria.

La MEMORIA VIRTUALE è una parte della gerarchia di memoria composta da una memoria e da un disco. Quando un processo viene mandato in esecuzione, il gestore della memoria virtuale carica solo quella porzione che contiene l'indirizzo di start del processo, cioè l'indirizzo dell'istruzione con cui la sua esecuzione comincia. Successivamente, carica altre porzioni del processo solo quando necessarie (questa tecnica prende il nome di caricamento su richiesta), si inizia con la componente di avvio di un processo. Questa soluzione fa sì che la richiesta totale di memoria di un processo possa superare la dimensione della memoria del sistema. Permette, inoltre, che un maggior numero di processi risiedano in memoria contemporaneamente, perché ognuno di loro occupa meno memoria della propria dimensione. Quando una componente di un processo in memoria non serve esegue una rimozione della componente.

Le prestazioni di un processo dipendono dalla percentuale delle sue porzioni che devono essere caricate in memoria dal disco. Il kernel implementa tale illusione tramite una combinazione di mezzi hardware e software, cioè attraverso la MMU (HW) e il gestore della memoria virtuale (SW).

Essa si basa sul modello di allocazione di memoria non contigua, per tale motivo:

- le parti di un processo (componenti) possono essere caricate in aree di memoria non adiacenti
- l'indirizzo di ciascun operando o istruzione di un processo è un indirizzo logico del tipo (p_i, b_i) ; è la MMU che traduce questo indirizzo logico nell'indirizzo di memoria effettivo in cui si trova l'operando o l'istruzione.



Nella memoria virtuale viene utilizzata la paginazione su richiesta con tabella delle pagine: La paginazione su richiesta, o caricamento su richiesta di pagina, è costituita da quelle azioni che effettua il gestore della memoria virtuale per tenere aggiornate le tabelle delle pagine dei vari processi. Viene invocata solo quando serve una parte.

Sul disco è gestita una copia dell'intero spazio di indirizzamento di un processo: L'area del disco è chiamata spazio di swap del processo. All'avvio del processo il gestore della memoria virtuale alloca lo spazio del swap del processo e vi alloca codice e dati.

PROPRITA DI OTTIMIZZAZIONE DEL PAGING ON-DEMAND [paging come si usa oggigiorno (con memoria virtuale)]:

- La dimensione della pagina determina il numero di bit richiesti per rappresentare il numero di byte in una pagina, spreco di memoria causata da frammentazione interna, dimensione della tabella delle pagine per un processo e i tassi di page fault quando una quantità fissa di memoria è allocata ad un processo.

La strategia usata per la giusta grandezza della pagina: supponiamo che l'entrata in ogni tabella delle pagine richieda 1 byte (bit validità etc...) la dimensione della pagina s è data \sqrt{z} ove z è la dimensione del processo.

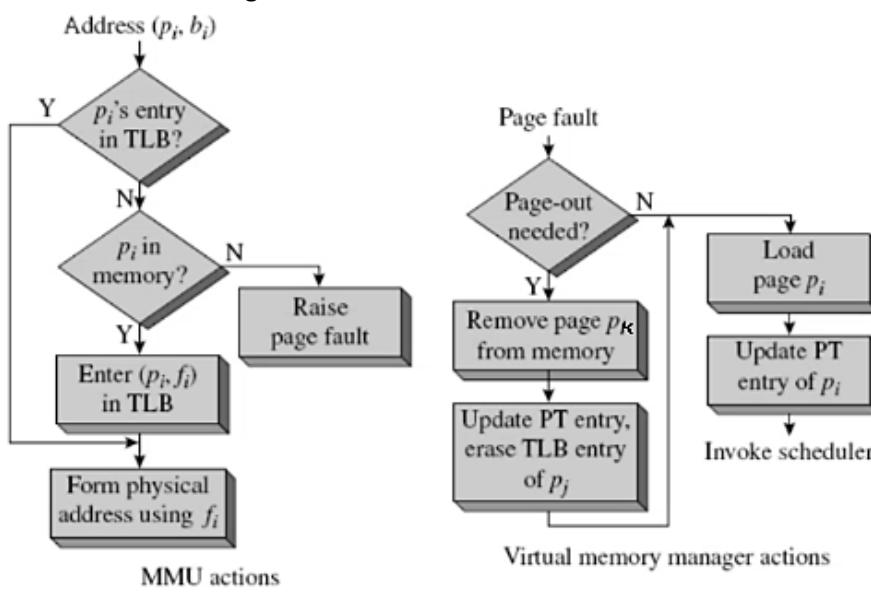
In realtà viene utilizzato una dimensione più grande: l'entrata in ogni tabella delle pagine richieda $byte < 1$ byte, se ho delle pagine più piccole i costi dell'hardware aumentano, poiché devo avere più bit identificativi, e quando i dischi lavorano con blocchi più piccoli sono inefficienti.

- Per la traduzione efficiente degli indirizzi si usa un supporto di memoria veloce ed associativa alla MMU, ossia la TLB (Transation Look-aside Buffer). Una volta fornito l'indirizzo logico la MMU deve far riferimento alla tabella delle pagine in memoria, chiaramente devo accedere in memoria, per cercare di ridurre gli accessi in memoria si introduce il TLB che risale alle informazioni dove sono memorizzate in memoria fisica le pagine. Ogni entrata del TLB ha come informazioni possiede: numero pagina, numero di frame, informazione sulla protezione. Mantiene solo per poche accedute recentemente. Funziona in questo modo:
 - fornire l'indirizzi logici al TLB
 - verificare che l'indirizzo della pagina si trova nel TLB
 - se sì, prendo il frame e lo restituisco
 - se non ci sta vado a cercarlo in memoria normalmente

- Ottimizzare e riferire in modo veloce e sicuro la giusta tabella delle pagine per n processi? Si utilizza il supporto hardware di supporto registro PTAR (Page Table Address Register) sta ad indicare l'indirizzo della tabella delle pagine per quel processo che viene schedulato. Il kernel per semplificare il compito quando schedula un processo prende l'indirizzo della tabella delle pagine iniziale dal PCB del processo schedulato e questo indirizzo lo mette all'interno del PTAR. La MMU si va a calcolare l'indirizzo fisico della pagina in questo modo:

$$\text{indirizzo fisico} = \langle \text{PTAR}(pid, PTSR(pid)) \rangle + p_i * \text{Length}(page).$$

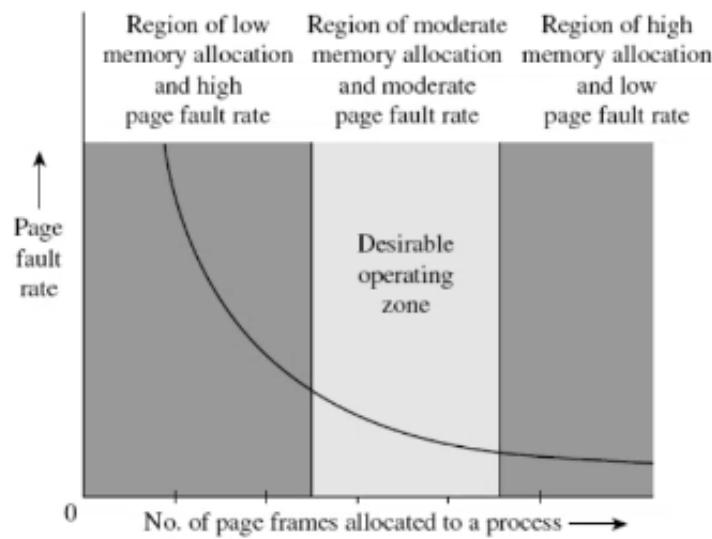
- prende il contenuto PTAR del processo
- vi aggiunge il numero delle pagine riferito all'interno dell'operando dell'istruzione (p_i)
- e lo moltiplica per la dimensione della pagina
- si controlla che l'indirizzo logico del processo rientra nello spazio indirizzamento logico del processo, se sfiora genera un'eccezione. (Di questo si occupa il registro di supporto PTSR (Page Table Support Register) nel PCB del processo). Inoltre, entra in gioco il Prot Info che è campo delle pagine che indica i diritti di lettura e scrittura. Se viola i diritti di accesso genera un'altra eccezione.



Tra le operazioni di paging su una memoria ne esistono principalmente 4:

1. PAGE FAULT E CARICAMENTO SU RICHIESTA DELLE PAGINE

Supponiamo che la CPU voglia usare un certo dato o una certa istruzione. La MMU, nell'effettuare la traduzione degli indirizzi verifica, facendo uso della tabella delle pagine, se quella pagina è presente o meno in memoria (fa riferimento al campo bit di validità). Se la pagina non è presente, si genera un page fault e il gestore della memoria virtuale carica la pagina in memoria e aggiorna la tabella delle pagine. Infatti, alcune pagine virtuali non possono avere corrispondenza in memoria fisica, il bit di validità è usato proprio per indicare se la pagina è presente o meno in memoria. Si ha page fault quando un processo tenta di usare una pagina non mappata, cioè non presente in memoria. Il controllo del bit di validità è effettuato dalla MMU controlla il bit di validità della tabella delle pagine; se è 0 vuol dire che la pagina non è presente in memoria, dunque la MMU genera un interrupt chiamato page fault, che è un interrupt di programma. A causa dell'interrupt viene invocato il gestore della memoria virtuale che carica la pagina in memoria e aggiorna la tabella delle pagine di quel processo. In pratica, MMU e gestore della memoria virtuale interagiscono per decidere quando una pagina di un processo deve essere caricata in memoria. Per minimizzare il page fault basta allocare più memoria al processo quindi aumento il numero delle pagine e diminuisco i fault.



Desirable variation of page fault rate with memory allocation.

Se alloco troppo poca memoria (sx) ho un calo delle prestazioni aumentando i page fault

Se alloco moderatamente (ce) ho il giusto rapporto tra page faulting e grado multiprogrammazione

Se alloco troppa la memoria (dx) ho un calo di prestazioni della CPU in stato idle diminuito il grado di multiprogrammazione

2. OPERAZIONI DI PAGE-IN

Quando un processo vuole usare una pagina non presente in memoria e ci sono frame liberi in memoria, il gestore della memoria virtuale carica la pagina dallo spazio di swap (sul disco) in memoria. Questa operazione prende il nome di page-in.

3. OPERAZIONI DI PAGE-OUT

Quando il gestore della memoria virtuale decide di rimuovere una pagina dalla memoria, copia la pagina nello spazio di swap (sul disco). Questa operazione prende il nome di page-out.

4. SOSTITUZIONE DELLE PAGINE

Quando un processo vuole usare una pagina non presente in memoria e non ci sono frame liberi in memoria è necessario ricorrere alla sostituzione delle pagine. La sostituzione di pagina diventa necessaria quando si verifica un page fault e non ci sono frame liberi in memoria. Questa operazione viene svolta dal gestore della memoria virtuale che seleziona una delle pagine attualmente in memoria utilizzando un algoritmo di sostituzione delle pagine. Scegliendo una pagina che non sia vicina alla località corrente (proprietà secondo cui un processo tende ad avere pagine di esecuzione al più contigue possibili, il 15% dei casi si effettua uno shift della località se si esegue un "salto") prendendo una che non possa servire nella prossima iterazione.

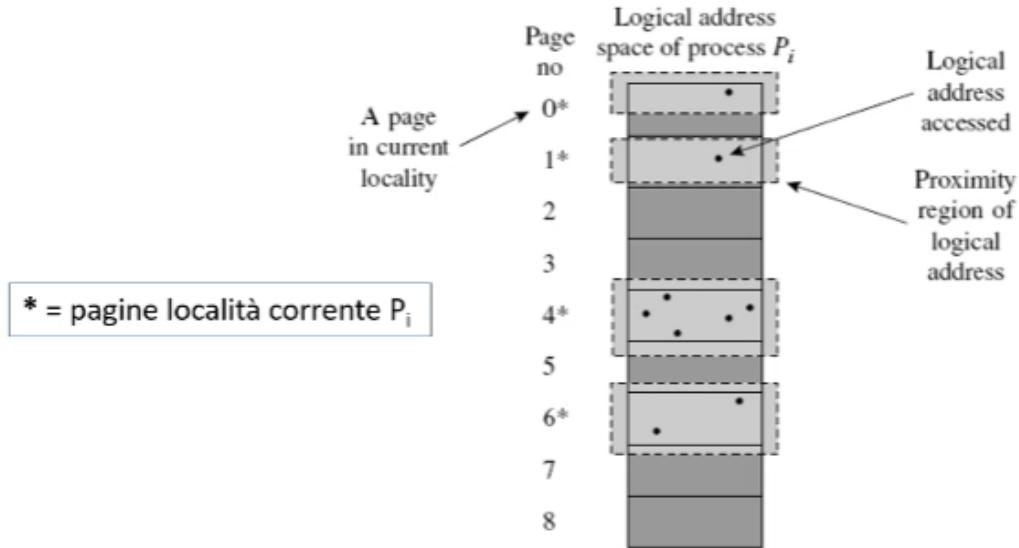


Figure 12.5 Proximity regions of previous references and current locality of a process.

Quindi l'algoritmo non toccherà le pagine con * poiché saranno utilizzata da lì a poco.

Ci sono differenti politiche di sostituzione delle pagine, per evincere la scelta migliore bisogna analizzarle a fondo. Per effettuare questa analisi ci basiamo sul concetto di stringa dei riferimenti alle pagine. Una stringa di riferimenti alle pagine di un processo è una sequenza di pagine a cui un processo ha fatto accesso durante la sua esecuzione. Può essere costruita monitorando l'esecuzione di un processo e formando una sequenza di numeri di pagina. Per convenienza, associamo una stringa dei tempi dei riferimenti t_1, t_2, t_3, \dots a ogni stringa dei riferimenti alle pagine. In questo modo, al k -esimo riferimento di pagina nella stringa dei riferimenti alle pagine è associato un istante di tempo t_k .

Esempio stringa riferimento pagine (lezione 28 maggio):

Un computer supporta istruzioni di 4 byte di lunghezza

- Usa una dimensione di pagina di 1KB
- I simboli A e B del programma in esecuzione sono nelle pagine 2 e 5, rispettivamente

```

START 2040
READ B
LOOP MOVER AREG, A
      SUB  AREG, B
      BC   LT, LOOP
      ...
STOP
A    DS 2500
B    DS 1
END

```

Stringa riferimento pagina	1, 5, 1, 2, 2, 5, 2, 1
Stringa riferimento temporale	$t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, \dots$

- Politica di sostituzione delle pagine ottimale: Strategia Belady

La **sostituzione ottimale** consiste nel sostituire le pagine in modo tale che il numero totale di page fault durante l'esecuzione di un processo sia il minimo possibile. In pratica, un algoritmo del genere sostituisce solo quelle pagine che non si useranno per il periodo di tempo più lungo.

Naturalmente, una politica del genere è irrealizzabile in quanto al momento del page fault, il SO non ha nessun modo di sapere quando si farà riferimento a ciascuna delle pagine, cioè non può conoscere il comportamento futuro di un processo.

Esempio:

Stringa di riferimento 2,3,2,1,5,2,4,5,3,2,5,2



- Politica di sostituzione delle pagine First-In First-Out (FIFO) : traccia ordine di arrivo

La **politica di sostituzione delle pagine FIFO**, ad ogni page fault, sostituisce la pagina che è stata caricata in memoria prima di ogni altra pagina del processo, cioè quella che risiede in memoria da più tempo. In pratica, il SO mantiene una lista di tutte le pagine correntemente in memoria, dove la pagina di testa è la più vecchia e la pagina in coda è quella arrivata più di recente. Al momento del page fault, la pagina in testa viene rimossa anche se è la pagina più utilizzata, per questo motivo viene raramente utilizzato nella sua forma pura.

Per tenere traccia dell'ordine di arrivo si utilizza il campo *Ref_info* della tabella delle pagine.

Esempio:

Stringa di riferimento 2,3,2,1,5,2,4,5,3,2,5,2

2	5	3		
3		2	5	9 page fault
1			4	2

- Politica di sostituzione delle pagine Least Recently Used (LRU): legge di località dei riferimenti

La **politica di sostituzione delle pagine LRU**, a ogni page fault, sostituisce la pagina utilizzata meno di recente con la pagina richiesta. In pratica, viene scaricata la pagina usata meno di recente.

Questa politica è realizzabile ma non è conveniente, poiché per implementarla completamente, è necessario mantenere liste concatenate di tutte le pagine in memoria, con la pagina usata più di recente in testa alla lista; la difficoltà sta nel fatto che la lista va aggiornata ad ogni riferimento alla memoria.

Esempio 12.6 Funzionamento delle politiche di sostituzione di pagine

Si considerino la seguente stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti per un processo *P*:

stringa dei riferimenti alle pagine 0, 1, 0, 2, 0, 1, 2, ... (12.4)

stringa dei tempi dei riferimenti $t_1, t_2, t_3, t_4, t_5, t_6, t_7, \dots$ (12.5)

La Figura 12.15 illustra il funzionamento delle politiche di sostituzione delle pagine ottimale, FIFO e LRU per questa stringa dei riferimenti alle pagine con *alloc* = 2. Per convenienza, mostriamo solo due campi della tabella delle pagine, il *bit di validità* e *ref info*. Nell'intervallo tra t_0 e t_3 (incluso), vengono riferite solo due pagine distinte: le pagine 0 e 1. Possono stare entrambe in memoria nello stesso istante perché *alloc* = 2. t_4 è il primo istante di tempo in cui un page fault determina una sostituzione di pagina.

La colonna di centro della Figura 12.15 mostra i risultati per la politica di sostituzione FIFO. Quando si verifica un page fault all'istante di tempo t_4 , il campo *ref info* mostra che la pagina 0 era stata caricata prima della pagina 1, e così la pagina 0 è sostituita dalla pagina 2.

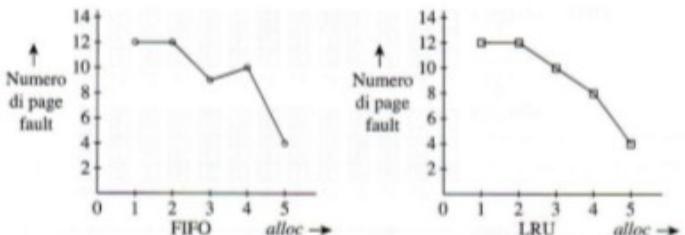
L'ultima colonna della Figura 12.15 mostra i risultati per la politica di sostituzione LRU. Il campo *Ref info* della tabella delle pagine indica quando una pagina era stata riferita l'ultima volta. All'istante di tempo t_4 , la pagina 1 è sostituita dalla pagina 2 perché l'ultimo riferimento della pagina 1 è precedente rispetto all'ultimo riferimento della pagina 0.

Il numero totale di page fault che si verificano con le politiche ottimale, FIFO e LRU sono 4, 6 e 5, rispettivamente. Per definizione, nessun'altra politica ha meno page fault rispetto alla politica di sostituzione ottimale delle pagine.

Istante	Rif. pagina	Ottimale		FIFO		LRU	
		Bit di validità <i>info</i>	Sostitu- zione	Bit di validità <i>info</i>	Sostitu- zione	Bit di validità <i>info</i>	Sostitu- zione
t_1	0	0 1 1	-	0 1 t_1	-	0 1 t_1	-
	1	1 0 0		1 0 0		1 0 0	
	2	0 0 0		2 0 0		2 0 0	
t_2	1	0 1 1	-	0 1 t_1	-	0 1 t_1	-
	1	1 1 1		1 1 t_2		1 1 t_2	
	2	0 0 0		2 0 0		2 0 0	
t_3	0	0 1 1	-	0 1 t_1	-	0 1 t_1	-
	1	1 1 1		1 1 t_2		1 1 t_2	
	2	0 0 0		2 0 0		2 0 0	

t_4	2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td></td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>2</td><td>1</td><td></td></tr></table>	0	1		1	0		2	1		Sostituisci 1 con 2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td>t_2</td></tr><tr><td>2</td><td>1</td><td>t_4</td></tr></table>	0	0		1	1	t_2	2	1	t_4	Sostituisci 0 con 2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>t_3</td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>2</td><td>1</td><td>t_4</td></tr></table>	0	1	t_3	1	0		2	1	t_4	Sostituisci 1 con 2
0	1																																	
1	0																																	
2	1																																	
0	0																																	
1	1	t_2																																
2	1	t_4																																
0	1	t_3																																
1	0																																	
2	1	t_4																																
t_5	0	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td></td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>2</td><td>1</td><td></td></tr></table>	0	1		1	0		2	1		-	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>t_5</td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>2</td><td>1</td><td>t_4</td></tr></table>	0	1	t_5	1	0		2	1	t_4	Sostituisci 1 con 0	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>t_5</td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>2</td><td>1</td><td>t_4</td></tr></table>	0	1	t_5	1	0		2	1	t_4	-
0	1																																	
1	0																																	
2	1																																	
0	1	t_5																																
1	0																																	
2	1	t_4																																
0	1	t_5																																
1	0																																	
2	1	t_4																																
t_6	1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td></td></tr><tr><td>2</td><td>1</td><td></td></tr></table>	0	0		1	1		2	1		Sostituisci 0 con 1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>t_5</td></tr><tr><td>1</td><td>1</td><td>t_6</td></tr><tr><td>2</td><td>0</td><td></td></tr></table>	0	1	t_5	1	1	t_6	2	0		Sostituisci 2 con 1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>t_5</td></tr><tr><td>1</td><td>1</td><td>t_6</td></tr><tr><td>2</td><td>0</td><td></td></tr></table>	0	1	t_5	1	1	t_6	2	0		Sostituisci 2 con 1
0	0																																	
1	1																																	
2	1																																	
0	1	t_5																																
1	1	t_6																																
2	0																																	
0	1	t_5																																
1	1	t_6																																
2	0																																	
t_7	2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td></td></tr><tr><td>2</td><td>1</td><td></td></tr></table>	0	0		1	1		2	1		-	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td>t_6</td></tr><tr><td>2</td><td>1</td><td>t_7</td></tr></table>	0	0		1	1	t_6	2	1	t_7	Sostituisci 0 con 2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td>t_6</td></tr><tr><td>2</td><td>1</td><td>t_7</td></tr></table>	0	0		1	1	t_6	2	1	t_7	Sostituisci 0 con 2
0	0																																	
1	1																																	
2	1																																	
0	0																																	
1	1	t_6																																
2	1	t_7																																
0	0																																	
1	1	t_6																																
2	1	t_7																																

La figura in basso illustra come variano i page fault per gli algoritmi di sostituzione di pagina FIFO e LRU per la stringa dei riferimenti dell'esempio in alto.



L'asse verticale mostra il numero totale di page fault, mentre sull'asse orizzontale è riportato l'alloc, cioè il numero di frame allocati al processo. Come si può notare, c'è un'anomalia nel grafico della politica FIFO: il numero di page fault aumenta quando aumenta il numero di frame. Questo comportamento anomalo è noto come anomalia di Belady. Per questo motivo, il gestore della memoria non può utilizzare la politica di sostituzione FIFO perché incrementando i frame allocati al processo può aumentare la frequenza dei page fault. Al contrario, nella politica LRU, il numero di page fault diminuisce all'aumentare dei frame.

Quindi il migliore è scegliere l'LRU che ha minor page fault e verifica le proprietà dello stack, ossia aumentando il numero di frame in memoria per il processo da 3 a 4 diminuiscono i page fault, ciò che non accade con la FIFO.

Ma nella maggior parte dei sistemi moderni non è possibile, poiché il Ref_info (storico delle pagine) servirebbero molti bit.

- Nei moderni sistemi operativi ha al più un bit per il campo Ref_info per tenere traccia i riferimenti delle varie pagine. L'NRU (Not Recently Used) è un'approssimazione dell'LRU.

Una semplice politica NRU è la seguente: il bit di riferimento di una pagina è inizializzato a 0 quando la pagina è caricata, ed è impostato a 1 quando la pagina viene referenziata. Quando si rende necessaria una sostituzione di pagina, se il gestore della memoria virtuale verifica che i bit di riferimento di tutte le pagine sono diventati 1, resetta i bit di tutte le pagine a 0 e arbitrariamente seleziona una delle pagine per la sostituzione; altrimenti, sostituisce una pagina il cui bit di riferimento è 0. Successive sostituzioni di pagina dipenderanno da quali pagine sono state riferite dopo il reset dei bit di riferimento.

L'attrattiva principale dell'algoritmo NRU è che è facile da capire, abbastanza semplice da implementare, e che dà delle prestazioni che spesso risultano soddisfacenti.

Si può fare meglio nell'avere una migliore discriminazione delle pagine, invece che sceglierne arbitrariamente ed evitare di settare tutti i bit dei frame a 0 una volta messi tutti ad 1.

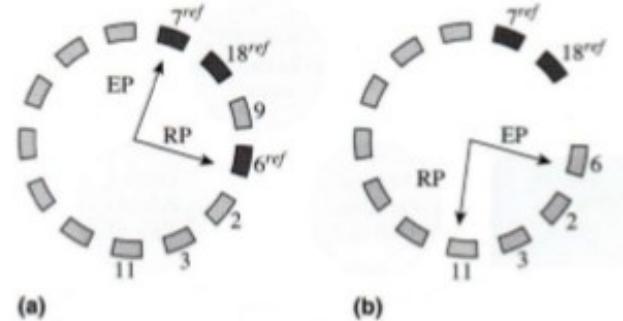
La strategia proposta è quello degli algoritmi di clock.

Gli algoritmi di clock sono una sottoclassificazione ampiamente usata degli algoritmi NRU. Essi forniscono la migliore discriminazione tra le pagine resettando i bit di riferimento delle pagine periodicamente, e non solo quando tutte diventano 1; in questo modo risulta possibile sapere se una pagina è stata riferita nell'immediato passato, piuttosto che dal momento in cui tutti i bit di riferimento sono stati resettati a 0.

Per implementare questi algoritmi, le pagine di tutti i processi in memoria sono memorizzate in una lista circolare e i puntatori usati dagli algoritmi si muovono sulle pagine ripetutamente in modo analogo al movimento delle lancette di un orologio. Il bit di riferimento di una pagina è impostato a 1 quando:

- la pagina è caricata per la prima volta in un frame in memoria

- quando la pagina caricata nel frame viene utilizzata
Quando si verifica un page fault, si effettua una scansione della lista e si sostituisce il frame che ha il bit di riferimento posto a 0.
- Nell'algoritmo di clock a una lancetta, una esecuzione è composta da due passi su ogni pagina. Nel primo passo, il gestore della memoria virtuale semplicemente resetta il bit di riferimento della pagina puntata dal puntatore. Nel secondo passo cerca tutte le pagine i cui bit di riferimento sono ancora off e li aggiunge alla lista dei frame liberi.
- Nell'algoritmo di clock a due lancette, sono utilizzati due puntatori. Un puntatore, chiamato resetting pointer (RP), serve per resettare i bit di riferimento, mentre l'altro puntatore, chiamato examining pointer (EP), viene utilizzato per controllare i bit di riferimento. Entrambi i puntatori vengono incrementati simultaneamente. Il frame al quale il puntatore EP punta è aggiunto alla lista dei frame liberi se il suo bit di riferimento è off.



Esempio 12.8 Algoritmo di clock a due lancette

La Figura 12.20 illustra il funzionamento dell'algoritmo di clock a due lancette utilizzato dal gestore dei frame liberi di Figura 12.19. Il simbolo ref su una pagina implica che il bit di riferimento della pagina è settato ad 1; l'assenza di questo simbolo implica che il bit di riferimento è 0. Quando viene attivato il gestore dei frame liberi, esamina la pagina 7, che è puntata dall'examining pointer [Figura 12.20(a)]. Il suo bit di riferimento è 1, quindi vengono avanzati entrambi i puntatori, il resetting e l'examining. In questo istante, il bit di riferimento della pagina 6 è resettato a 0 perché RP stava puntando a esso. Il puntatore examining si sposta su pagina 18 (e il resetting si sposta su pagina 2) perché, anch'esso ha il suo bit di riferimento settato ad 1. Ora resta sulla pagina 9. La pagina 9 ha il suo bit di riferimento a 0, quindi è rimossa dalla lista delle pagine in memoria e aggiunta a quella dei frame liberi. I puntatori resetting e examining puntano ora alle pagine 6 e 11, rispettivamente [Figura 12.20(b)].

La distanza tra i puntatori RP ed EP fornisce proprietà diverse agli algoritmi. Se i puntatori sono molto vicini, sarà esaminata una pagina subito dopo il reset del suo bit di riferimento, di conseguenza solo le pagine usate di recente rimarranno in memoria. Se i puntatori del clock sono abbastanza lontani, solo le pagine che non sono state utilizzate da molto tempo saranno rimosse.

Organizzazione pratica delle tabelle delle pagine da parte della memoria virtuale: Abbiamo detto che le tabelle delle pagine sono memorizzate nella memoria. Negli attuali computer, vengono mandati in esecuzione molti processi contemporaneamente, per questo motivo, gran parte della memoria RAM potrebbe essere usata per la memorizzazione delle tabelle delle pagine dei processi. Per risolvere questo spreco, derivante dalla presenza costante in memoria di tabelle delle pagine molto grosse, vengono seguiti fondamentalmente due approcci (in entrambi gli approcci, il TLB viene usato per ridurre il numero di riferimenti di memoria necessari per eseguire la traduzione degli indirizzi) per ridurre la dimensione in memoria assegnata alle tabelle delle pagine:

- Tabella delle pagine invertita

La tabella delle pagine invertita (IPT) contiene un elemento per ogni frame (pagina fisica) di memoria. Ciascun elemento contiene il numero di pagina che occupa il frame e l'ID del processo. La tabella viene così denominata perché l'informazione in essa contenuta è invertita rispetto alla classica tabella delle pagine, che contiene un elemento per ogni pagina.

La dimensione di una IPT non dipende dal numero e dalle dimensioni dei processi, ma dipende dalla dimensione della memoria. Quindi questo approccio riduce la quantità di memoria occupata.

In questo approccio, però, è più difficile effettuare la traduzione degli indirizzi da indirizzo logico a indirizzo fisico. Supponiamo di avere un processo R. Abbiamo detto che ogni elemento di una IPT contiene due informazioni: l'ID del processo e il numero di pagina che occupa quel frame. Quindi una coppia (R, p_i) nella f-esima entry (riga) della tabella invertita indica che il frame f_i è occupato dalla pagina p_i di un processo R. Se si vuole effettuare la traduzione di questo indirizzo, non si dovrà cercare più un indice ma una coppia di elementi (R, p_i) nella tabella IPT. Per velocizzare questa ricerca, vengono usate la TLB, e una hash table nel caso in cui si verifichi un page miss, cioè se la pagina cercata non è nel TLB.

- Ogni entrata della IPT è una coppia ordinata (id processo, numero pagina)
- La coppia (R, p_i) nell'entrata f_i -esima indica che il frame f_i è occupato dalla pagina p_i del processo R
- Lo scheduler, selezionato un processo, ne preleva l'id dal PCB e lo invia in un registro della MMU

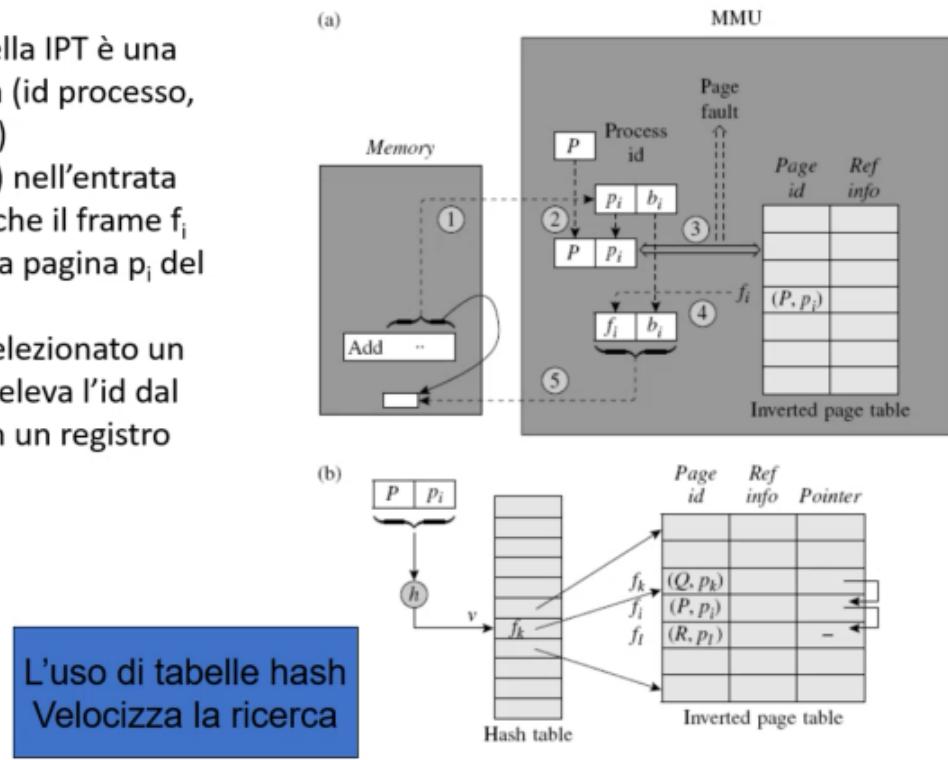
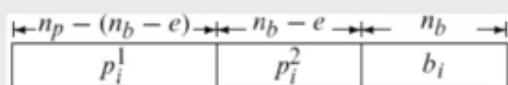


Figure 12.10 Inverted page table: (a) concept; (b) implementation using a hash table.

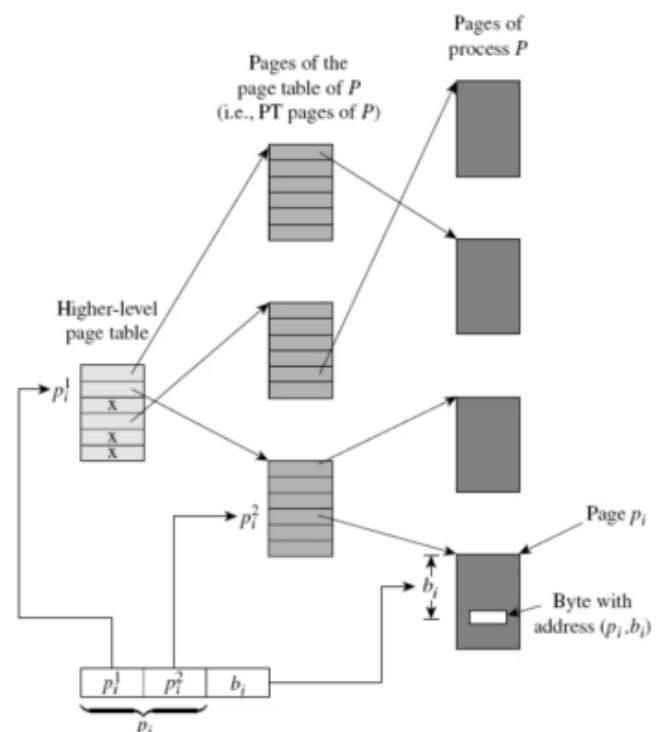
- Tabella delle pagine multilivello

La tabella delle pagine multilivello consiste nel suddividere una tabella delle pagine, di grosse dimensioni, in più livelli, evitando di tenere tutti i livelli in memoria per tutto il tempo. In pratica, in questo modo, si effettua una paginazione della tabella dei processi: una tabella delle pagine di alto livello viene utilizzata per accedere alle varie pagine della tabella delle pagine. Se la tabella delle pagine di alto livello fosse grande, potrebbe essere essa stessa paginata e così via.

- Se la dimensione di un'entrata della tabella è 2^e byte, il numero di entrate della tabella in una pagina PT è $2^{n_b}/2^e$
- L'indirizzo logico (p_i, b_i) è raggruppato in tre campi:



- La pagina in PT con p_i^1 contiene l'entrata per p_i
- p_i^2 è il numero di entrata per p_i nella pagina della PT
- b_i



Quindi sostanzialmente il gestore della memoria di occupa di:

Funzione	Descrizione
Gestione spazio di indirizzamento logico	Set up dello spazio di swap di un processo. Organizzare il suo spazio di indirizzamento logico in memoria attraverso le operazioni di page-in e page-out, e gestire la sua tabella delle pagine.
Gestione della memoria	Tenere traccia dei frame occupati e liberi in memoria.
Implementare la protezione di memoria	Conservare le informazioni utili per la protezione di memoria.
Raccogliere informazioni sui riferimenti alle pagine	Il supporto hardware alla paginazione fornisce le informazioni circa i riferimenti alle pagine. Queste informazioni sono tenute in appropriate strutture dati per l'uso da parte dell'algoritmo di sostituzione di pagine.
Eseguire sostituzione di pagine	Eseguire la sostituzione di una pagina quando viene generato un fault e tutti i page frame in memoria, o tutti i frame allocati per un processo, sono occupati.
Allocare memoria fisica	Decidere quanta memoria allocare a un processo e rivedere questa decisione di volta in volta per adattarsi alle necessità del processo e del SO.
Implementare la condivisione delle pagine	Organizzare la condivisione delle pagine da elaborare.

Quanti frame bisogna allocare ad un processo, diciamo molto superficialmente che sia il sovradimensionamento che il sottodimensionamento di memoria concesso a un processo porta a cali di prestazioni del sistema e scarsa efficienza della CPU. Tuttavia, non è chiaro il modo in cui il gestore della memoria virtuale decide il giusto numero di frame da allocare a ogni processo, ossia il giusto valore di alloc per ogni processo.

Vengono utilizzati due approcci per controllare l'allocazione dei frame per un processo:

- allocazione di memoria fissa

l'allocazione di memoria per un processo è fissa; di conseguenza, la prestazione di un processo è indipendente dagli altri processi del sistema. Quando si verifica un page fault in un processo, viene sostituita una delle sue pagine. Questo approccio è detto sostituzione di pagina locale. Nell'allocazione fissa in ambito globale, le decisioni riguardanti l'allocazione di memoria sono eseguite staticamente. La memoria da allocare a un processo è determinata in base ad alcuni criteri quando il processo viene inizializzato. La sostituzione di pagina è sempre eseguita localmente. Questo metodo risente di tutti i problemi connessi a una decisione statica: un sottodimensionamento o un sovradimensionamento di memoria per un processo può influenzare la prestazione del processo stesso e quella del sistema.

- allocazione di memoria variabile

l'allocazione di memoria può essere variata in due modi.

- o Quando si verifica un page fault, tutte le pagine di tutti i processi che sono in memoria possono essere prese in considerazione per la sostituzione. Ciò è identificato come sostituzione globale delle pagine.
- o Alternativamente, il gestore della memoria virtuale può rivedere l'allocazione della memoria per un processo periodicamente sulla base della sua località e sul comportamento riguardo ai page fault, ma quando si verifica un page fault esegue una sostituzione locale di pagina. Nell'allocazione variabile in ambito globale, l'allocazione per il processo attualmente in esecuzione può diventare troppo grande.

La miglior soluzione tra le tre è senza dubbio l'allocazione variabile in ambito locale che utilizza la sostituzione locale delle pagine, perché il gestore della memoria virtuale determina il giusto valore di alloc per un processo, di volta in volta.

Il modello/concetto di un working set fornisce una base per decidere quanti e quali pagine di un processo dovrebbero essere in memoria per ottenere una buona prestazione di processo. L'insieme di tutte le pagine di ogni

processo viene detto working set (insieme di lavoro). È stato creato il modello working set per ridurre sensibilmente il tasso di page fault e consiste nell'assicurarsi che l'insieme di lavoro sia caricato totalmente in memoria prima di consentire ad un altro processo di andare in esecuzione (altrimenti si verificherebbero mille page fault per ogni processo).

Il numero di pagine nel working set può aumentare o diminuire, a seconda della disponibilità delle pagine stesse. Il working set aumenta come un processo page fault. Al contrario esso diminuisce con la diminuzione delle pagine disponibili. Per evitare la consumazione completa della memoria, le pagine devono essere rimosse dai working set del processo e trasformate in pagine disponibili per un loro eventuale utilizzo.

Il sistema operativo diminuisce i working set del processo nei seguenti modi:

- scrivendo su pagine modificate in un'area dedicata, su di un dispositivo memoria di massa (generalmente conosciuti come spazio di swapping o paging)
- contrassegnando pagine non modificate come libere (non vi è alcuna necessità di scrivere queste pagine su disco in quanto queste non sono state modificate)

Per determinare i working set appropriati per tutti i processi, il sistema operativo deve possedere tutte le informazioni sull'utilizzo per tutte le pagine. In questo modo, il sistema operativo, determina le pagine usate in modo attivo (risiedendo sempre nella memoria) e quelle non utilizzate (e quindi da rimuovere dalla memoria). In molti casi, una sorta di algoritmo non usato di recente, determina le pagine che possono essere rimosse dai working set dei processi.

L'algoritmo base dell'insieme di lavoro è ingombrante poiché ad ogni page fault deve essere analizzata tutta la tabella delle pagine, fino a quando non viene trovata una pagina candidata adatta.

L'algoritmo WSClock (working set clock) è un algoritmo perfezionato, basato sull'algoritmo dell'orologio ma che utilizza anche le informazioni dell'insieme di lavoro; per la sua semplicità di implementazione e per le buone prestazioni è utilizzato largamente nella pratica.

La struttura dati necessaria è una lista circolare di pagine fisiche, come nell'algoritmo dell'orologio. Inizialmente la lista è vuota; quando viene caricata la prima pagina, essa viene aggiunta alla lista e ogni volta che viene caricata una pagina, questa va a finire nella lista in modo da formare un anello. Ogni elemento contiene il campo relativo al tempo di ultimo utilizzo derivante dall'algoritmo base dell'insieme di lavoro, e contiene inoltre il bit R ed il bit M. Come nell'algoritmo dell'orologio, ad ogni page fault, la pagina puntata dalla lancetta viene esaminata per prima: se il bit R assume il valore 1, la pagina è stata usata durante il tick corrente, quindi non è una buona candidata ad essere rimossa, il bit R viene quindi impostato a 0, quindi si passa ad esaminare la pagina successiva e l'algoritmo viene ripetuto per quella pagina.

Ricapitolando la politica di funzionamento della memoria virtuale con tabella dei frame liberi avviene così:

1. Il gestore dei frame liberi viene attivato dal gestore della memoria virtuale quando il numero di frame di pagina che sono liberi all'interno della lista scendono sotto una soglia predefinita di sicurezza.
2. Il gestore dei frame liberi va a scandire le pagine che sono in memoria e va a cercare qualche pagina che può essere rimossa per ottenere qualche frame e la aggiunge alla lista dei frame liberi che può essere disponibile per caricare nuove pagine
3. Se la pagina individuata dal frame del gestore dei frame liberi è dirty (modificata) viene sottoposta a page out del gestore dell'I/O
4. Il gestore di I/O Esegue il reset dei bit di validità della pagina che prima stava allocata al frame cercato
5. il gestore del page fault viene attivato dall'interrupt della MMU che chiama il page fault handler.
6. Cerca la pagina che ha causato il page fault nei frame liberi nella lista dei frame liberi (che può essere messo in sovrascrittura dal gestore dei frame liberi).
7. Se si trova li, rimuove il frame dalla lista dei frame liberi, aggiorna il bit di validità e lo ricarica in memoria (con il dirty bit modificato).
8. Se non si trova nella tabella dei frame liberi deve prelevare un frame libero dalla lista qualsiasi e a questo punto viene avviata l'operazione page-in
9. viene preso dal disco la pagina richiesta e messa nel frame trovato in memoria.

Ricapitolando e andando in generale, il gestore della memoria virtuale mantiene una lista di frame liberi e prova a tenere pochi frame in questa lista a ogni istante di tempo. Esso consiste di tre thread (deamon):

- gestore dei frame liberi (la politica di sostituzione di pagina è implementata è implementata qui)

- gestore dell'I/O della pagina
- gestore page fault

FILE SYSTEM è quella parte del SO che si occupa complessivamente dei file poiché tutte le applicazioni hanno bisogno di memorizzare e rintracciare informazioni e dati.

Durante l'esecuzione, un processo può memorizzare nella RAM solo una parte di queste informazioni, la restante parte è memorizzata su dischi e/o supporti esterni.

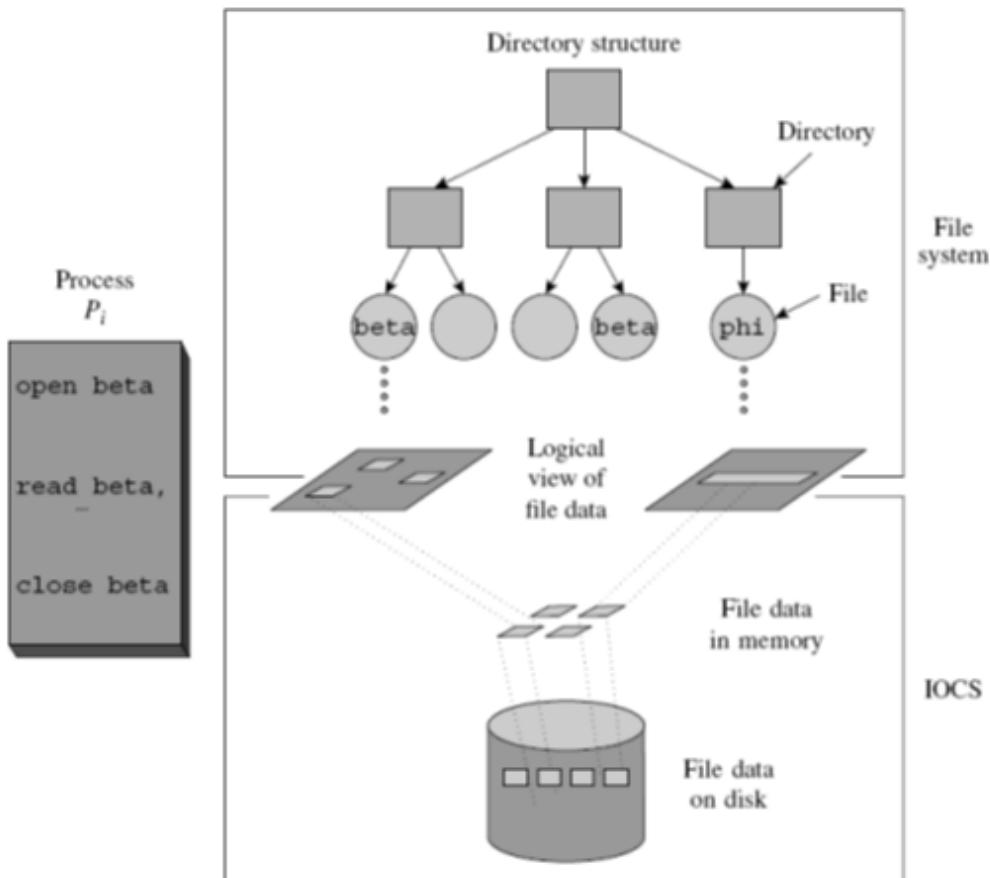
Gli utenti, invece, richiedono efficienza nella creazione e nella manipolazione dei file e nella condivisione degli stessi con altri utenti del sistema. Inoltre, richiedono che il file system implementi caratteristiche di protezione, sicurezza ed affidabilità in modo che i propri file non siano soggetti ad accessi illegali da parte di altri utenti o siano danneggiati da malfunzionamenti del sistema.

Pertanto, gli obiettivi principali di un file system sono:

- accesso conveniente e veloce ai file
- memorizzazione affidabile dei file
- condivisione dei file con altri utenti

Per raggiungere questi obiettivi in maniera efficace, il file system è strutturato in due strati:

- i moduli del file system che si occupano della condivisione, della protezione e dell'affidabilità dei file:
 - Strutture delle directory per il raggruppamento conveniente dei file
 - Protezione dei file contro gli accessi illeciti
 - Semantica condivisione dei file per gli accessi concorrenti a un file
 - Memorizzazione affidabile dei file
- l'IOCS (I/O Control System) che si occupa dell'implementazione delle operazioni sui file:
 - Funzionamento efficiente dei dispositivi di I/O
 - Accesso efficiente ai dati in un file



A livello di linguaggio di programmazione, un file è un oggetto che possiede attributi che descrivono l'organizzazione dei suoi dati e il metodo di accesso agli stessi. Un programma contiene le istruzioni per l'elaborazione dei file, cioè l'istruzione per dichiarare un file, per specificare i suoi attributi, per aprirlo, per eseguire le operazioni di

lettura/scrittura e per chiuderlo. Durante l'esecuzione del programma, l'elaborazione dei file è di fatto implementata dai moduli di libreria del file system e del sistema IOCS.

Col termine elaborazione dei file si indica la sequenza generale delle operazioni di:

- apertura del file
- lettura dei dati dal file o scrittura dei dati nel file
- chiusura del file

Esempio:

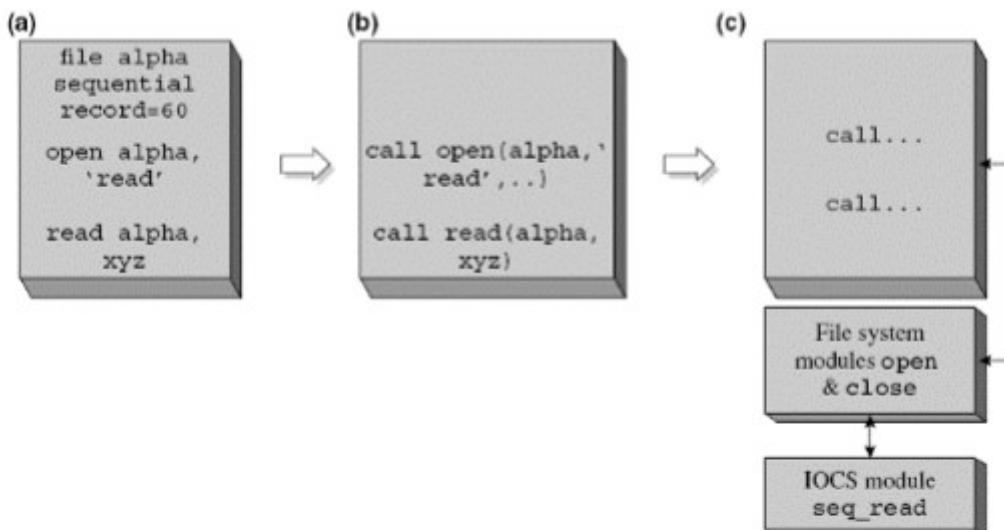


Figure 13.2 Implementing a file processing activity: (a) program containing file declaration statements; (b) compiled program showing calls on file system modules; (c) process invoking file system and IOCS modules during operation.

Un file system consiste di due tipi di dati:

- i dati contenuti nei file e i dati usati per accedere ai file. I dati contenuti nei file vengono chiamati dati;
- i dati usati per accedere ai file sono chiamati metadati (o dati di controllo).

Un file system fornisce diversi tipi di file:

- File strutturato, ovvero può contenere record di dati: o è una collezione di record. Un record è una collezione di campi e un campo contiene un singolo elemento dei dati. Si assume che ogni record in un file contenga un campo chiave, il cui valore è unico nel file, cioè non esistono due record che contengono la stessa chiave.
- File non strutturato, detto anche stream di byte: è un file che non contiene né record né campi, ma viene visto semplicemente come una sequenza di byte dai processi che lo usano.

Ogni tipo di file fornisce la propria vista astratta dei dati in un file, che chiamiamo vista logica dei dati. L'IOCS organizza i dati di un file in un dispositivo di I/O relativamente al tipo di file. Questa è denominata vista fisica dei dati di un file. L'IOCS effettua la mappatura tra la vista logica e la vista fisica dei dati del file. Inoltre, fornisce un'organizzazione che velocizza l'attività di elaborazione di un file.

Un attributo di un file è un'importante caratteristica sia per gli utenti che per il file system. Per comodità degli utenti, ogni file ha un nome che si usa come riferimento. Alcuni sistemi, nella composizione dei nomi, distinguono tra maiuscole e minuscole, altri le considerano equivalenti.

Un file ha altri attributi che possono variare secondo il SO, ma che tipicamente comprendono:

- Nome – Il nome simbolico del file è l'unica informazione in forma umanamente leggibile
- Identificatore – Si tratta solitamente di un numero che identifica il file all'interno del file system
- Tipo – Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi
- Dimensione – Si tratta della dimensione corrente del file e eventualmente, di quella massima
- Posizione sul disco – Si tratta del "percorso" in cui è memorizzato il file
- Protezione – Le informazioni utili per l'accesso al file in lettura, scrittura ed esecuzione

Durante l'elaborazione di un file, il file system usa gli attributi di un file per localizzarlo e assicurare che ogni operazione da effettuare su di esso sia consistente con gli attributi.

Le operazioni sui file sono:

Operazione	Descrizione
Apertura di un file	Il file system recupera l'elemento della directory corrispondente al file e controlla se l'utente il cui processo sta cercando di aprire il file ha i privilegi di accesso necessari per il file. Successivamente, esegue alcune azioni di gestione per avviare l'elaborazione del file.
Leggere o scrivere un record	Il file system considera l'organizzazione del file (Paragrafo 13.3) e implementa le operazioni di lettura/scrittura in maniera appropriata.
Chiusura di un file	L'informazione relativa alla dimensione del file nell'elemento della directory relativo al file viene aggiornata.
Copia di un file	Viene eseguita una copia del file, viene creato un nuovo elemento della directory per la copia e il suo nome, la sua dimensione, la posizione e le informazioni di protezione vengono memorizzate nella voce corrispondente.
Cancellazione del file	L'elemento della directory relativo al file viene cancellato e l'area sul disco occupata viene liberata.
Ridenominazione del file	Il nuovo nome viene registrato nell'elemento della directory relativo al file.
Specificare i privilegi di accesso	Le informazioni di protezione contenute nell'elemento della directory relativo al file vengono aggiornate.

Ricordiamo che le operazioni come apertura, chiusura, ridenominazione e cancellazione sono eseguite dai moduli del file system, mentre l'accesso (lettura/scrittura) viene implementato dai moduli del sistema IOCS in modo adatto alla tipologia di file.

Un'organizzazione dei file è una combinazione di due caratteristiche:

- Un modo per organizzare i record in un file.

L'organizzazione dei file viene scelta in base alle caratteristiche del dispositivo di I/O che si usa, in modo tale da fornire un accesso efficiente:

- Un hard disk può accedere direttamente, mediante l'indirizzo, a qualunque record,
- Un drive a nastro può accedere al record solo in modo sequenziale.

Ci sono tre principali organizzazioni dei file strutturati usati dal file system: organizzazione sequenziale dei file (gli accessi ai file sono implementati da un modulo del sistema IOCS chiamato metodo di accesso):

1. Organizzazione sequenziale dei file :

Nell'organizzazione sequenziale dei file le informazioni sono memorizzate in ordine crescente o decrescente in base al campo chiave. Di conseguenza l'elaborazione di queste informazioni supporta solo due operazioni:

- a. legge l'informazione (record) successiva (o precedente)
- b. salta l'informazione (record) precedente (o successiva)

Un file ad accesso sequenziale viene utilizzato nelle applicazioni se i suoi dati possono essere preordinati convenientemente in ordine crescente o decrescente.

2. Organizzazione diretta dei file:

L'organizzazione diretta dei file fornisce efficienza e convenienza perché permette di accedere alle informazioni (record) in ordine casuale. Di conseguenza l'elaborazione di queste informazioni permette di leggere o scrivere byte senza alcun ordine particolare. Il metodo di accesso diretto si fonda su un modello di file che si rifà al disco (i dischi permettono, infatti, l'accesso diretto ad ogni blocco di file).

In questa organizzazione occorre generare l'indirizzo del record usato dalla periferica. Se il file è memorizzato su un hard disk, la trasformazione genera un indirizzo (num_traccia, num_record) in modo tale che le testine dell'hard disk vengono posizionate sulla traccia num_traccia prima che venga eseguita l'operazione di lettura o scrittura sul record num_record.

Abbiamo detto che questo tipo di organizzazione fornisce efficienza e convenienza, ma presenta anche due svantaggi rispetto all'organizzazione sequenziale:

- a. - il calcolo dell'indirizzo del record consuma tempo di CPU
- b. - una parte della memoria viene sprecata in quanto vengono memorizzati meno dati sulla traccia esterna del disco rispetto a quelli realmente memorizzabili

3. Organizzazione indicizzata dei file

Nell'organizzazione indicizzata dei file un indice aiuta a determinare la posizione di un record a partire dal valore della sua chiave. Ci sono due principali varianti:

- a. Nell'organizzazione indicizzata pura,
esiste un indice per ogni record. Un indice di un file è costituito dalla coppia (valore della chiave, indirizzo del disco). Per accedere a un record con chiave k, viene trovato l'elemento indice contenente k tramite la ricerca per indice, e si utilizza l'indirizzo del disco annotato nell'elemento trovato per accedere al record.
- b. L'organizzazione sequenziale indicizzata
è un'organizzazione ibrida che combina gli elementi delle organizzazioni dei file indicizzata e sequenziale: esiste un indice per ogni sezione del disco. Per accedere ad un record, si cerca un indice che punta ad una sezione del disco che può contenere il record. Poi si effettua la ricerca sequenziale dei record in questa sezione del disco per trovare il record desiderato. La ricerca ha buon esito se il record è presente nel file; altrimenti avrà esito negativo. Questa organizzazione richiede un indice molto più piccolo rispetto all'indicizzazione pura poiché l'indice contiene elementi solo per alcuni valori della chiave.
Per un file di grandi dimensioni l'indice può contenere un gran numero di elementi, per cui il tempo richiesto per la ricerca mediante l'indice può risultare molto lungo. Si può utilizzare una "gerarchia di indici": un indice di livello più alto può essere utilizzato per ridurre il tempo di ricerca; un elemento nell'indice di livello più alto punta a una sezione dell'indice.

• Una procedura per accedervi.

Esistono vari modi per accedere ai file salvati sul computer. Chiamiamo "pattern di accesso al record" l'ordine in cui un processo accede ai record in un file. I due metodi fondamentali di accesso ai record sono:

- c. Accesso sequenziale - in cui l'accesso ai record avviene nell'ordine in cui si trovano nel file (o nell'ordine inverso).
- d. Accesso casuale - secondo cui si può accedere ai record in qualsiasi ordine.

Un metodo di accesso dei file è un modulo del sistema IOCS che implementa gli accessi a una classe di file che utilizza una specifica organizzazione dei file. Il tipo di accesso da utilizzare per accedere ai record in un file dipende dall'organizzazione del file. E' possibile utilizzare alcune tecniche di programmazione dell'I/O per rendere più efficiente l'accesso al file:

- BUFFERING DEI RECORD: Il metodo di accesso legge i record di un file di input prima che siano effettivamente richiesti da un processo e li memorizza temporaneamente in aree di memoria chiamate buffer finché non vengono richiesti dal processo. Lo scopo del buffering è quello di ridurre o eliminare l'attesa per il completamento di un'operazione di I/O.
- BLOCKING DEI RECORD: Il metodo di accesso legge o scrive sempre grandi blocchi di dati, che contengono diversi record di file, da o verso un dispositivo di I/O. Questa caratteristica riduce il numero totale di operazioni di I/O richieste per elaborare un file, migliorando di conseguenza l'efficienza nell'elaborazione di un file da parte di un processo. Il blocking, inoltre, migliora l'utilizzo di un dispositivo di I/O e il throughput di un dispositivo.

Per la protezione dei file il SO deve anche implementare la protezione dei file, in particolare deve dare la possibilità ad un utente di voler condividere i suoi file solo con una parte degli altri utenti del sistema. Questa esigenza è detta condivisione controllata e viene implementata in questo modo: il proprietario del file specifica quali utenti possono accedere al file e in che modo. Queste informazioni sono contenute nel campo Protection_Info dell'elemento della directory relativo al file. L'informazione relativa alla protezione è solitamente memorizzata nella forma di una access control list (ACL). Ogni elemento della ACL è una coppia (nome utente, lista dei privilegi di accesso). Quindi prima di eseguire un'operazione viene anche controllata l'informazione relativa alla protezione, in particolare viene controllato se l'utente può accedere a quel file, e in tal caso, con quale accesso.

La dimensione della ACL di un file dipende dal numero di utenti e dal numero di privilegi di accesso definiti nel sistema. Per ridurre le dimensioni, è possibile specificare una ACL per ogni classe di utenti piuttosto che per ogni utente. In questo modo una ACL ha solo tante coppie quante sono le classi di utenti. Per esempio, Unix specifica i privilegi di accesso per tre classi di utenti: il proprietario del file, gli utenti nello stesso gruppo del proprietario, tutti gli altri utenti del sistema. In molti file system i privilegi sono di tre tipi: lettura, scrittura ed esecuzione.

Struttura della directory:

Campo	Descrizione
Nome del file	Nome del file. Se questo campo ha dimensione fissa, i nomi lunghi oltre una certa lunghezza saranno troncati.
Tipo e dimensione	Il tipo e la dimensione del file. In molti file system, il tipo di file è implicito nella sua estensione; per esempio, un file con estensione .c è un file che contiene un programma C e un file con estensione .obj è un file oggetto, che spesso è un file strutturato.
Informazioni sulla posizione	Informazioni sulla posizione del file sul disco. Queste informazioni sono tipicamente sotto forma di una tabella o di una lista concatenata contenente gli indirizzi dei blocchi sul disco allocati al file.
Informazioni sulla protezione	Le informazioni relative agli utenti cui è concesso l'accesso a questo file e in che modo.
Open count	Numero di processi che attualmente accedono al file.
Lock	Indica se un processo sta accedendo al file in maniera esclusiva.
Flag	Informazioni sulla natura del file, quali se il file è una directory, un link o un file system montato.
Misc info	Informazioni varie come l'id del proprietario, la data e l'ora della creazione, l'ultimo accesso e l'ultima modifica.

La figura mostra i campi di un tipico elemento di una directory. I campi Open count e Lock sono usati quando diversi processi aprono un file in maniera concorrente. Il campo Lock viene usato quando un processo richiede l'accesso esclusivo al file. Il campo Flag è usato per differenziare i diversi tipi di file di una directory (D=directory, L=link, M=file system). Il campo Misc_Info contiene informazioni aggiuntive come il proprietario, la data di creazione e ultima modifica

L'organizzazione delle directory tiene conto che una directory contiene i file appartenenti a diversi utenti, quindi deve garantire due importanti prerogative:

- libertà nella scelta del nome – possibilità per gli utenti di dare gli stessi nomi ai propri file
- condivisione dei file – possibilità per un utente di accedere ai file creati da altri utenti e possibilità di concedere ad altri utenti il permesso di accedere ai propri file.

Esistono vari tipi di organizzazioni delle directory:

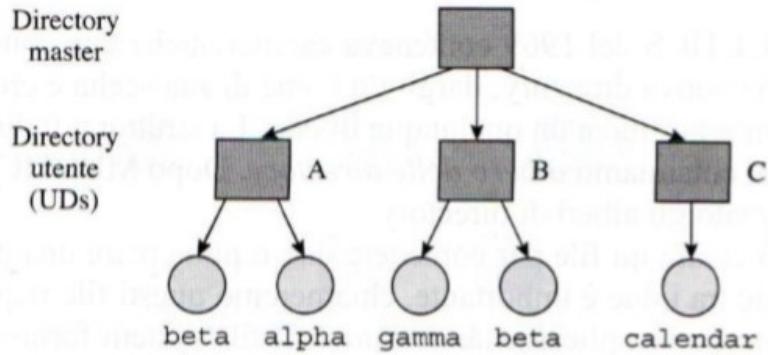
1. Struttura di directory a due livelli

dove con i rettangoli sono indicate le directory, mentre con i cerchi sono indicati i file. Questa struttura presenta due tipi di directory:

- la directory master contiene informazioni relative alle directory utente di tutti gli utenti del sistema; ogni elemento di un directory master è una coppia che consiste di un ID utente e di un puntatore a una directory utente
- una directory utente (UD) contiene elementi che descrivono i file appartenenti a un utente

Questa soluzione richiede che i nomi siano unici solo all'interno dell'area del singolo utente. Infatti, l'uso di UD separate permette la libertà nell'assegnazione di un nome ad un file, infatti la figura mostra che vi sono più file con lo stesso nome (beta). Una tale organizzazione garantisce l'accesso al file corretto anche se nel sistema esistono molti file con lo stesso nome.

Tuttavia, l'utilizzo delle UD ha uno svantaggio: impedisce agli utenti di condividere i loro file con altri utenti. Per raggiungere tale obiettivo sono necessarie istruzioni speciali che permettono a degli utenti di accedere ai file di altri utenti; per fare ciò occorre verificare i permessi dei file (che si trovano nel campo Prot_Info).

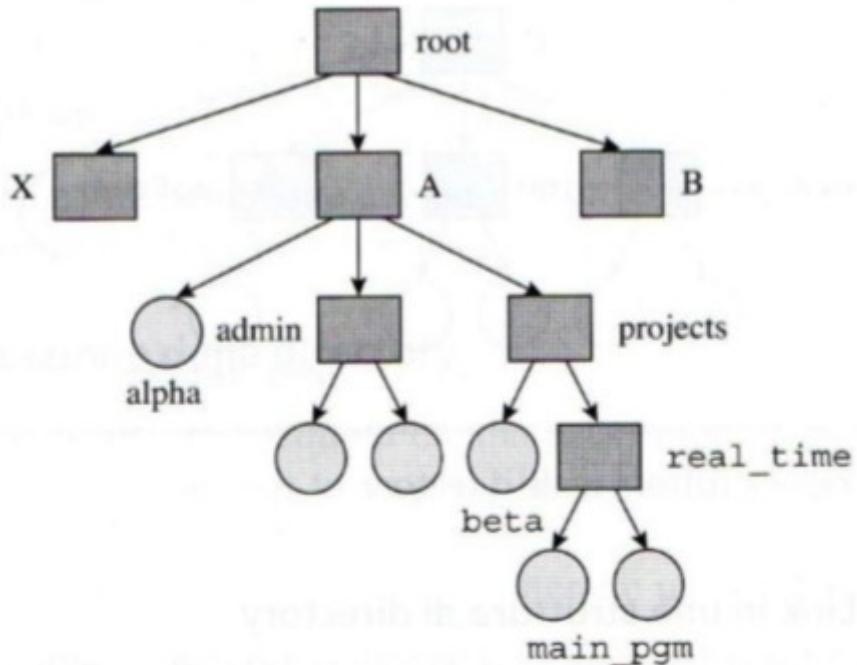


A,B,C sono sottodirectory degli utenti A,B,C

2. Struttura gerarchica o ad albero delle directory

è un approccio più potente e flessibile, in questa struttura, il file system fornisce all’utente una directory chiamata root che contiene la directory home per ciascun utente, ovvero una directory che, solitamente, ha lo stesso nome del nome utente. Un utente può creare file oppure può creare directory all’interno della sua home, con la possibilità di organizzare le proprie informazioni in una struttura basata su diversi livelli di sottodirectory. Ad ogni istante, si dice che un utente si trova in una specifica directory, chiamata directory corrente. Quando l’utente vuole aprire un file, il file viene cercato in questa directory. Quando l’utente effettua il login, il SO permette all’utente di operare solo nella sua directory home. Il nome dato a uno specifico file può non essere unico nel file system, per cui un utente o un processo utilizza un path per identificarlo in maniera univoca. Il path è il percorso di tutte le sottodirectory in cui si trova il file. Nel path ogni riferimento è un directory tranne l’ultima che è, appunto, il file stesso.

- I path relativi sono path per localizzare un file a partire dalla directory corrente , che sono spesso corti e convenienti da usare; tuttavia, possono essere fonte di confusione poiché un file può avere diversi path relativi quando vi si accede a partire da diverse directory correnti.
- Il path assoluto di un file parte dalla directory root dell’albero delle directory del file system. I file con lo stesso nome creati in directory differenti differiscono nei rispettivi path assoluti.

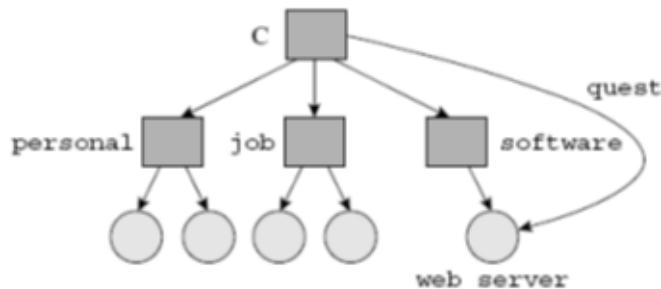


3. Struttura a grafo

In un albero di directory, ogni file eccetto la directory root ha esattamente una directory genitore. Una struttura del genere separa totalmente i file di utenti differenti; in pratica non ammette la condivisione di file e directory tra più utenti. Questo problema può essere risolto organizzando le directory in una struttura a grafo aciclico. Un grafo aciclico permette di avere sottodirectory e file condivisi. Il fatto che il file sia condiviso non significa che ci siano due copie del file. In questa struttura, un file può avere molte

directory genitore, per cui un file condiviso può essere puntato dalle directory di tutti gli utenti che hanno accesso ad esso. Le strutture a grafo aciclico possono essere implementate in vari modi.

(`~C, ~C/software/web_server, quest`)



(Un metodo molto diffuso prevede la creazione di un nuovo elemento di directory, chiamato link (o collegamento). Un link è un puntatore ad un altro file o directory, in pratica è una connessione diretta tra due file esistenti.)

Una directory dovrebbe essere costituita da una lista lineare in modo da poter effettuare ricerche lineari per trovare il file richiesto. Tuttavia, questa organizzazione è inefficiente se la directory contiene un elevato numero di elementi. Per avere una maggiore efficienza, cioè per diminuire notevolmente il tempo di ricerca nelle directory, sono usate organizzazioni che fanno uso di una tabella hash o di un albero B+.

Sulle directory è possibile, come nei file svolgere delle operazioni: La ricerca di informazioni o file è l'operazione più frequente sulle directory. Altre operazioni sulle directory sono operazioni come la creazione o la cancellazione di file, l'aggiornamento degli elementi relativi ai file quando questo viene chiuso da un programma, l'elencazione del contenuto di una directory o la cancellazione della stessa. L'operazione di cancellazione risulta complessa quando la struttura è un grafo poiché un file può avere molti genitori. Un file è cancellato quando ha un solo genitore; altrimenti viene semplicemente reso inaccessibile dalla directory che si vuole cancellare. Per semplificare l'operazione di cancellazione, il file system associa ad ogni file un contatore di link: il contatore è impostato a 1 quando viene creato il file, viene incrementato di 1 quando un link punta al file, viene decrementato di 1 quando viene eseguita una cancellazione. Il file viene cancellato definitivamente quando il contatore dei link diventa 0.

Con l'allocazione dello spazio su disco un disco può contenere molti file system, ognuno nella sua partizione del disco. Il file system ha informazioni sulla partizione in cui è presente un file, ma il sistema IOCS no; dunque, l'allocazione dello spazio sul disco è eseguita dal file system.

Esistono principalmente 2 tipi di Allocazione:

- Allocazione contigua (non più usata): I primi file system usavano il modello di allocazione contigua della memoria, cioè allocavano una singola area di memoria a ogni file al momento della creazione. In questa allocazione ogni file occupa un insieme di blocchi contigui sul disco, quindi questa allocazione risulta essere particolarmente semplice poiché basta conoscere il blocco iniziale e la lunghezza del file.
Questa allocazione porta però alla frammentazione esterna, cioè si hanno aree di memoria troppo piccole per poter essere riutilizzate; ma portava anche alla frammentazione interna poiché il file system era progettato per allocare spazio extra sul disco per consentire al file di crescere.
- Allocazione non contigua (usata): I moderni file system adottano l'allocazione non contigua della memoria per l'allocazione dello spazio sul disco. In questo approccio, una parte di spazio sul disco è allocata su richiesta, ovvero, quando viene creato un file oppure quando la sua dimensione aumenta a seguito di un'operazione di aggiornamento. Il file system deve risolvere tre problemi per implementare questo approccio:
 1. gestione dello spazio libero sul disco: deve tenere traccia dello spazio libero sul disco e allocarlo quando un file richiede un nuovo blocco del disco. Utilizza una free list o DSM (Disk Status Map) che indica per ogni blocco del disco sia allocato oppure no (a seconda se il bit DSM è libero oppure no), questa tecnica viene anche detta bitmap.

2. evitare movimenti eccessivi della testina del disco: garantire che un file non sia “sparpagliato” in diverse parti del disco, poiché ciò causerebbe un movimento eccessivo delle testine del disco durante l’elaborazione del file. Si risolve come
3. accesso ai dati del file: mantenere le informazioni sui file per trovare i blocchi del disco che lo contengono

L’allocazione non contigua può essere implementata in modi differenti:

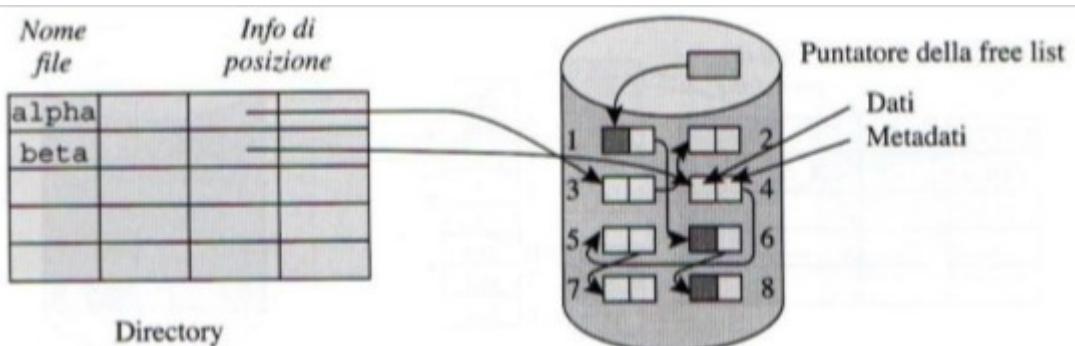
- **Allocazione concatenata**

L’allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni del file, entrambi presenti nell’allocazione contigua della memoria. Tuttavia, in mancanza di una FAT, l’assegnazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine. In questa allocazione, ogni file è rappresentato da una lista concatenata di blocchi del disco, che possono essere sparpagliati ovunque sul disco. Ogni blocco del disco ha due campi al suo interno:

- dati, che contiene, appunto, i dati
- metadati, che è un campo di tipo link e punta al prossimo blocco

Il campo *Info_di_posizione* dell’elemento della directory punta al primo blocco sul disco del file. Agli altri blocchi si accede seguendo i puntatori dei vari blocchi. L’ultimo blocco del disco contiene un’informazione null nel campo metadati.

Esempio:



Lo spazio libero sul disco è rappresentato da una free list in cui ogni blocco libero contiene un puntatore al successivo. Quando viene richiesto un blocco, viene estratto un blocco dalla free list per poi essere aggiunto alla lista dei blocchi del file. Per cancellare un file, la lista di blocchi del file viene semplicemente aggiunta alla free list. Ad esempio, nella figura:

- il file alpha è costituito da due blocchi, il numero 3 e il numero 2;
- il file beta è costituito da tre blocchi, il numero 4, il 5, seguito dal 7 - la free list è di tre blocchi, l’1, il 6 e l’8.

VANTAGGI: Il vantaggio principale di questa allocazione è che è sufficiente memorizzare in ogni elemento della directory solamente l’indirizzo su disco del primo blocco, mentre la parte rimanente si può trovare a partire da esso. Questo porta anche ad una lettura sequenziale semplice da effettuare.

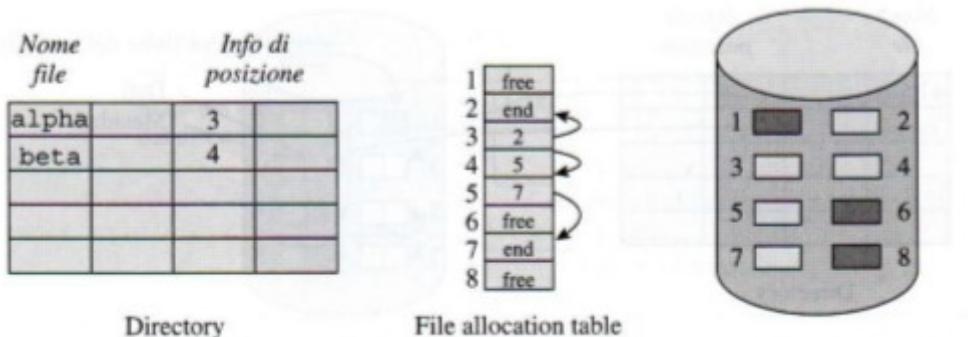
SVANTAGGI: Tuttavia ci sono anche degli svantaggi. L’accesso diretto è estremamente lento, perché per arrivare al blocco n occorre necessariamente aver letto gli n-1 blocchi che lo precedono. Un ulteriore svantaggio riguarda lo spazio richiesto per i puntatori. La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi continui in gruppi (cluster) e nell’assegnare i gruppi di blocchi anziché i singoli blocchi. Un altro problema riguarda l’affidabilità. Se si danneggiasse il campo metadati di un blocco, cioè se si danneggiasse il puntatore, i dati successivi al blocco danneggiato potrebbero essere persi. Ci sono delle soluzioni a questo problema, come l’uso di liste doppiamente concatenate, che però sono onerose da implementare.

- **FILE ALLOCATION TABLE (FAT)**

Una variante importante del metodo di assegnazione concatenata consiste nell’uso della tabella di assegnazione dei file, una tabella di questo tipo, tenuta in memoria, si chiama FAT (File Allocation Table). Per contenere tale tabella si riserva una sezione del disco all’inizio di ciascuna partizione; la FAT ha un elemento per ogni blocco del disco. Per un blocco allocato a un file, il corrispondente elemento della FAT

contiene l'indirizzo del blocco successivo. In questo modo il blocco e il suo elemento nella FAT insieme formano una coppia che contiene la stessa informazione contenuta nel blocco nel classico schema dell'allocazione concatenata.

L'elemento di una directory relativo a un file contiene l'indirizzo del primo blocco sul disco. L'elemento della FAT corrispondente a questo blocco contiene l'indirizzo del secondo blocco e così via. L'elemento della FAT corrispondente all'ultimo blocco contiene un valore speciale di fine file.



La figura illustra la FAT per il disco della figura precedente. Ad esempio:

- il file alpha si compone dei blocchi 3 e 2; il campo Info_di_posizione contiene 3; l'elemento della FAT relativo al blocco 3 contiene 2 e l'elemento della FAT relativo al blocco 2 indica che il file termina con quel blocco
- il file beta si compone dei blocchi 4, 5 e 7; il campo Info_di_posizione contiene 4; l'elemento della FAT relativo al blocco 4 contiene 5, e così via.

La FAT può anche essere usata per memorizzare l'informazione relativa allo spazio libero. La lista dei blocchi liberi è costruita nello stesso modo in cui viene costruita la lista dei blocchi di un file. In alternativa, per ogni blocco libero, può essere utilizzato un valore speciale (free) che indica che quel blocco è libero.

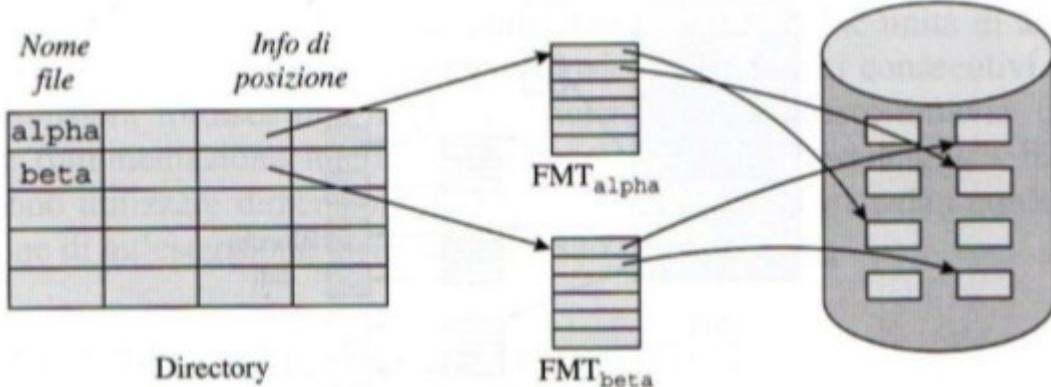
VANTAGGI: L'uso della FAT fornisce maggiore affidabilità rispetto alla classica allocazione concatenata poiché il danneggiamento di un blocco contenente i dati del file comporta danni limitati.

SVANTAGGI: In questa allocazione, il danneggiamento di un blocco usato per memorizzare la FAT risulta disastroso. Inoltre, le prestazioni sono peggiori poiché è necessario accedere alla FAT per ottenere l'indirizzo del blocco successivo.

o Allocazione Indicizzata

risolve il problema dell'accesso diretto, presente nell'allocazione concatenata, raggruppando tutti i puntatori in una sola locazione: il blocco indice. Nell'allocazione indicizzata si mantengono tutti i puntatori ai blocchi di un file in una tabella indice chiamata file map table (FMT). In questa tabella sono riportati gli indirizzi dei blocchi del disco allocati a un file.

Ogni file ha il proprio blocco indice (o tabella indice): nella sua forma più semplice, un FMT è un array di indirizzi di blocchi del disco. Ogni blocco ha un solo campo, il campo dati. L'i-esimo elemento del blocco indice punta all'i-esimo blocco del file.



Il campo Info_di_posizione dell'elemento della directory relativo a un file contiene l'indirizzo della FMT, cioè punta alla FMT. Una volta creato il file, tutti i puntatori del blocco indice sono impostati a null.

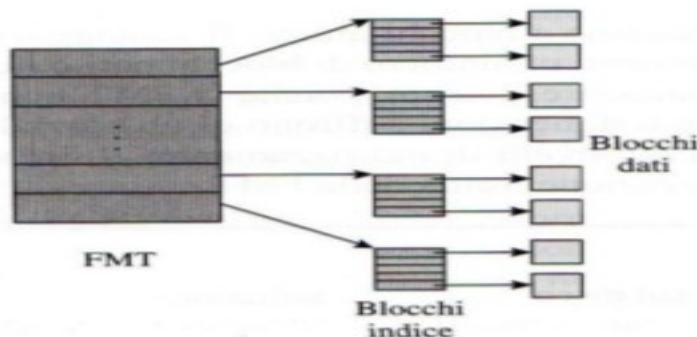
Quando le dimensioni del file crescono, viene localizzato un blocco libero e l'indirizzo di questo blocco viene aggiunto alla FMT del file.

VANTAGGI: Questa allocazione permette di accedere direttamente a un blocco del file direttamente dalla FMT senza avere frammentazione esterna. Inoltre, l'affidabilità è migliore in quanto il danneggiamento di un elemento della FMT non compromette l'intero file, ma solo una parte di esso.

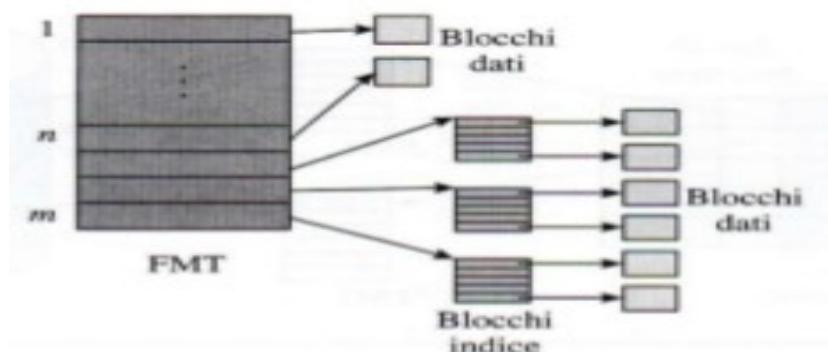
SVANTAGGI: Il problema principale consiste nella dimensione del blocco indice, cioè della FMT. Se essa è troppo piccola non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione.

- **INDICE A PIU' LIVELLI**

Una prima soluzione consiste nell'utilizzo di un blocco indice a più livelli. In questa organizzazione, ogni elemento della FMT contiene l'indirizzo di un blocco indice. Un blocco indice non contiene dati; contiene elementi che contengono gli indirizzi dei blocchi dati. Per accedere ai blocchi dati, prima accediamo a un elemento della FMT e otteniamo l'indirizzo di un blocco indice. Successivamente, accediamo a un elemento del blocco indice per ottenere l'indirizzo di un blocco dati.



- **IBRIDA FMT** che include alcune delle caratteristiche dell'allocazione classica e dell'allocazione indicizzata multilivello. I primi elementi nella FMT, ad esempio n elementi, puntano a blocchi dati come nell'allocazione indicizzata. Gli altri elementi puntano a blocchi indice.
Il vantaggio di questa organizzazione è che piccoli file continuano a essere accessibili in maniera efficiente, poiché la FMT non utilizza i blocchi indice. I file di medie o grandi dimensioni soffrono di un parziale degrado delle prestazioni di accesso a causa dei livelli di indirizzamento.



Affidabilità del file system è il grado di funzionamento corretto di un file system, anche quando si verificano malfunzionamenti come la corruzione dei dati nei blocchi del disco e fault di sistema dovuti a corrente.

I due aspetti principali dell'affidabilità del file system sono:

- garantire la correttezza della creazione, cancellazione e aggiornamento dei file: riguarda la consistenza e la correttezza dei metadati, ovvero i dati di controllo del file system
- prevenire la perdita dei dati contenuti nei file: riguarda la consistenza e la correttezza dei dati memorizzati nei file.

Quando si parla di affidabilità bisogna tenere conto:

- del **fault** (o guasto) è un difetto in qualche parte del sistema
- del **failure** (o insuccesso) è un comportamento erroneo, o che differisce dal comportamento atteso.

L'occorrenza di un fault causa un failure.

Tecniche di correzione dei guasti.

Quando l'esecuzione di un file system viene terminata dall'utente del sistema, il file system copia tutti i dati e i metadati dalla memoria RAM sul disco, in modo che la copia sul disco sia completa e consistente. Tuttavia, quando si verifica una mancanza di corrente elettrica o quando il sistema viene spento all'improvviso, il file system non ha l'opportunità di copiare i file dati e i metadati sul disco. Questo spegnimento è detto spegnimento sporco e causa una perdita dei dati dei file e dei metadati contenuti in memoria.

Tradizionalmente, i file system utilizzavano delle tecniche di ripristino per proteggersi contro la perdita di dati e metadati poiché erano molto semplici da implementare. Per questo motivo, si creavano backup periodici e i file erano ripristinati a partire dalle copie di backup nel momento in cui si verificavano dei malfunzionamenti l'overhead per la correzione delle inconsistenze era elevato ed inoltre, il sistema non era disponibile durante il ripristino.

Per ridurre l'overhead per la creazione di backup (quando è usata l'allocazione indicizzata) sono copiati solo l'FMT ed il blocco del disco i cui contenuti sono aggiornati dopo aver creato il backup. Praticamente risparmia sia spazio su disco che tempo

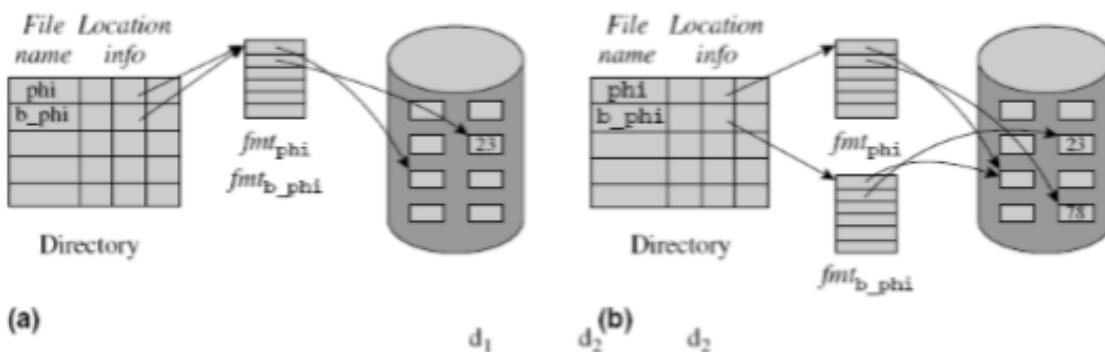


Figure 13.27 Creating a backup: (a) after backing up file *phi*; (b) when *phi* is modified.

Un file system moderno utilizza tecniche di fault tolerance in modo da poter riprendere l'esecuzione velocemente dopo uno spegnimento improvviso. Un journaling file system implementa la fault tolerance mantenendo un journal, un diario giornaliero, dove vengono salvate le azioni che il file system si accinge ad eseguire prima di eseguirle effettivamente. Quando l'esecuzione del file system viene ripristinata dopo uno spegnimento improvviso, il file system consulta il journal per identificare le azioni non ancora eseguite e le esegue, garantendo in questo modo la correttezza dei dati dei file e dei metadati. L'uso di tecniche di fault tolerance genera un elevato overhead. Per questo motivo il JFS offre diverse modalità journaling.

L'affidabilità del FS può essere migliorata ricorrendo a due precauzioni:

- Prevenendo la perdita dei dati o metadati a causa di malfunzionamenti del dispositivo di I/O usando dispositivi stabili: Mantiene due copie dei dati. Può tollerare un guasto nella memorizzazione di un dato. Involvi un elevato overhead di spazio e tempo e non può indicare se una copia è vecchia o nuova.
- Prevenendo inconsistenza dei metadati dovute ai guasti usando azioni atomiche: consiste di un insieme di sotto-azioni la cui esecuzione ha la proprietà che gli effetti di tutte le sue sotto-azioni si compiono, oppure gli effetti di nessuna delle sue sotto-azioni si compiono. Per l'implementazione sono usate due strutture dati (gestite in dispositivi stabili):

Per l'implementazione sono usate due strutture dati (gestite in dispositivi stabili)

- *intention list* (entrate del tipo *<disk block id>, <nuovo contenuto>*)
 - Le entrate sono create ad ogni modifica di dati o metadati
- *commit flag* (due campi transaction id e valore)
 - Creato con **begin atomic action** (Ai, NC)
 - NC diventa C in corrispondenza di **end atomic action**
 - Cancellato quando tutti le modifiche della *intention list* sono eseguite

Algorithm 13.2 Implementation of an Atomic Action

1. Execution of an atomic action A_i :

- When the statement **begin atomic action** is executed, create a *commit flag* and an *intentions list* in stable storage, and initialize them as follows:
commit flag := (A_i , "not committed");
intentions list := "empty";
- For every file update made by a subaction, add a pair (d, v) to the intentions list, where d is a disk block id and v is its new content.
- When the statement **end atomic action** is executed, set the value of A_i 's *commit flag* to "committed" and perform Step 2.

2. Commit processing:

- For every pair (d, v) in the intentions list, write v in the disk block with the id d .
- Erase the commit flag and the intentions list.

3. On recovering after a failure:

If the commit flag for atomic action A_i exists,

- If the value in commit flag is "not committed": Erase the commit flag and the intentions list. Reexecute atomic action A_i .
- Perform Step 2 if the value in commit flag is "committed."

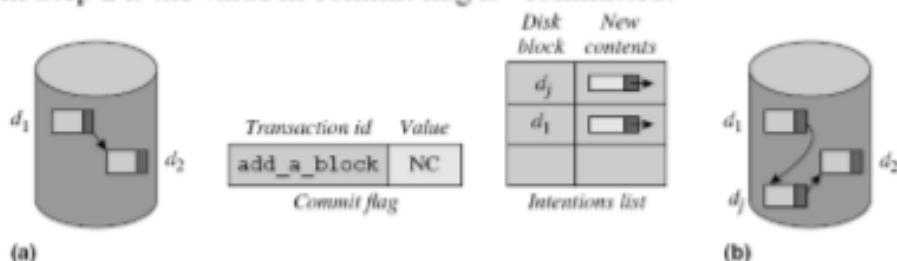


Figure 13.30 (a) Before and (b) after commit processing. (Note: NC means not committed.)

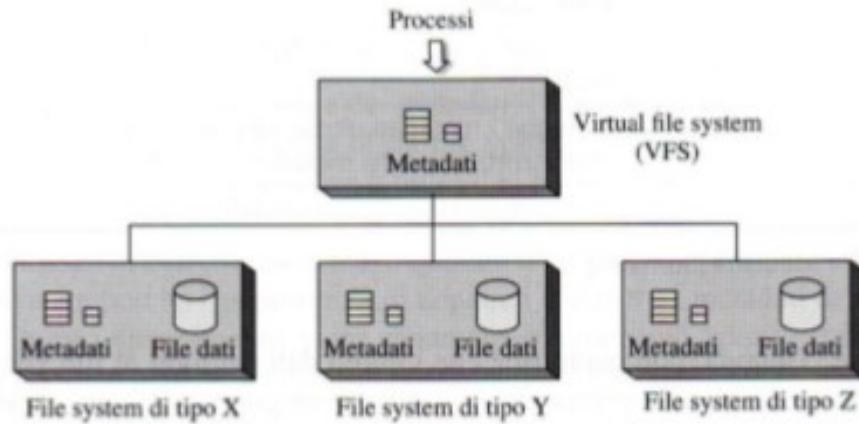
Un amministratore di sistema può scegliere una modalità journaling da adattare al tipo di affidabilità necessaria nell'ambiente di elaborazione.

Modalità	Descrizione
Write behind	Protegge solo i metadati. Non fornisce alcuna protezione per i dati.
Ordered data	Protegge i metadati. Protezione limitata per i file dati – vengono scritti prima dei metadati a essi relativi.
Full data	Protegge sia i dati che i metadati.

Gli utenti richiedono requisiti differenti a un file system, come convenienza, alta affidabilità, risposte veloci e accesso ai file su altri sistemi. Un singolo file system non può fornire tutte queste caratteristiche, per cui un sistema operativo fornisce un File System Virtuale (VFS) che facilita l'esecuzione simultanea di diversi file system. In questo modo ogni utente può usare il file system che preferisce.

In pratica un file system virtuale è un livello software che consente a diversi file system di essere in funzione su un computer simultaneamente, in modo che un utente possa scegliere il file system che è più adatto per le sue applicazioni.

Un processo invoca il livello VFS utilizzando comandi generali per l'accesso ai file e il livello VFS redireziona il comando al file system appropriato.



Ciò è implementato da un layer VFS situato tra un processo e un file system.

Il layer VFS ha due interfacce:

- un'interfaccia col file system
- un'interfaccia con i processi.

Ogni file system conforme alle specifiche dell'interfaccia del file system VFS può essere installato per funzionare con il VFS. Questa caratteristica rende facile aggiungere un nuovo file system.

L'interfaccia del VFS con il processo fornisce le funzionalità per eseguire le generiche operazioni sui file (come open, close, read, write) e le operazioni mount e umount sul file system.

L'interfaccia con il file system serve per determinare a quale file system appartiene il processo, invocando le operazioni open, close, read e write dello specifico file system.

Il FSV risulta molto utile con i dispositivi rimovibili come le penne USB o i CD/DVD in quanto consente all'utente di montare il file system presente in questi dispositivi nella sua directory corrente e accedere ai file senza preoccuparsi del fatto che i dati sono memorizzati in un formato differente.

IL File System di Unix contiene Le informazioni che costituiscono l'elemento della directory relativo a un file (nome, tipo, dimensione, locazione, protezione, open count, lock, flag, Misc Info), in Unix, sono divise tra l'elemento della directory e l'i-node del file. L'elemento della directory contiene solo il nome e il numero di inode; la maggior parte delle informazioni di un file è contenuta nel suo inode. Quindi, un file in Unix è rappresentato da un inode (nodo indice). In Unix, l'amministratore di sistema può specificare una quota disco per ogni utente. Questo impedisce a un utente di occupare uno spazio elevato su disco rigido oppure tutto lo spazio disponibile.

La struttura dati inode viene mantenuta sul disco. Essa contiene le seguenti informazioni:

- Tipo di file (directory, link o file speciale)
- Numero di link al file
- Dimensione del file
- ID del dispositivo su cui è memorizzato il file
- Numero seriale dell'inode
- ID utente e gruppo del proprietario
- Permessi di accesso
- Informazioni sull'allocazione

La divisione dell'elemento della directory tra l'elemento della directory e l'inode facilita la cancellazione dei link. Un file può essere cancellato quando il suo numero di link va a zero. In Unix, in memoria vengono mantenute gli inode, i descrittori di file e le strutture file. Una struttura file contiene due campi, la posizione corrente in un file aperto e un puntatore all'inode del file. In questo modo un inode e una struttura file insieme contengono tutte le informazioni necessarie per accedere al file. I descrittori di file sono memorizzati in una tabella per ogni processo.

Input Output Control System (IOCS): In particolare i moduli del file system consentono di avere libertà nella scelta dei nomi, la condivisione e la protezione dei file e l'affidabilità; le operazioni sui file vengono implementate dall'IOCS. Quando un processo effettua una richiesta di lettura o scrittura di dati da un file, il file system passa la richiesta al sistema IOCS ed avere un throughput elevato. Per fare ciò è suddiviso in 2 livelli:

- Il metodo di accesso gestisce la lettura e scrittura dei dati per rendere efficiente l'elaborazione di un file.

-L'IOCS fisico esegue l'I/O a livello di dispositivo e utilizza le politiche di scheduling per migliorare il throughput dei dispositivi di I/O (non è implementato nel kernel, quindi può usufruire dei suoi servizi attraverso SystemCall). Questa struttura dell'IOCS consente di separare le problematiche relative all'implementazione delle operazioni sui file a livello di processo da quelle a livello di dispositivo.

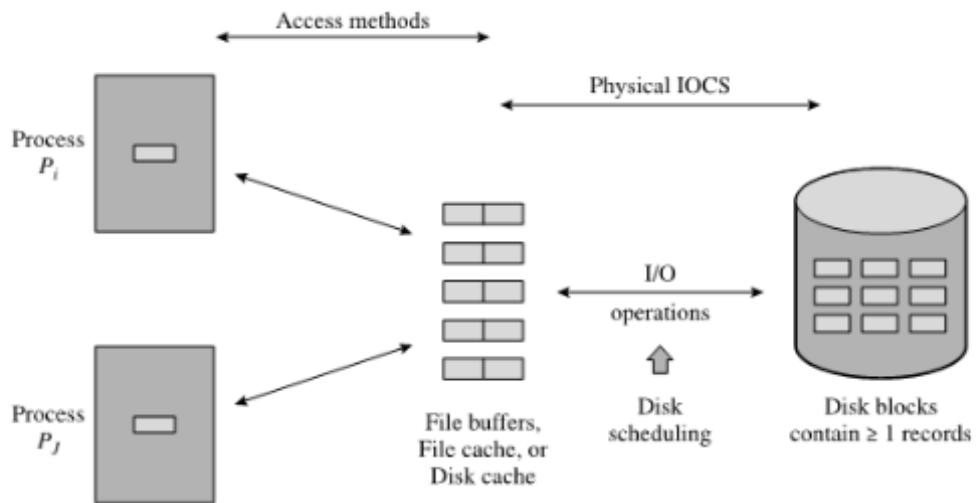


Figure 14.1 Implementation of file operations by the IOCS.

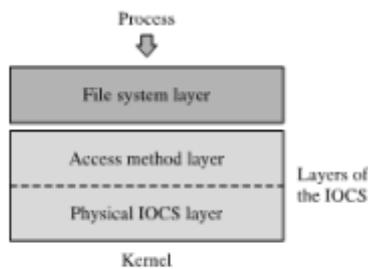


Figure 14.2 Layers of the file system and the IOCS.

Nella tabella sono riassunti i tre modi per effettuare le operazioni di I/O:

Modalità di I/O	Descrizione
I/O programmato	Il trasferimento dati tra la periferica di I/O e la memoria avviene attraverso la CPU. La CPU non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O.
Interrupt di I/O	La CPU è libera di eseguire altre istruzioni dopo aver eseguito l'istruzione di I/O. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte. Questa sequenza di operazioni viene ripetuta finché tutti i byte sono trasferiti.
I/O basato sul direct memory access (DMA)	Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus. La CPU non è coinvolta nel trasferimento dei dati. Il controller DMA genera un interrupt quando il trasferimento di tutti i byte è stato completato.

I dispositivi di I/O sono dotati, tipicamente, di una componente meccanica e di una componente elettronica ed è spesso possibile separare le due parti. La componente elettronica prende il nome di controllore del dispositivo. Il

compito del controllore del dispositivo è di convertire la sequenza di bit in un blocco di byte e di eseguire eventualmente la correzione degli errori. Ogni periferica ha un suo controller di dispositivo.

In questo tipo di trasferimento la CPU delega il DMA per effettuare il trasferimento dei dati. Inoltre, i dispositivi che effettuano questo tipo di trasferimento sono dei dispositivi collegati al DMA. Visto che questi dispositivi devono effettuare operazioni di lettura o scrittura a velocità molto alte, ci possono essere dei problemi per la contesa del buffer del DMA.

-Durante un'operazione di lettura (input), i dati vengono trasferiti dalla periferica ad un buffer del DMA, questi sono poi trasferiti dal buffer del DMA alla memoria al termine dell'operazione di I/O.

-Durante un'operazione di scrittura (output), i dati vengono prima trasferiti dalla memoria al buffer del DMA, poi da questo alla periferica.

durante tutte le operazione Input/Output la CPU non viene usufruita in quanto delega.

In generale:

- Quando è eseguita un'istruzione di I/O
 - Il controller del DMA passa i dettagli dei comandi di I/O al controller del dispositivo di I/O
 - Il dispositivo consegna i dati al controller di dispositivo
 - Il trasferimento dei dati da controller del dispositivo a memoria avviene come segue
 - Il controller del dispositivo invia un segnale DMA request quando è pronto al trasferimento
 - Il DMA, ricevuto il segnale, ottiene il controllo del bus e vi pone l'indirizzo di memoria che partecipa la trasferimento. Infine, invia un DMA ack al controller del dispositivo
 - Il controller del dispositivo trasferisce i dati verso o dalla memoria
 - Alla fine del trasferimento, il controller del DMA genera un interrupt di completamento I/O con codice uguale all'indirizzo del dispositivo
 - La routine di servizio degli interrupt analizza il codice per trovare quale dispositivo ha completato la sua operazione di I/O e intraprende le azioni appropriate

Esistono differenti tipologie di dispositivi di I/O che funzionano sulla base di vari principi fisici

- Generazione di segnali elettromeccanici
- Memorizzazione dati ottica o elettromagnetica

I dispositivi di I/O possono essere classificati sulla base dei seguenti criteri:

- **Scopo:** dispositivi di input, di stampa e di memorizzazione
- **Natura dell'accesso**
 - Sequenziale: tastiera, mouse, rete, nastro
 - Casuale: dischi
- **Modalità trasferimento dati:** caratteri o blocchi

L'informazione letta o scritta in un comando di I/O costituisce un record

Quindi le modalità di trasferimento dati possono essere diverse

Dipende dalla velocità di trasferimento

- Dispositivo di I/O lento (tastiera, mouse e stampante sono dispositivi a carattere)
 - Lavora nella modalità carattere: è trasferito un carattere per volta tra memoria e periferica
 - Contiene un buffer che memorizza il carattere
 - Il controller genera un interrupt in conseguenza di una lettura dal buffer (dispositivo di input) o di una scrittura nel buffer (dispositivo di output)
 - I controller possono essere connessi direttamente al bus
- Dispositivi di I/O veloci (nastri, dischi)
 - Lavora in modalità a blocco
 - Connesso ad un controller di DMA
 - Devono trasferire i dati a specifiche velocità
 - I dati sono trasferiti tra la periferica di I/O e un *buffer del DMA*

$$t_{io} = t_a + t_x$$

Il tempo di I/O read/write è dato da:

t_{io} (tempo di I/O) -> intervallo tra esecuzione istruzione inizio I/O e completamento operazione

t_a (tempo di accesso) -> intervallo tra un comando read o write e l'inizio del trasferimento

t_x (tempo di trasferimento) -> tempo necessario per trasferire dati da/verso una periferica durante un'operazione read o write (inizio trasferimento primo byte, fine trasferimento ultimo byte)

Ovviamente una memoria ad accesso casuale non si sa il tempo esatto.

Quando succede un I/O è auspicabile individuare e correggere errori di trasferimento dati:

Gli errori possono verificarsi durante la scrittura o la lettura dei dati o durante il trasferimento tra un dispositivo di I/O e la memoria

I dati trasmessi sono visti come flusso di bit

- Usati codici speciali per rappresentarli

Individuazione errori

- Si memorizzano informazioni ridondanti con i dati
 - Informazione di individuazione errori
 - Determinata dai dati con tecniche standard
 - Quando i dati sono letti da una periferica di I/O sono lette anche le informazioni di individuazione errori
 - Inoltre, tali informazioni sono calcolate nuovamente dai dati letti, usando la stessa tecnica
 - Si confrontano le info lette dal mezzo di I/O e quelle determinate dai dati letti
 - Un mismatch indica l'occorrenza di un errore in fase di memorizzazione

La memorizzazione e la lettura di informazioni ridondanti causa overhead

- La correzione degli errori comporta maggiore overhead rispetto alla loro all'individuazione

Approcci all'individuazione e correzione

- Bit di parità
 - Sono calcolati bit di parità dai bit di dati
 - Non distinguibili dai bit di dati se non all'algoritmo di individuazione/correzione
 - Individua errori su singolo bit
- Controllo di ridondanza ciclico (CRC)
 - Utilizza una funzione di hash per rilevare errori su più bit

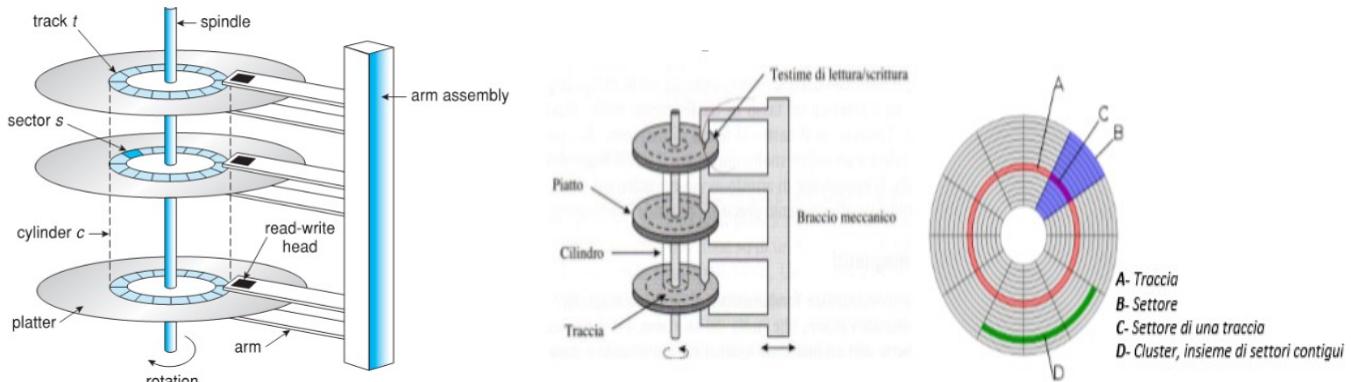
In ambo gli approcci si usa l'aritmetica modulo-2

- L'addizione è rappresentata come un OR-esclusivo

La correzione dell'errore è fatta in modo analogo

- Si usano algoritmi per determinare l'informazione di correzione
- Tali informazioni possono sia individuare un errore che suggerire come correggerlo
 - In ambo gli approcci si usa l'aritmetica modulo-2
 - L'addizione è rappresentata come un OR-esclusivo

Trattiamo ora la periferica di Hard Disk o Disco magnetico



I dischi magnetici sono costituiti da vari piatti, ognuno dei quali contiene tante tracce. Tutte le tracce uguali posizionate su piatti differenti formano un cilindro. L'elemento di memorizzazione di un disco magnetico è un oggetto circolare chiamato piatto, che ruota sul suo asse ed è ricoperto da un materiale magnetico. Ogni piatto memorizza i byte lungo delle tracce circolari sulla sua superficie. Ogni traccia è divisa in settori ed il numero di settori può variare a seconda delle tracce. I dischi moderni sono divisi in zone con più settori e zone con meno settori, ma grazie alla virtualizzazione, il SO vede tutte le zone come se avessero lo stesso numero di settori, dunque le richieste vengono fatte su quella base. La testina è capace di leggere e scrivere sulla superficie del piatto. Essa è capace di "muoversi" sul piatto grazie al braccio in modo tale da scorrere le tracce.

Riassumiamo le parti essenziali di un disco:

- Piatto: un disco rigido si compone di uno o più dischi paralleli, di cui ogni superficie, detta piatto, è identificata da un numero univoco ed è destinata alla memorizzazione dei dati
- Traccia: Ogni piatto si compone di numerosi anelli concentrici numerati, detti tracce, ciascuna identificata da un numero univoco
- Cilindro: L'insieme di tracce alla stessa distanza dal centro presenti su tutti i dischi è detto cilindro. Corrisponde a tutte le tracce aventi il medesimo numero, ma diverso piatto
- Settore: Ogni piatto è suddiviso in settori circolari, ovvero in spicchi radiali uguali ciascuno identificato da un numero univoco
- Blocco: L'insieme di settori posti nella stessa posizione in tutti i piatti
- Testina: Su ogni piatto è presente una testina per accedere in scrittura o in lettura ai dati memorizzati sul piatto; la posizione di tale testina è solidale con tutte le altre sugli altri piatti. In altre parole, se una testina è posizionata sopra una traccia, tutte le testine saranno posizioane nel cilindro a cui la traccia appartiene.

$$t_a = t_s + t_r$$

Il tempo di accesso al disco è dato da:

- t_s è il tempo di ricerca (o tempo di seek), cioè il tempo necessario per muovere la testina sulla traccia desiderata (valori soliti 5-15ms).
- t_r è la latenza rotazionale, cioè il tempo impiegato dal disco per ruotare portando il settore interessato sotto la testina richiesta; la latenza rotazionale media è il tempo impiegato per mezza rivoluzione del disco (valore soliti 3-4ms).

Nozione di cilindro

- Consiste di tracce posizionate in modo uguale su tutti i piatti di un disco
- Tutte le sue tracce possono essere accedute dalla stessa posizione della testina
- L'uso riduce il movimento della testina
- Pone dati adiacenti di un file su tracce dello stesso cilindro

Indirizzo di un record: (*numero cilindro, numero superficie, numero record*)

Per ottimizzare l'uso della superficie del disco le tracce sono organizzate in settori

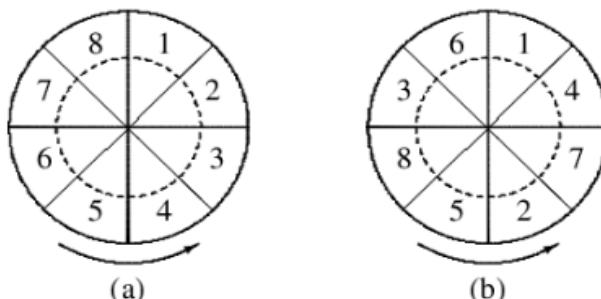
- Slot di dimensione standard in una traccia per un record
- Dimensione scelta per minimizzare lo spreco di capacità di memorizzazione

La suddivisione in settori può essere parte dello hw (hard sectoring) o implementata da software (soft sectoring)

Bisogna adottare delle tecniche sul disco in modo tale da rendere più efficiente il tempo di accesso ai dati, per questo utilizzo 3 principali tecniche: Alternanza dei settori e asimmetria di testina e di cilindro.

- Alternanza dei settori

- Tecnica: un piccolo numero di settori sono saltati mentre si memorizzano record adiacenti di un file
- In numero di settori saltati è chiamato fattore di alternanza (inf)



(a) nessuna alternanza; i record adiacenti in un file occupano settori adiacenti

(b) fattore di alternanza = 2; ci sono due settori tra record adiacenti

- Testina asimmetrica

- Il disco richiede del tempo per commutare dalla lettura dei dati di una traccia ai dati di un'altra traccia in un cilindro (tempo commutazione testina)
 - Alcuni settori (record/blocchi) passano sotto la testina durante questo tempo
 - Asimmetria testina: distribuisce i dati sulle tracce
 - Il primo settore di una nuova traccia deve passare sotto la testina solo dopo che la testina del disco è pronta per leggere

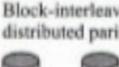
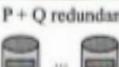
- Asimmetria cilindro

- Il disco ruota mentre le testine si spostano sulle tracce di un cilindro adiacente
 - Asimmetria cilindro: i dati sono resi asimmetrici come per l'asimmetria della testina

DISCHI RAID (redundant array of inexpensive disks) usa un insieme di dischi rigidi per condividere o replicare i dati e ha come obiettivi principali l'aumento delle prestazioni e una migliore affidabilità. Per fare ciò distribuisce i dati

coinvolti in un'operazione di I/O su diversi dischi ed esegue le operazioni di I/O su questi dischi in parallelo. Questa caratteristica può fornire accessi veloci o alti tassi di trasferimento, in base alla configurazione adottata. L'alta affidabilità è ottenuta memorizzando informazioni ridondanti. L'accesso alle informazioni ridondanti non necessita di tempo di I/O aggiuntivo in quanto si può accedere in parallelo sia ai dati che alle informazioni ridondanti. Da notare che la gestione di un RAID è più dispendiosa della gestione di un unico disco poiché vi sono più dischi da controllare. Esistono diverse configurazioni RAID che utilizzano differenti tecniche di ridondanza e organizzazione. Queste tecniche sono chiamate livelli RAID. Tutti i livelli RAID hanno però delle caratteristiche in comune:

- i vari hard disk configurati in RAID sono visti dal SO come un unico disco
- i dati sono divisi in strisce (stripes) distribuite sui vari dischi; una strip è memorizzata su più dischi nella stessa posizione in modo che, sincronizzati i dischi, è possibile leggere una striscia simultaneamente
- la capacità di ridondanza del disco è usata per memorizzare informazioni di parità che permettono il recupero dei dati in caso di guasti.

Livello	Tecnica	Descrizione
Livello 0	Disk striping 	I dati sono alternati su diversi dischi. Durante un'operazione di I/O, l'accesso ai dischi avviene in parallelo. Potenzialmente, utilizzando n dischi questa configurazione può garantire un incremento dell'ordine di n nel trasferimento dei dati.
Livello 1	Disk mirroring  Disco 1 Disco 2	Gli stessi dati sono memorizzati su due dischi. Durante la lettura dei dati, viene utilizzata la copia accessibile più velocemente. Una delle copie è accessibile anche dopo un guasto. Le operazioni di lettura possono essere eseguite in parallelo se non si verificano errori.
Livello 2	Codici di correzione degli errori  D D P P	Le informazioni di ridondanza sono memorizzate per rilevare e correggere gli errori. Ogni bit di dati o di informazione ridondante è memorizzato su un disco differente e letto o scritto in parallelo. Garantisce tassi di trasferimento elevati.
Livello 3	Bit-interleaved parity  D D P	Analogo al livello 2, fatta eccezione per il fatto che utilizza un singolo disco di parità per la correzione degli errori. Un errore che si verifica durante la lettura dei dati da un disco viene rilevato dal controller. Il bit di parità viene utilizzato per ripristinare i dati persi.
Livello 4	Block-interleaved parity  D D P	Scrive un blocco di dati, ovvero byte di dati consecutivi, in una strip e calcola una singola strip di parità per le strip di una stripe. Garantisce tassi di trasferimento elevati per operazioni di lettura molto grandi. Le operazioni di lettura piccole hanno bassi tassi di trasferimento; tuttavia, molte di queste operazioni possono essere eseguite in parallelo.
Livello 5	Block-interleaved distributed parity  D D P	Analogo al livello 4, fatta eccezione per il fatto che le informazioni sono distribuite su tutti i dischi. Consente di evitare che il disco di parità diventi un collo di bottiglia per l'I/O come nel livello 4. Inoltre garantisce migliori prestazioni in lettura rispetto al livello 4.
Livello 6	P + Q redundancy  D D P P	Analogo al livello 5, fatta eccezione per il fatto che utilizza due schemi di parità distribuita indipendenti. Supporta il ripristino dal guasto di due dischi.

Nota: D e P indicano dischi che contengono, rispettivamente, solo dati e solo informazioni di parità.  indica una strip. • Indica i bit di un byte memorizzati su dischi differenti e i loro bit di parità.  indica una strip contenente solo informazioni di parità.

Ritornando all'IOCS fisico sarebbe complicato aggiungere una nuova classe di dispositivi I/O. Nei moderni SO esiste il driver di dispositivo che aggiunto all'IOCS semplifica l'implementazione dei dispositivi. Quindi ogni dispositivo collegato al computer ha bisogno di codice specifico che controlla il dispositivo stesso. Tale codice è il driver del dispositivo. È detto driver l'insieme di procedure che permette ad un SO di utilizzare un dispositivo hardware senza sapere come esso funzioni.

Ogni dispositivo ha un suo driver anche se in realtà possono esistere unici driver per più di un dispositivo dello stesso tipo. Tale situazione resta comunque poco comune per la grande diversità che vi è tra un dispositivo e l'altro.

I driver sono caricati dalla procedura di boot del sistema in base alla classe dei dispositivi di I/O connessi al sistema. In alternativa, i driver possono essere caricati quando necessario durante il funzionamento del SO (ad esempio una penna USB); questa caratteristica è particolarmente utile per la funzione plug-and-play.

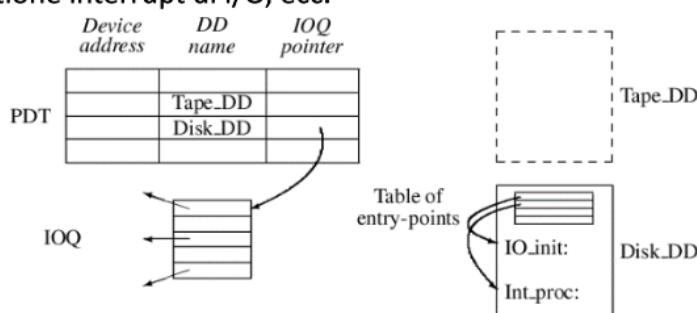
Soltanamente i driver di dispositivo sono posizionati al di sotto del resto del SO. In realtà sarebbe anche possibile trattare i driver di dispositivo a livello utente utilizzando chiamate di sistema per interfacciarsi con il kernel; questa situazione porterebbe anche il vantaggio di isolare il kernel dai dispositivi evitando crash causati dai driver che in genere interferiscono con il kernel. Le moderne architetture, in ogni caso, utilizzano driver che sono installati all'interno del SO.

Per connettere l'IOCS fisico e driver di dispositivi è utilizzato una struttura dati apposita PDT:

L'entrata della tabella dei dispositivi fisici di un dispositivo contiene il nome del driver

Un driver gestisce le operazioni di I/O su una specifica classe di dispositivi, inizia le operazioni di I/O e gestisce gli interrupt dal dispositivo nella classe

Il driver ha degli entry-point per funzionalità standard come avvio I/O, gestione interrupt di I/O, ecc.



SCHEDULING DEL DISCO: Una delle funzioni dello IOCS e dei driver dei dispositivi è quella di utilizzare lo scheduling del disco per eseguire le operazioni di I/O in un ordine tale da ridurre il movimento delle testine e il tempo medio di attesa delle operazioni di I/O. Quindi al variare del tipo di accesso, si influenzano diversamente le prestazioni del sistema. Resta il fatto che l'obiettivo comune di tutte le politiche di scheduling è quello di servire il massimo numero di richieste nell'unità di tempo. Esistono diversi algoritmi di scheduling:

- FIRST-COME, FIRST-SERVED (FCFS)

Seleziona l'operazione di I/O con tempo di richiesta inferiore, cioè le richieste vengono servite in modo sequenziale, in base all'ordine di arrivo. Questa, solitamente, è la politica peggiore: si ottengono buone prestazioni solo se molte richieste riguardano settori ravvicinati.

- SHORTEST SEEK TIME FIRST (SSTF)

Seleziona l'operazione di I/O con il più breve tempo di ricerca rispetto alla posizione corrente delle testine del disco, cioè seleziona la richiesta che richiede il minor movimento della testina dalla sua posizione corrente. Può essere soggetta a starvation: se il disco viene usato molto frequentemente il braccio tenderà a rimanere sempre nella parte vicina alla traccia richiesta correntemente, quindi le richieste relative alle tracce più esterne dovranno aspettare molto tempo prima di essere servite.

- SCAN

Questa politica è nota come algoritmo dell'ascensore. Muove le testine del disco da un estremo all'altro del piatto, servendo tutte le richieste che ci sono lungo la sua strada indipendentemente dal loro ordine di arrivo; una volta raggiunta l'ultima traccia, cioè l'altra estremità del piatto, la direzione di movimento viene invertita e le nuove richieste vengono servite nella scansione inversa. Una variante chiamata LOOK inverte la direzione delle testine del disco quando non ci sono più richieste di I/O nella direzione corrente.

- CIRCULAR SCAN O C-SCAN

Questa politica esegue la scansione come nello scheduling SCAN. Tuttavia, non esegue mai la scansione inversa, quindi la scansione viene ridotta ad un'unica direzione. Invece di procedere in direzione opposta, questa politica sposta le testine nella posizione di partenza sul piatto e inizia un'altra scansione. La variante circular look (che chiameremo C-LOOK) muove le testine solo finché ci sono richieste da eseguire prima di

iniziare una nuova scansione. Il vantaggio rispetto alla SCAN è che i tempi di attesa delle richieste sono più uniformi.

- **CIRCULAR SCAN O C-SCAN**

Questa politica esegue la scansione come nello scheduling SCAN. Tuttavia, non esegue mai la scansione inversa, quindi la scansione viene ridotta ad un'unica direzione. Invece di procedere in direzione opposta, questa politica sposta le testine nella posizione di partenza sul piatto e inizia un'altra scansione. La variante circular look (che chiameremo C-LOOK) muove le testine solo finché ci sono richieste da eseguire prima di iniziare una nuova scansione. Il vantaggio rispetto alla SCAN è che i tempi di attesa delle richieste sono più uniformi.

- **N-STEP-SCAN E F-SCAN**

Sia con SSTF che con SCAN o C-SCAN può succedere che il braccio non si muova da un determinato settore per un lungo periodo di tempo, se, ad esempio, uno o più processi hanno alte frequenze di accesso ad una particolare traccia. Per evitare questo fenomeno si possono usare due approcci: - n-step-scan: la coda viene suddivisa in sottocode sulle quali viene applicato lo SCAN fin quando non vengono svuotate. n è il numero di sottocode, dunque se $n=1$ allora l'algoritmo si trasforma in un semplice algoritmo di SCAN - f-scan: la coda viene suddivisa in due sottocode. La prima coda viene servita con lo SCAN finché non si svuota, le richieste che arrivano successivamente vengono accodate nella seconda coda che verrà servita solo dopo che la prima è stata svuotata.

Per migliorare le prestazioni dell'attività di elaborazione di un file all'interno di un processo, un metodo di accesso ricorre alle tecniche di buffering e blocking dei record.

La tecnica del buffering tenta di sovrapporre le attività di I/O e di CPU di un processo. Questo obiettivo è raggiunto in due modi:

- prefetching di un record di input in un buffer di I/O
- postwriting di un record di output da un buffer di I/O

Un buffer è un'area di memoria utilizzata per mantenere temporaneamente i dati coinvolti in un'operazione di I/O, cioè dei dati che, o devono essere letti da un dispositivo o devono essere scritti su di esso.

Per un file di input, la tecnica del buffering usa il prefetching: viene anticipata la lettura del prossimo dato per poi memorizzarlo nel buffer. In pratica, mentre il processo è impegnato nell'elaborazione di un dato, viene avviata un'operazione che legge il dato successivo e lo memorizza in un buffer prima che questo sia richiesto dal processo.

Questa tecnica è capace, quindi, di ridurre o addirittura eliminare il tempo di attesa, in quanto vengono sovrapposte le due operazioni (elaborazione del dato corrente, lettura del dato successivo).

Per un file di output, la tecnica del buffering usa il postwriting: il record che deve essere scritto viene semplicemente copiato in un buffer quando il processo esegue un'operazione di scrittura in modo che l'esecuzione del processo possa proseguire. L'effettiva scrittura è eseguita dal buffer in un momento successivo.

La tecnica del blocking riduce il numero di operazioni di I/O da eseguire, leggendo o scrivendo molti record in una singola operazione di I/O. In pratica legge più dati da un dispositivo in una singola operazione di I/O rispetto a quelli richiesti da un processo.

Quando più record vengono letti o scritti insieme, è necessario differenziare l'accesso e l'elaborazione dei dati e come vengono scritti sul dispositivo di I/O: - un record logico è l'unità di dati utilizzata in un processo per l'accesso e l'elaborazione - un record fisico, chiamato anche blocco, è l'unità dei dati utilizzata per il trasferimento da e per un dispositivo di I/O Il fattore di blocking di un file è il numero di record logici in un record fisico. Si dice che un file adotta il blocking dei record se il fattore di blocking è maggiore di 1.

Le azioni necessarie per l'estrazione di un record logico da un blocco, in modo che possa essere utilizzato in un processo, sono collettivamente chiamate azioni di deblocking.

Una tecnica generale per velocizzare l'accesso ai dati consiste nell'utilizzare una gerarchia di memoria composta da una parte della memoria e dai file memorizzati sul disco. Il caching è la tecnica di mantenere alcuni file in memoria, in modo che vi si possa accedere senza dover eseguire un'operazione di I/O. Il caching riduce il numero di operazioni di I/O necessarie per accedere ai dati memorizzati in un file, migliorando le prestazioni delle attività di elaborazione dei file nei processi e migliorando inoltre le prestazioni del sistema. L'IOCS fisico implementa una cache del disco per ridurre il numero di operazioni di I/O per accedere ai file memorizzati sul disco. Un metodo di accesso implementa una cache di file per ridurre il numero di operazioni di I/O eseguite durante l'elaborazione di un file.