Operating systems

▼ 0.0 - Info

Exam

- Prova scritta al pc
  - Parte teorica (1 ora)

    2 domande a risposta aperta + 1 esercizio di analisi di un programma.
  - Parte pratica (3 ore)

    1 esercizio di programmazione concorrente in Java + 1 esercizio di programmazione in linux (sviluppo di programma di sistema con system calls in C oppure realizzazione di un file comandi shell).

    È possibile consultare materiale cartaceo + librerie su USB create dallo studente da copiare entro i primi 5 minuti.
- Prova orale facoltativa

$$v_{scritta/pratica} = (v_{teoria} + v_{java} + v_{C/shell})/3 + bonus$$
$$v_{complessivo} = (v_{scritta/pratica} * 60 + v_{orale} * 40)/100$$

Prove vecchie su csunibo.

▼ 1.0 - Introduzione

▼ 1.1 - Introduzione ai sistemi operativi

Sistema operativo

Il sistema operativo è un programma (o un insieme di programmi) che agisce come intermediario tra l'utente e l'hardware del computer: fornisce una visione astratta e semplificata dell'HW, gestisce in modo efficace ed

efficiente le risorse del sistema di calcolo, mette a disposizione un ambiente di esecuzione e di sviluppo per i programmi degli utenti.

Il SO mappa le risorse HW in risorse logiche, accessibili attraverso interfacce ben definite (es. CPU → processi, memoria secondaria → file system, memoria centrale → memoria virtuale).

Cenni storici

Prima generazione (anni ‘50)

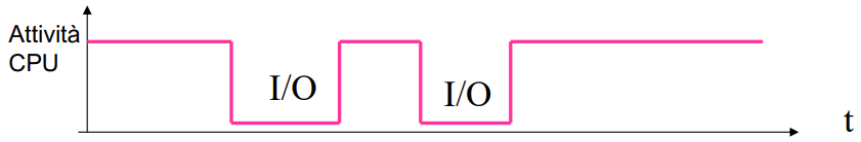
Architettura basata su valvole termoioniche, programmazione in linguaggio macchina, controllo del sistema manuale/meccanico, non è presente il sistema operativo.

Seconda generazione (anni ‘55 - ‘65)

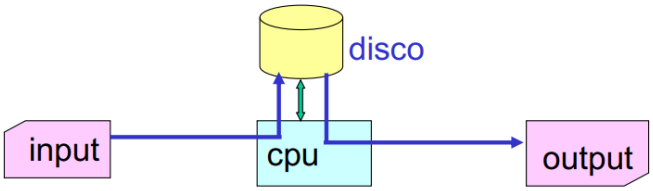
Sistemi batch semplici: architettura basata su transistor, linguaggio di alto livello (fortran), input mediante schede perforate, aggregazione di programmi in lotti (batch) con esigenze simili.

Batch: insieme di programmi (job) da eseguire in modo sequenziale.

Il compito del SO è quello di trasferire il controllo da un job (appena terminato) al prossimo da eseguire. Una caratteristica dei SO dell’epoca è quella di non avere interazione tra utente e job, e la presenza di una scarsa efficienza, infatti durante l’I/O del job corrente, la CPU rimane inattiva. Infatti, in memoria centrale, ad ogni istante, è caricato (al più) un solo job.



Una possibile soluzione a ciò è lo Spooling (Simultaneous Peripheral Operation On Line), ovvero la simultaneità di I/O e attività di CPU. Introduzione di un disco impiegato come buffer molto ampio dove: memorizzare in anticipo i programmi da eseguire, leggere in anticipo i dati, memorizzare temporaneamente i risultati (in attesa che il dispositivo di output sia pronto), caricare codice e dati del job successivo: → possibilità di sovrapporre I/O di un job con elaborazione di un altro job.



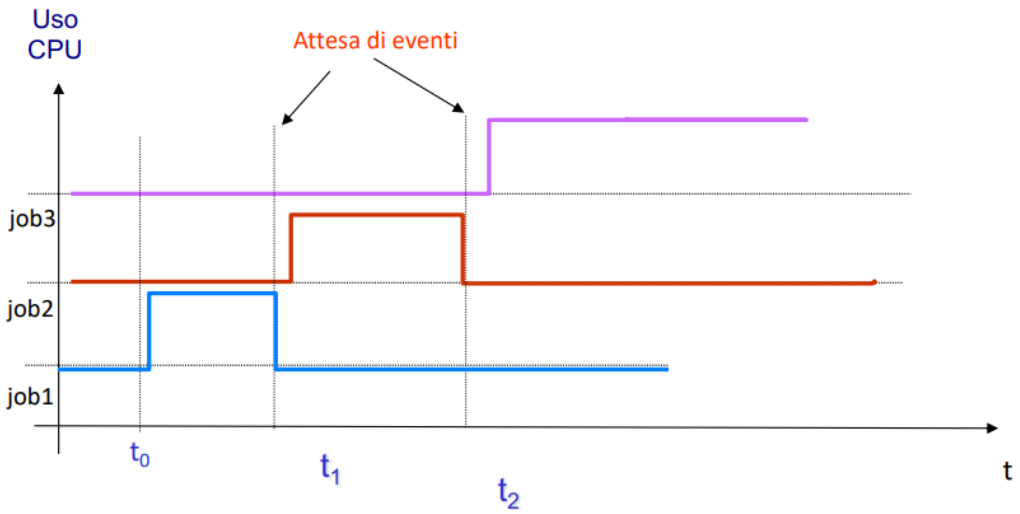
Sistemi batch multiprogrammati

Il SO è in grado di portare avanti l'esecuzione di più job contemporaneamente.

Ad ogni istante: un solo job utilizza la CPU, più job, appartenenti al pool selezionato e caricati in memoria centrale, attendono di acquisire la CPU.

Quando il job che sta utilizzando la CPU si sospende in attesa di un evento: SO decide a quale job assegnare la CPU (scheduling) ed effettua lo scambio (cambio di contesto o context switch). Quando avviene un cambio di contesto è necessario: salvare in memoria le informazioni (contenute nei registri di CPU) relative allo stato di esecuzione di J1; caricare nei registri di CPU le informazioni (precedentemente salvate in memoria) relative allo stato di J2, in modo che possa riprendere l'esecuzione dal punto in cui era stata interrotta.

SO effettua delle scelte tra tutti i job: quali job caricare in memoria centrale: scheduling dei job (long-term scheduling), a quale job assegnare la CPU: scheduling della CPU o (short-term scheduling).



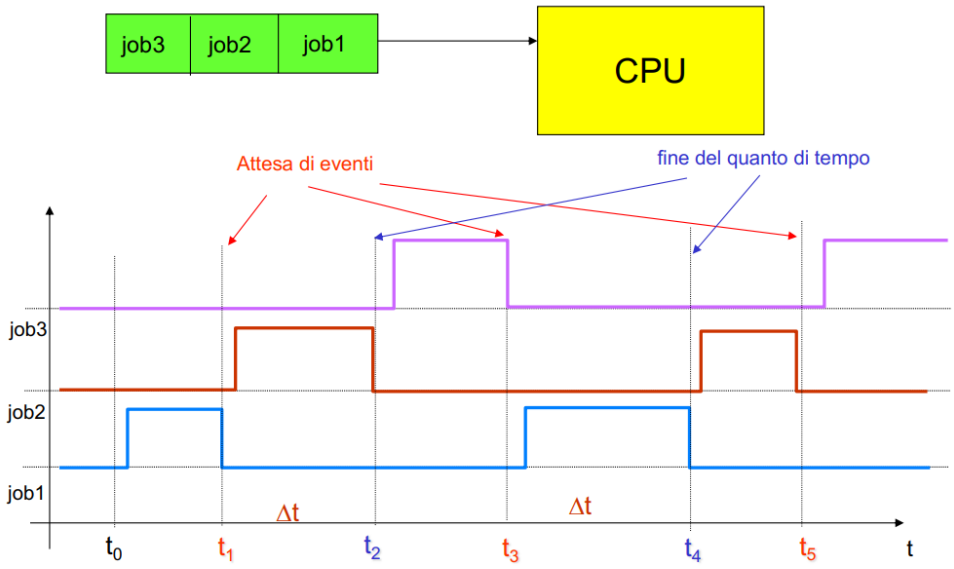
Sistemi time-sharing

Nascono dalla necessità di: interattività con l'utente e multi-utenza: più utenti interagiscono contemporaneamente con il SO.

Multiutenza: il sistema operativo presenta ad ogni utente una macchina "virtuale" completamente dedicata in termini di: utilizzo della CPU, utilizzo di altre risorse, ad es. file system.

Interattività: per garantire un'accettabile velocità di "reazione" alle richieste dei singoli utenti, il S.O. interrompe l'esecuzione di ogni job dopo un intervallo di tempo prefissato Dt (quanto di tempo, o time slice) assegnando la CPU a un altro job. La frequenza di commutazione della CPU è tale da fornire l'illusione ai vari utenti di una macchina completamente dedicata (macchina virtuale).

Il cambio di contesto (context switch) avviene quando il job corrente ha esaurito il quanto di tempo, oppure quando il job corrente si sospende in attesa di un evento.



Interruzioni

Le varie componenti (HW e SW) del sistema interagiscono con SO mediante interruzioni asincrone (interrupt).
Ad ogni interruzione è associata una routine di servizio (handler) per la gestione dell'evento.

- Interruzioni hardware: dispositivi inviano segnali a CPU per notificare particolari eventi al SO (es. completamento di un'operazione).
- Interruzioni software (trap): programmi in esecuzione possono generare interruzioni SW
    - esecuzione di operazioni non lecite (ad es. divisione per 0)
    - esecuzione di servizi al SO - system call

Alla ricezione di un'interruzione, il SO:

1. Interrompe la sua esecuzione ⇒ salvataggio dello stato in memoria
2. Attiva la routine di servizio all'interruzione (handler)
3. Ripristina lo stato salvato

Protezione delle risorse - Dual mode

Nei sistemi che prevedono multiprogrammazione e multiutenza sono necessari alcuni meccanismi HW (e non solo...) per esercitare protezione. Le risorse allocate a programmi/utenti devono essere protette nei confronti di accessi illeciti di altri programmi/utenti.

Per garantire protezione, molte architetture di CPU prevedono più modi di funzionamento (ring di protezione). Ad esempio, 2 modi (dual mode architecture): user mode e kernel mode (supervisor, monitor mode). Per fare ciò viene utilizzato un bit di modo (kernel: 0, user: 1) (più è alto il numero di modo, meno è il privilegio). Istruzioni privilegiate: istruzioni che possono essere eseguite soltanto se il sistema si trova in kernel mode.

System call

Solitamente il S.O. esegue in modo kernel, mentre ogni programma utente esegue in user mode: quando un programma utente tenta l'esecuzione di una istruzione privilegiata, l'hardware lo impedisce (può essere generato un trap). Se il programma necessita di operazioni privilegiate: chiamata a system call, possibile tramite l'utilizzo dell'Application Programming Interface (API).

La chiamata di system call provoca i seguenti effetti:

1. invio di un’interruzione software al S.O.
2. salvataggio dello stato (PC, registri, bit di modo, ...) del programma chiamante e trasferimento del controllo al SO
3. il SO esegue in modo kernel l’operazione richiesta tramite una specifica routine di gestione
4. al termine dell’operazione, il controllo ritorna al programma chiamante (ritorno al modo user): ripristino dello stato del programma chiamante

▼ 1.2 - Componenti di un sistema operativo

Componenti di un sistema operativo:

- gestione dei processi
- gestione della memoria centrale
- gestione di memoria secondaria e file system
- gestione dell’I/O
- protezione e sicurezza
- interfaccia utente/programmatore

Gestione dei processi

Il programma è un’entità passiva (un insieme di byte contenente le istruzioni che dovranno essere eseguite). Il processo è un’entità attiva: è l’unità di lavoro/esecuzione all’interno del sistema. Ogni attività all’interno del SO è rappresentata da un processo. È l’istanza di un programma in esecuzione.

Il SO ha un ruolo nella:

- creazione/terminazione dei processi
- sospensione/ripristino dei processi
- sincronizzazione/comunicazione dei processi
- gestione del blocco critico (deadlock) di processi

Gestione della memoria centrale

HW di sistema di elaborazione è equipaggiato con un unico spazio di memoria accessibile direttamente da CPU e dispositivi.

Il SO ha un ruolo nel:

- separare gli spazi di indirizzi associati ai processi
- allocare/deallocare memoria ai processi
- memoria virtuale - gestire spazi logici di indirizzi di dimensioni complessivamente superiori allo spazio fisico
- realizzare i collegamenti (binding) tra memoria logica e memoria fisica

Gestione dei dispositivi di I/O

Il SO effettua:

- interfaccia tra programmi e dispositivi
- per ogni dispositivo: device driver, che contiene la routine per l'interazione con un particolare dispositivo e conoscenza specifica sul dispositivo (ad es., routine di gestione delle interruzioni)

Gestione della memoria secondaria

Il SO effettua:

- allocazione/deallocazione di spazio
- gestione dello spazio libero
- scheduling delle operazioni sul disco

Gestione del file system

Il SO fornisce una visione logica uniforme della memoria secondaria (indipendente dal tipo e dal numero dei dispositivi) e deve:

- realizzare il concetto astratto di file, come unità di memorizzazione logica
- fornire una struttura astratta per l'organizzazione dei file (directory)
- permettere la creazione/manipolazione/cancellazione di file e directory
- gestire l'associazione tra file e dispositivi di memorizzazione secondaria

Protezione e sicurezza

Protezione: controllo dell'accesso alle risorse del sistema da parte di processi (e utenti) mediante autorizzazioni e modalità di accesso.

Sicurezza: se il sistema appartiene a una rete, la sicurezza misura l'affidabilità del sistema nei confronti di accessi indesiderati e potenzialmente malevoli (attacchi) dal mondo esterno.

Interfaccia utente

SO presenta un'interfaccia che consente l'interazione con l'utente:

- interprete comandi (shell): l'interazione avviene mediante una linea di comando.
- interfaccia grafica (graphical user interface, GUI): l'interazione avviene mediante interazione con elementi grafici sul desktop; di solito è organizzata a finestre.

▼ 1.3 - Struttura di un sistema operativo

Come sono organizzate le varie componenti all'interno del sistema operativo?

Varie soluzioni:

- struttura Monolitica
- struttura Modulare
- Microkernel
- Macchine virtuali

Struttura Monolitica

Il sistema operativo è costituito da un unico modulo eseguibile contenente un insieme di procedure, che realizzano le varie componenti: l'interazione tra le diverse componenti avviene mediante il meccanismo di chiamata a procedura.

Principale Vantaggio: basso costo di interazione tra le componenti.

Svantaggi: Il SO è un sistema complesso e un'impostazione monolitica non è ideale per garantire queste proprietà.

Struttura modulare

Le varie componenti del SO vengono organizzate in moduli caratterizzati da interfacce ben definite.

Esempio: Sistemi Stratificati (a livelli)

Il sistema operativo è costituito da livelli sovrapposti, ognuno dei quali realizza un insieme di funzionalità:

- ogni livello realizza un'insieme di funzionalità che vengono offerte al livello superiore mediante un'interfaccia
- ogni livello utilizza le funzionalità offerte dal livello sottostante, per realizzare altre funzionalità

Ad esempio: **THE (5 livelli)**

Vantaggi:

- Astrazione: ogni livello è un oggetto astratto, che fornisce ai livelli superiori una visione astratta del sistema (Macchina Virtuale), limitata alle astrazioni presentate nell'interfaccia.
- Modularità: le relazioni tra i livelli sono chiaramente esplicitate dalle interfacce → possibilità di sviluppo, verifica, modifica in modo indipendente dagli altri livelli.

Svantaggi:

- Organizzazione gerarchica tra le componenti: non sempre è possibile -> difficoltà di realizzazione.
- Scarsa efficienza: costo di attraversamento dei livelli

Microkernel

Il kernel è la parte del sistema operativo che esegue in modo kernel.

I SO a Microkernel presentano la struttura del nucleo ridotta a poche funzionalità di base, mentre il resto del SO è rappresentato da processi di utente.

Kernel ibridi

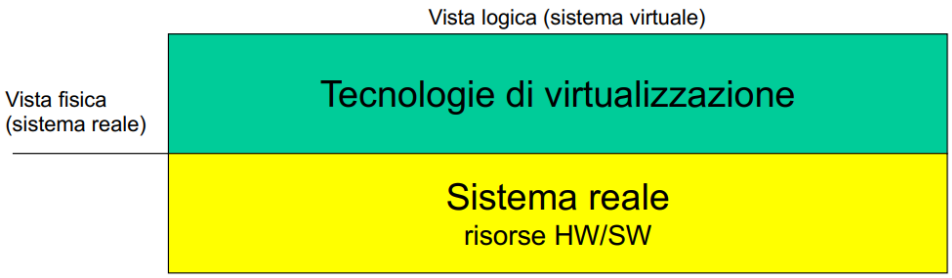
Microkernel che integrano a livello kernel funzionalità non essenziali.

Molti moderni SO implementano il kernel in maniera modulare, in cui ogni modulo può essere caricato nel kernel quando e ove necessario. Strutturazione simile ai livelli, ma con maggiore flessibilità.

Macchine virtuali

Le macchine virtuali (VMWare, VirtualBox, kvm, xen..) virtualizzano sia hardware che kernel del SO.

Dato un sistema caratterizzato da un insieme di risorse (hardware e software), virtualizzare il sistema significa presentare all'utilizzatore una visione delle risorse del sistema diversa da quella reale. Ciò si ottiene introducendo un livello di indirezione tra la vista logica e quella fisica delle risorse.



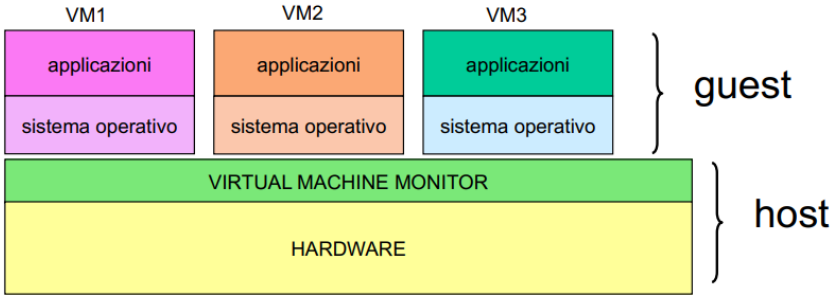
La macchina fisica viene trasformata, tramite un software che realizza la virtualizzazione: Virtual Machine Monitor (VMM), in N interfacce (macchine virtuali), ognuna delle quali e` una replica della macchina fisica.

Su ogni macchina virtuale è possibile installare ed eseguire un “sistema operativo” (eventualmente diverso da macchina a macchina).

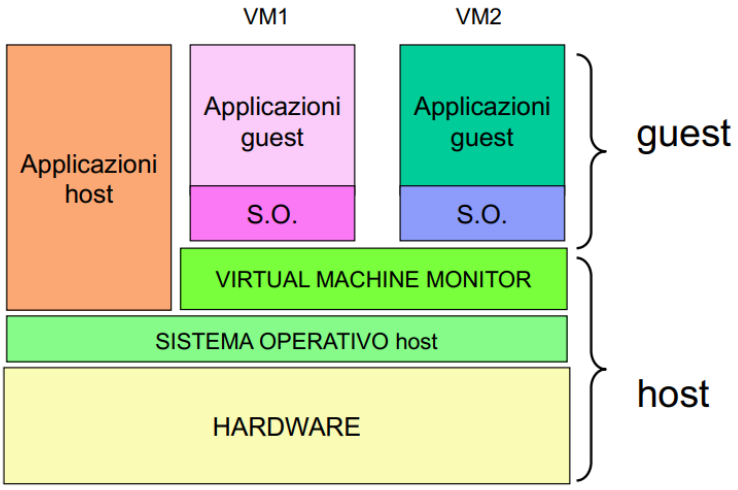
Host: piattaforma di base sulla quale si realizzano macchine virtuali. Comprende la macchina fisica, l’eventuale sistema operativo ed il VMM.

Guest: la macchina virtuale. Comprende applicazioni e sistema operativo

VMM di Sistema: le funzionalità di virtualizzazione vengono integrate in un sistema operativo leggero, costituendo un unico sistema posto direttamente sopra l’hardware dell’elaboratore.



VMM ospitato: il VMM viene installato come un’applicazione sopra un sistema operativo esistente, che opera nello spazio utente e accede l’hardware tramite le system call del S.O. su cui viene installato. Installazione più semplice ma performaces peggiori.



Vantaggi:

- Uso di piu` S.O. sulla stessa macchina fisica.
- Isolamento degli ambienti di esecuzione e conseguente possibilita di effettuare testing di applicazioni preservando l'integrita degli altri ambienti e del VMM.
- Possibilita di concentrare piu macchine (ad es. server) su un'unica architettura HW per un utilizzo efficiente dell'hardware (es. server farm).
- Gestione facilitata delle macchine: e` possibile effettuare in modo semplice:
  - la creazione di macchine virtuali
  - l'amministrazione di macchine virtuali
  - migrazione a caldo di macchine virtuali tra macchine fisiche (manutenzione hw senza interrompere i servizi)

▼ 1.3 - Linux

Introduzione

Linux è la versione libera realizzata da Linus Torvalds di Unix, un sistema operativo con kernel monolitico, multiprogrammato time-sharing e multi-utente.

In aggiunta alle caratteristiche di Unix Linux è multi-threaded e il kernel è monolitico con caricamento dinamico dei moduli.

Utenti e gruppi

Essendo Linux un sistema multiutente, questo implica la necessità di proteggere / nascondere informazione. In Linux viene quindi utilizzato un concetto di gruppo e ogni utente appartiene a un gruppo ma può far parte anche di altri a seconda delle esigenze.

Comandi della shell

I comandi eseguibili nella shell esistono nel file system come file binari, generalmente eseguibili da tutti gli utenti (direttorio /bin). È possibile realizzare nuovi comandi: programmazione in shell.

L’esecuzione dei comandi avviene in due modi:

- comandi builtin: definiti internamente ed eseguiti dentro il processo shell, come una chiamata a funzione.
- comandi external: la shell crea una shell figlio, dedicata all’esecuzione del comando.

type <comando> stampa il tipo (builtin/external).

File

Un file è l’unità logica di memorizzazione delle informazioni in memoria secondaria. Il file è una astrazione molto potente che consente di trattare allo stesso modo entità fisicamente diverse.

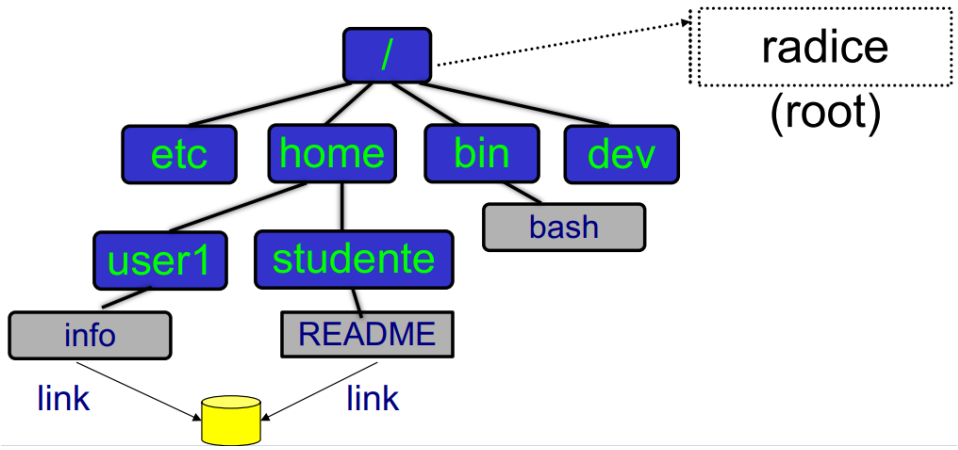
Tipi di file:

- ordinari: archivi di dati, testi, comandi, programmi sorgente, eseguibili
- directory: file gestiti direttamente solo dal S.O., che contengono riferimenti ad altri file
- speciali: dispositivi hardware, memoria centrale, hard disk
- FIFO (pipe): file per la comunicazione tra processi
- soft link: riferimenti (puntatori) ad altri file o direttori

Ogni file ha un nome, è possibile nominare un file con una qualsiasi sequenza di caratteri (max. 255), a eccezione di ‘.’ e ‘..’ (sono nomi che hanno un significato particolare).

File system

Il file system è organizzato come un grafo diretto aciclico (DAG).



▼ 2.0 - Processi

▼ 2.1 - Introduzione ai processi

Concetto di processo

Il processo è un programma in esecuzione.

Il processo è rappresentato da:

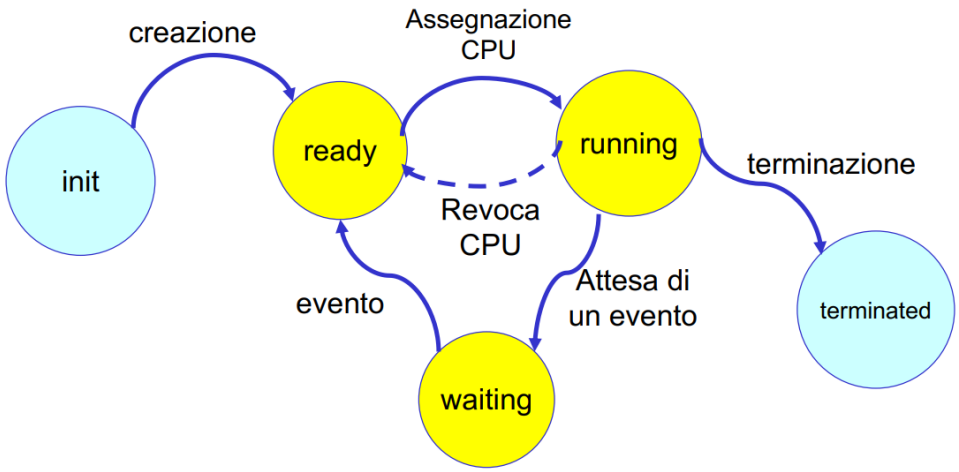
- codice del programma eseguito
- dati: variabili globali
- registri di CPU
- stack: parametri, variabili locali a funzioni/procedure
- risorse (file aperti, connessioni di rete utilizzate, altri dispositivi di I/O in uso, ecc.)

Stati di un processo

Un processo, durante la sua esistenza può trovarsi in vari stati:

- Init: stato transitorio durante il quale il processo viene caricato in memoria e SO inizializza i dati che lo rappresentano
- Ready: processo è pronto per acquisire la CPU
- Running: processo sta utilizzando la CPU
- Waiting: processo è sospeso in attesa di un evento

- Terminated: stato transitorio relativo alla fase di terminazione e deallocazione del processo dalla memoria



In un sistema monoprocessore e multiprogrammato:

- un solo processo (al massimo) si trova nello stato running
- più processi possono trovarsi contemporaneamente negli stati ready e waiting

Rappresentazione dei processi

Ad ogni processo viene associata una struttura dati (descrittore): Process Control Block (PCB).

Il PCB contiene tutte le informazioni relative al processo (Stato del processo, contenuto dei registri della CPU, informazioni di scheduling, ecc.)

Il sistema operativo gestisce i PCB di tutti i processi, organizzandoli in opportune strutture dati (ad esempio code) a seconda del loro stato.

Scheduling dei processi

È l'attività mediante la quale il SO effettua delle scelte tra i processi, riguardo a caricamento in memoria centrale e assegnazione della CPU.

In generale, il SO compie tre diverse attività di scheduling:

- scheduling a lungo termine

- scheduling a medio termine (o swapping)

  viene invocato a minore frequenza (sec-min) → può essere anche più lento

- scheduling a breve termine (o di CPU)

  viene invocato con alta frequenza (ms) → deve essere molto efficiente

Scheduler a lungo termine

Lo scheduler a lungo termine è quella componente del SO che seleziona i programmi da eseguire dalla memoria secondaria per caricarli in memoria centrale (creando i corrispondenti processi).

Nei sistemi time sharing non è presente, per via dell'interattività infatti di solito è l'utente che stabilisce direttamente quali/quanti processi caricare.

Scheduler a medio termine (o swapping)

Nei sistemi operativi multiprogrammati la quantità di memoria fisica può essere minore della somma delle dimensioni delle aree di memoria da allocare a ciascun processo.

Swapping: trasferimento temporaneo in memoria secondaria di processi (o di parti di processi), in modo da consentire il caricamento di altri processi.

Scheduler a breve termine

È quella parte del SO che si occupa della selezione dei processi a cui assegnare la CPU.

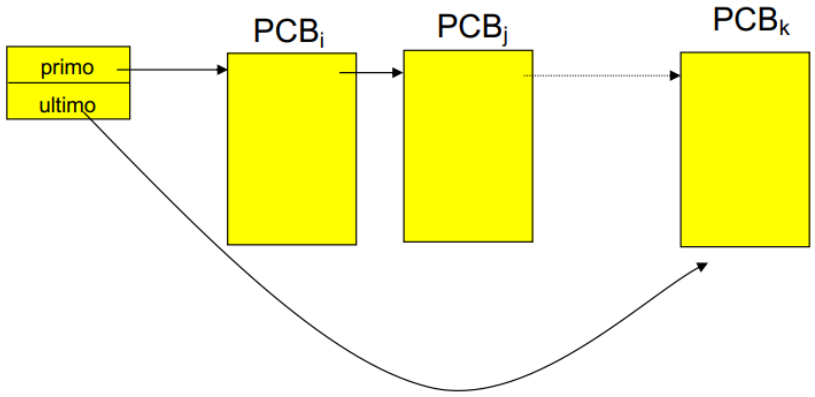
Nei sistemi multiprogrammati, ogni volta che il processo corrente lascia la CPU (attesa di un evento o, in sistemi time sharing, allo scadere del quanto di tempo) il SO decide a quale processo assegnare la CPU (scheduling di CPU) ed effettua il cambio di contesto (context switch).

Quando avviene un cambio di contesto tra un processo Pi ad un processo Pi+1 (ovvero, Pi cede l'uso della CPU a Pi+1):

- Salvataggio dello stato di Pi: SO copia PC, registri, ...del processo deschedulato Pi nel suo PCB.
- Ripristino dello stato di Pi+1: SO trasferisce i dati del processo Pi+1 dal suo PCB nei registri di CPU, che può così riprendere l'esecuzione.

Lo scheduler a breve termine gestisce:

- la coda dei processi pronti: contiene i PCB dei processi che si trovano in stato Ready
- code di waiting (una per ogni tipo di attesa: dispositivi I/O, timer, ...): ognuna di esse contiene i PCB dei processi waiting in attesa di un evento del tipo associato alla coda



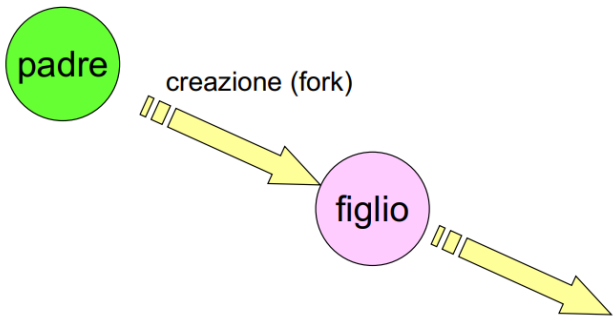
Operazioni sui processi

Ogni SO multiprogrammato prevede dei meccanismi per la gestione dei processi, i quali prevedono la creazione, terminazione e interazione tra processi (operazioni privilegiate, definizione di system call).

Creazione di processi

Un processo (padre) può richiedere la creazione di un nuovo processo (figlio). È possibile realizzare gerarchie di processi.

Ogni processo è figlio di un altro processo e può a sua volta essere padre di processi.



Terminazione di processi

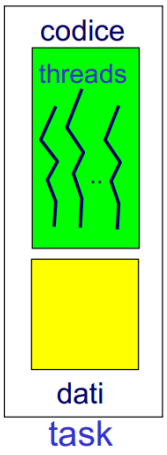
Se un processo termina:

- il padre può rilevare il suo stato di terminazione.
- possibili effetti sui figli:
  - tutti i figli terminano, oppure.
  - i figli continuano l'esecuzione e assumono come padre «adottivo» un altro processo (es: Unix).

Processi leggeri (thread)

Un thread (o processo leggero) è un'unità di esecuzione che condivide codice e dati con altri thread ad esso associati.

Task = insieme di thread che riferiscono lo stesso codice (text) e gli stessi dati.



Un processo pesante equivale invece a un task con un solo thread (single-threaded process).

Vantaggi:

- Condivisione di memoria: a differenza dei processi (pesanti), un thread può condividere variabili con altri thread (appartenenti allo stesso task).

- Minor costo di context switch: PCB di thread non contiene alcuna informazione relativa a codice e dati globali, quindi il cambio di contesto fra thread dello stesso task ha un costo notevolmente inferiore al caso dei processi “pesanti”.

Svantaggi:

- Minor protezione: thread appartenenti allo stesso task possono modificare dati gestiti da altri thread.

Molti SO offrono l’implementazione del concetto di thread, potendolo realizzare:

- A livello kernel:
  - SO gestisce direttamente i cambi di contesto tra thread dello stesso task (trasferimento di registri) e tra task
  - SO fornisce strumenti per la sincronizzazione per l’accesso di thread a variabili comuni
- A livello utente:
  - il passaggio da un thread al successivo (nello stesso task) non richiede interruzioni al SO (maggior rapidità)
  - SO vede processi pesanti: la sospensione di un thread causa la sospensione di tutti i thread del task
- Soluzioni miste (es. Solaris)
  - thread realizzati a entrambi i livelli

Interazione tra processi

I processi, pesanti o leggeri, possono interagire

Classificazione:

- processi indipendenti: due processi P1 e P2 sono indipendenti se l’esecuzione di P1 non è influenzata da P2, e viceversa
- processi interagenti: P1 e P2 sono interagenti se l’esecuzione di P1 è influenzata dall’esecuzione di P2, e/o viceversa

L’interazione può essere realizzata mediante:

- memoria condivisa (modello ad ambiente globale): SO consente ai processi (in questo caso, thread) di condividere variabili; l’interazione avviene tramite l’accesso a variabili condivise.
- scambio di messaggi (modello ad ambiente locale): i processi non condividono variabili (processi pesanti) e interagiscono mediante meccanismi di trasmissione/ ricezione di messaggi.

Tipi di interazione:

- Cooperazione: interazione prevedibile e desiderata, insita nella logica del programma concorrente. I processi cooperanti collaborano per il raggiungimento di un fine comune.

  Esempio: due thread (produttore e consumatore) accedono a un buffer condiviso di dimensione limitata. C’è necessità di sincronizzare i processi, infatti quando il buffer è vuoto il consumatore NON può prelevare messaggi, mentre quando il buffer è pieno il produttore NON può depositare messaggi. Per ovviare al problema vengono solitamente utilizzare variabili condivise (es. buffer_vuoto e buffer_pieno)

- Competizione: interazione prevedibile ma "non desiderata" tra processi che interagiscono per sincronizzarsi nell’accesso a risorse comuni

  Per risolvere alcuni problemi di competizione è possibile utilizzare la mutua esclusione, ovvero utilizzare istruzioni indivisibili. Data un'istruzione I(d), che opera su un dato d, essa e indivisibile (o atomica) , se, durante la sua esecuzione da parte di un processo P, il dato d non e accessibile ad altri processi.

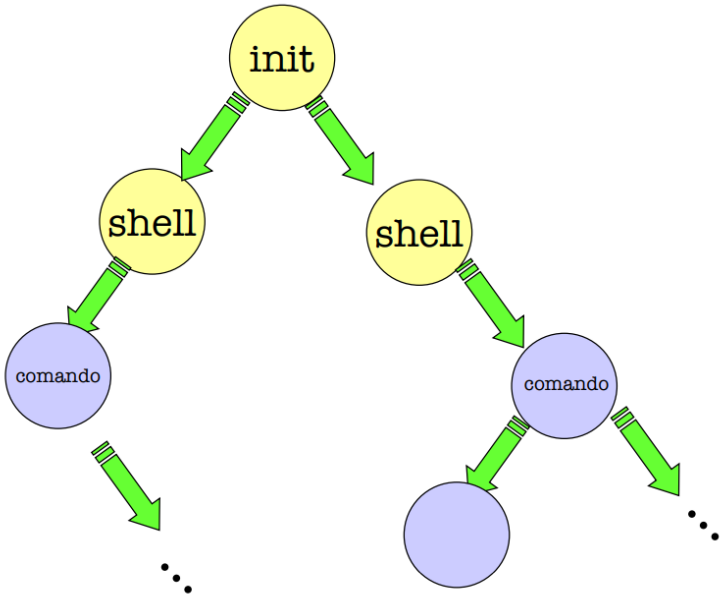
- Interferenza: interazione non prevista e non desiderata, potenzialmente deleteria tra processi

▼ 2.2 - Processi nel SO Unix

UNIX è un sistema operativo multiprogrammato a divisione di tempo, in cui l’unità di computazione è il processo.

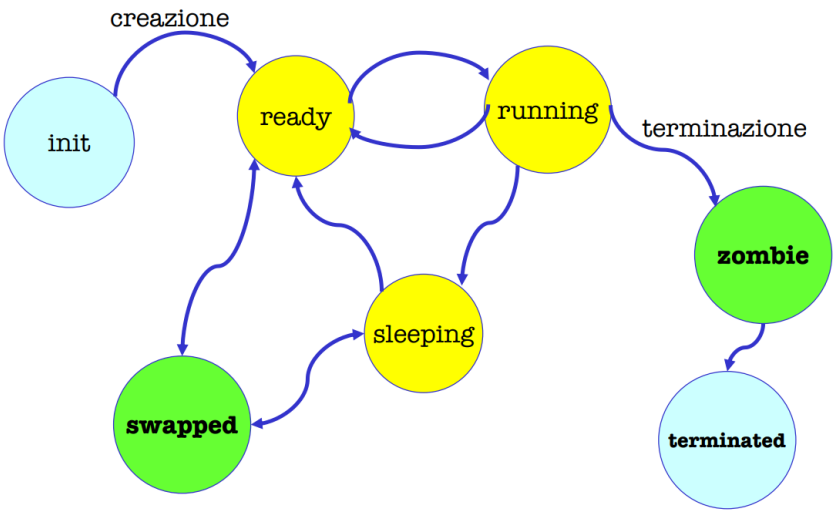
I processi sono di tipo pesante con codice rientrante, ovvero dati non condivisi e codice condivisibile con altri processi.

Ogni processo ha un proprio spazio di indirizzamento locale e non condiviso ad eccezione del codice, il quale può essere condiviso.



Gerarchie di processi UNIX.

Stati di un processo UNIX



- **Init**: caricamento in memoria del processo e inizializzazione delle strutture dati del SO
- **Ready**: processo pronto
- **Running**: il processo usa la CPU
- **Sleeping**: il processo è sospeso in attesa di un evento (v. waiting)
- **Terminated**: eliminazione del processo dalla memoria e dal SO
- **Zombie**: il processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione
- **Swapped**: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria

Lo scheduler a medio termine (swapper) gestisce i trasferimenti dei processi

- da memoria centrale a secondaria (dispositivo di swap): swap out.

  si applica preferibilmente ai processi bloccati (sleeping), prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo (preferibilmente i processi più lunghi)
- da memoria secondaria a centrale: swap in

  si applica preferibilmente ai processi più corti

Struttura dei processi

Il SO gestisce una struttura dati globale in cui sono contenuti i puntatori ai codici utilizzati, (eventualmente condivisi) dai processi: text table.
L'elemento della text table si chiama text structure e contiene:

- puntatore al codice (se il processo è swapped, riferimento a memoria secondaria)
- numero dei processi che lo condividono

Il process control block del processo in UNIX è rappresentato da 2 strutture dati distinte:

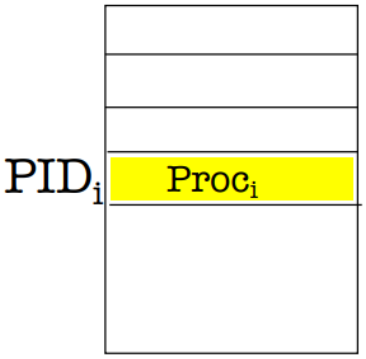
- Process structure: informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato in cui si trova il processo)

  Ad ogni process structure è associato un valore intero non negativo che rappresenta l'identificatore unico del processo (Process IDentifier, PID)

  La process structure contiene le seguenti informazioni:

  - Lo stato del processo
  - puntatori alle aree dati e stack associati al processo
  - riferimento indiretto al codice, ovvero il riferimento all'elemento della text table (text structure) associato al codice del processo
  - informazioni di scheduling (es: priorità, tempo di CPU, ...)
  - riferimento al processo padre ovvero il PID del padre;
  - informazioni relative alla gestione di segnali (es. segnali inviati ma non ancora gestiti, v. segnali..)
  - Puntatore al processo successivo nella coda di processi (ad esempio, ready queue) alla quale il processo eventualmente appartiene
  - puntatore alla User Structure

  Tutte le Process structure sono organizzate in un vettore gestito dal sistema operativo: Process table.



- User structure: informazioni necessarie solo se il processo è residente in memoria centrale

  Contiene le seguenti informazioni:

  - copia dei registri di CPU (Program Counter, ecc.)
  - informazioni sulle risorse allocate (file aperti)
  - informazioni sulla gestione di segnali (puntatori a handler, ecc., v. segnali..)
  - ambiente del processo: direttorio corrente, utente, gruppo, argc/argv, path, ...

Immagine di un processo

L'immagine di un processo è l'insieme delle aree di memoria e strutture dati associate al processo.

Non tutta l'immagine è accessibile in modo user, in quanto esiste una parte di kernel e una parte di utente.

Inoltre non tutta l'immagine può essere trasferita in memoria con lo swapping, in quanto ne esiste una parte swappable e una non swappable.

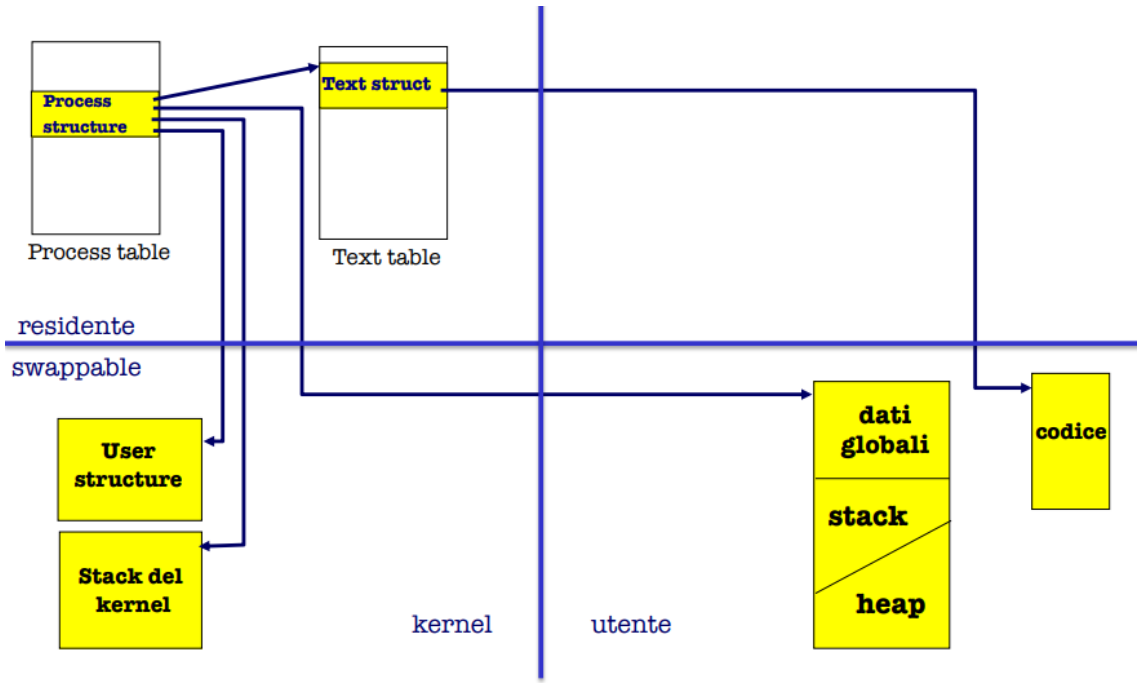


Immagine di un processo UNIX.

Gestione dei processi

Esistono diverse system call utili per la gestione dei processi:

- creazione di processi: fork()
- terminazione: exit()
- sospensione in attesa della terminazione di figli: wait()
- sostituzione di codice e dati: exec()

Creazione di processi: fork()

```
int fork(void);
```

La funzione fork() consente a un processo di generare un processo figlio:

- padre e figlio condividono lo STESSO codice
- il figlio EREDITA una copia dei dati (di utente e di kernel) del padre

La funzione non richiede parametri e restituisce un intero che per il processo creato (figlio) vale 0, per il processo padre è un valore positivo che rappresenta il PID del processo figlio ed è un valore negativo in caso di errore (la creazione non è andata a buon fine).

Effetti della fork:

- Allocazione di una nuova process structure nella process table associata al processo figlio e sua inizializzazione
- Allocazione di una nuova user structure nella quale viene copiata la user structure del padre
- Allocazione dei segmenti di dati e stack del figlio nei quali vengono copiati dati e stack del padre
- Aggiornamento del riferimento text al codice eseguito (condiviso col padre): incremento del contatore dei processi, ...

Dopo una fork c'è concorrenza, in quanto padre e figlio procedono contemporaneamente. Il figlio nasce con lo stesso program counter del padre: la prima istruzione eseguita dal figlio è quella che segue

immediatamente fork(). Per questo solitamente viene utilizzata la seguente struttura di codice:

```
if (fork()==0) {
    /* codice eseguito dal figlio */
} else {
    /* codice eseguito dal padre */
}
```

Terminazione di processi: exit()

```
void exit(int status);
```

Un processo può terminare involontariamente, con tentativi di azioni illegali o interruzione mediante segnale, oppure volontariamente, tramite l'esecuzione dell'ultima istruzione oppure la chiamata alla funzione exit()

La funzione exit() prevede un parametro (status) mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione (ad esempio esito dell'esecuzione).

Effetti di una exit():

- chiusura dei file aperti non condivisi
- terminazione del processo: se il processo che termina ha figli in esecuzione, il processo init adotta i figli dopo la terminazione del padre (nella process structure di ogni figlio al pid del processo padre viene assegnato il valore 1), se il processo termina prima che il padre ne rilevi lo stato di terminazione con la system call wait(), i processi figli passano nello stato zombie

Attesa della terminazione dei figli: wait()

```
int wait(int *status);
```

Lo stato di terminazione può essere rilevato dal processo padre, mediante la system call wait().

Il parametro status è il puntatore alla variabile in cui viene memorizzato lo stato di terminazione del figlio, mentre il risultato prodotto dalla wait() è pid del processo terminato, oppure un codice di errore (<0).

Gli effetti della system call wait() sono i seguenti:

- se tutti i figli non sono ancora terminati, il processo si sospende in attesa della terminazione del primo di essi
- se almeno un figlio F è già terminato ed il suo stato non è stato ancora rilevato (cioè F è in stato zombie), wait() ritorna immediatamente con il suo stato di terminazione (nella variabile status)
- se non esiste neanche un figlio, wait() NON è sospensiva e ritorna un codice di errore (valore ritornato < 0)

Lo standard POSIX.1 prevede delle macro (definite nell'header file <sys/wait.h> per l'analisi dello stato di terminazione. In particolare:

- WIFEXITED(status): restituisce vero se il processo figlio è terminato volontariamente. In questo caso la macro WEXITSTATUS(status) restituisce lo stato di terminazione
- WIFSIGNALED(status): restituisce vero se il processo figlio è terminato involontariamente. In questo caso la macro WTERMSIG(status) restituisce il numero del segnale che ha causato la terminazione

Gestione degli errori: errno e perror()

In caso di fallimento, ogni system call ritorna un valore negativo (tipicamente, -1), in aggiunta, UNIX prevede la variabile globale di sistema errno, alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita. Per interpretarne il valore è possibile usare la funzione perror("stringa"), la quale stampa "stringa" seguita dalla descrizione testuale del codice di errore contenuto in errno.

Sostituzione di codice e dati: exec()

Mediante fork() i processi padre e figlio condividono il codice e lavorano su aree dati duplicate. In UNIX è possibile differenziare il codice dei due processi mediante una system call della famiglia exec: execl(), execle(), execlp(), execv(), execve(), execvp()...

L’effetto principale di system call famiglia exec è quello di sostituire codice ed eventuali argomenti di invocazione del processo che chiama la system call, con codice e argomenti di un programma specificato come parametro della system call (exec non crea un nuovo processo: il processo è lo stesso (stesso pid) prima e dopo exec).

- execl()

  ```
  int execl(char *pathname, char *arg0, ..., char *argN,
  ```

  - pathname: nome (assoluto o relativo) dell’eseguibile da caricare
  - arg0: nome del programma (argv[0])
  - arg1, ..., argN: argomenti da passare al programma
  - (char *)0: puntatore nullo che termina la lista

  In assenza di errori, execl è una chiamata senza ritorno. Se la funzione execl() ritorna (ovvero restituisce un valore al chiamante), significa che la chiamata è fallita, pertanto il processo continua ad eseguire il codice iniziale. In questo caso, il valore restituito è -1 e la variabile di sistema errno, assumerà il valore associato al particolare errore.

- le altre varianti di exec differiscono per la modalità di passaggio al programma degli argomenti

Effetti dell’exec() sul processo:

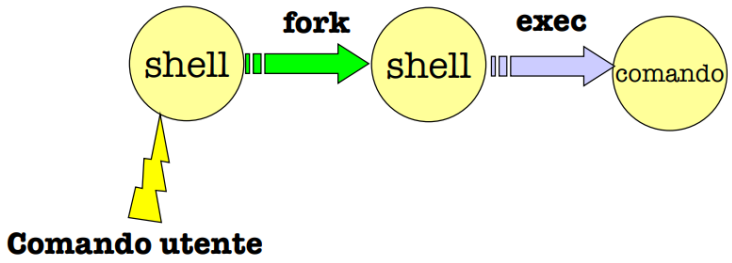
- mantiene la stessa process structure (salvo le informazioni relative al codice), quindi stesso pid e stesso pid del padre.
- ha codice, dati globali, stack e heap nuovi
- riferisce un nuovo text
- mantiene user structure (a parte PC e informazioni legate al codice) e stack del kernel, dunque mantiene le stesse risorse (es: file aperti) e lo stesso environment (a meno che non sia execle o execve)

Interazione dell’utente tramite shell

Ogni utente può interagire con lo shell mediante la specifica di comandi, i quali sono presenti nel file system come file eseguibili (direttorio /bin).

Per ogni comando, shell genera un processo figlio dedicato all'esecuzione del comando, e c'è la possibilità di due diversi comportamenti:

- il padre si pone in attesa della terminazione del figlio (esecuzione in foreground);
  es: ls -l pippo
- il padre continua l'esecuzione concorrentemente con il figlio (esecuzione in background): es. ls -l pippo &



▼ 2.3 - Interazione tra processi

Tipologie di interazioni tra processi

Classificazione dei processi:

- processi interagenti: l'esecuzione di un processo è influenzata dall'esecuzione dell'altro processo o viceversa.
  - cooperanti: i processi interagiscono volontariamente per raggiungere obiettivi comuni (fanno parte della stessa applicazione).
  - in competizione: i processi, in generale, non fanno della stessa applicazione, ma interagiscono indirettamente, per l'acquisizione di risorse comuni.
- processi indipendenti: l'esecuzione di ognuno non è in alcun modo influenzata dall'altro.

L'interazione tra processi può avvenire mediante due meccanismi:

- Comunicazione: scambio di informazioni tra i processi interagenti.

Nel modello ad ambiente locale (processi pesanti), in cui non c'è condivisione di variabili, la comunicazione avviene attraverso scambio di messaggi.

Nel modello ad ambiente globale (thread), in cui c'è possibilità di condividere variabili, la comunicazione avviene attraverso variabili condivise + strumenti di sincronizzazione (lock, semafori ecc.).

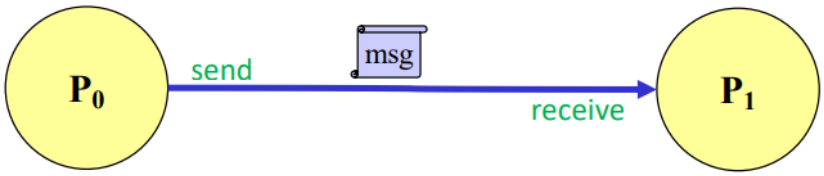
- Sincronizzazione: imposizione di vincoli temporali sull'esecuzione dei processi.

  Nel modello ad ambiente locale avviene attraverso scambio di eventi, in quanto in questo modello non c'è possibilità di condividere memoria, dunque gli accessi alle risorse condivise vengono controllati e coordinati dal sistema operativo.

  Nel modello ad ambiente globale avviene attraverso variabili condivise + strumenti di sincronizzazione.

Comunicazione con scambio di messaggi (ambiente locale)

Lo scambio di messaggi avviene mediante un canale di comunicazione e 2 operazioni necessarie: send e receive.



Naming

Il naming indice il modo in cui viene specificata la destinazione di un messaggio:

- Comunicazione diretta: al messaggio viene associato l'identificatore del processo destinatario (naming esplicito)

```
send(Proc, msg)
```

Il canale viene creato automaticamente tra i due processi ed è di tipo punto a punto e bidirezionale.

- Comunicazione indiretta: il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio:

```
send(Mailbox, msg)
```

  La mailbox è un canale utilizzabile da più processi che funge da contenitore di messaggi e ne esistono due tipologie:

  - mailbox di sistema: molti-a-molti
  - mailbox del processo destinatario: molti-a-uno

Buffering

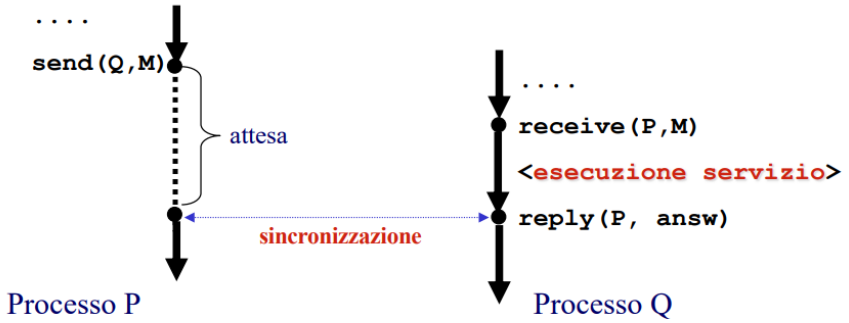
Ogni canale di comunicazione è caratterizzato da una capacità (>=0): numero dei messaggi che è in grado di contenere contemporaneamente. La gestione del canale viene solitamente fatto tramite politica FIFO, ovvero tramite una coda, e i canali si possono suddividere in base alla loro capacità:

- Capacità nulla: non vi è accodamento perchè il canale non è in grado di gestire messaggi in attesa. Processo mittente e destinatario devono sincronizzarsi all'atto di spedire (send) / ricevere (receive) il messaggio: comunicazione sincrona. Send e receive possono essere sospensive.

- Capacità non nulla (limitata): esiste un limite N alla dimensione della coda. Se la coda è piena la send è sospensiva, se la coda è vuota la

receive può essere sospensiva.

Oltre al send con capacità nulla, esiste un altro send sincrono, chiamato send con sincronizzazione estesa, in cui il mittente si sospende fino a che il destinatario non restituisce una risposta (reply) al messaggio inviato.



Sincronizzazione

La sincronizzazione permette di imporre vincoli sulle operazioni dei processi interagenti.

Nella cooperazione:

- Per imporre un particolare ordine cronologico alle azioni eseguite dai processi interagenti.
- Per notificare l'avvenimento di un evento

Nella competizione:

- Per garantire la mutua esclusione dei processi nell'accesso alla risorsa condivisa.

  Se la risorsa R è già «occupata» da uno dei due thread (che sta eseguendo un'operazione su R), l'altro thread sia costretto ad attendere. Quando la risorsa viene liberata, il thread eventualmente in attesa sia riattivato.
- Per realizzare politiche di accesso alle risorse condivise.

▼ 2.4 - Segnali Unix

I segnali Unix

Unix adotta il modello ad ambiente locale: la sincronizzazione può realizzarsi mediante segnali.

Segnale: è un'interruzione software, che notifica un evento in modo asincrono al processo che la riceve.

Per ogni versione di Unix esistono vari tipi di segnale (in Linux, 32 segnali), ognuno identificato da un intero. Ogni segnale, è associato a un particolare evento e prevede una specifica azione di default. È possibile riferire i segnali con identificatori simbolici (SIGxxx): SIGKILL, SIGSTOP, SIGUSR1, etc. L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di Unix) è specificata nell'header file <signal.h>.

Un segnale può essere inviato ad un processo dal kernel o da un altro processo, e quando un processo riceve un segnale, può comportarsi in tre modi diversi:

- gestire il segnale con una funzione handler definita dal programmatore
- eseguire un'azione predefinita dal S.O. (azione di default)
- ignorare il segnale (nessuna reazione)

Casi particolari: esistono 2 segnali che non possono essere gestiti esplicitamente dai processi: SIGKILL e SIGSTOP non sono nè intercettabili, nè ignorabili. Qualunque processo, alla ricezione di SIGKILL o SIGSTOP esegue sempre l'azione di default.

Se, durante l'esecuzione di un handler, arriva un secondo segnale uguale a quello che ha causato la sua esecuzione, il segnale viene accodato e gestito una volta terminato il primo handler. Segnali che arrivano mentre c'è un segnale uguale accodato vengono "accorpati" in uno: i.e. solo un segnale verrà consegnato al processo (e.g. molte chiamate a kill() eseguite in tempi ravvicinati).

Gestione del segnale

Ogni processo può gestire esplicitamente un segnale utilizzando la system call signal:

```
void (* signal(int sig, void (*func)())) (int);
```

- sig è l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro func è un puntatore a una funzione che indica l'azione da associare al segnale; in particolare func può:
  - puntare alla routine di gestione dell'interruzione (handler)

    l'handler prevede sempre un parametro formale di tipo int che rappresenta il numero del segnale effettivamente ricevuto, e non restituisce alcun risultato.
  - valere SIG_IGN (nel caso di segnale ignorato)
  - valere SIG_DFL (nel caso di azione di default)
- ritorna un puntatore a funzione:
  - al precedente gestore del segnale in caso di successo
  - la costante SIG_ERR, nel caso di errore

Gestione dopo fork e exec

Le associazioni segnali-azioni vengono registrate nella User Structure del processo. Sappiamo che una fork copia la User Structure del padre nella User Structure del figlio padre e figlio condividono lo stesso codice quindi il figlio eredita dal padre le informazioni relative alla gestione dei segnali.

Sappiamo che una exec sostituisce codice e dati del processo che la chiama in seguito a una chiamata a exec il contenuto della User Structure viene mantenuto, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo l'exec non sono più visibili!) quindi dopo un'exec, un processo ignora gli stessi segnali ignorati prima di exec, i segnali a default rimangono a default ma i segnali che prima erano gestiti, vengono riportati a default.

Invio del segnale

Il comando di shell kill serve ad inviare segnali ai processi.

```
kill -signal_number pid
```

I processi possono inviare segnali ad altri processi con la system call kill:

```
int kill(int pid, int sig);
```

- pid specifica il destinatario del segnale.
- sig è l'intero (o il nome simbolico) che individua il segnale da gestire
  - pid > 0: l'intero è il pid dell'unico processo destinatario
  - pid = 0: il segnale è spedito a tutti i processi appartenenti al gruppo del mittente
  - pid < -1: il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid
  - pid = -1: vari comportamenti possibili (Posix non specifica)

Altre system call

- Sleep

  ```
  unsigned int sleep(unsigned int N)
  ```

  provoca la sospensione del processo per N secondi (al massimo), se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato prematuramente.

  Ritorna:

  - 0, se la sospensione non è stata interrotta da segnali
  - se il risveglio è stato causato da un segnale al tempo Ns, sleep restituisce il numero di secondi non utilizzati dell'intervallo di sospensione (N-Ns)

- Alarm

  ```
  unsigned int alarm(unsigned int N)
  ```

  Imposta un timer che dopo N secondi invierà al processo il segnale SIGALRM

  Ritorna:

    - 0, se non vi erano time-out impostati in precedenza
    - il numero di secondi mancante allo scadere del timeout precedente
- Pause

```
int pause(void)
```

Sospende il processo fino alla ricezione di un qualunque segnale.

▼ 3.0 - File System

▼ 3.1 - Introduzione al file system

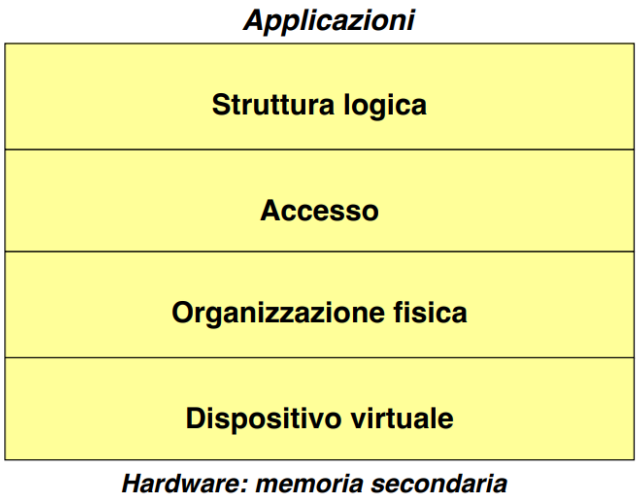
E' quella componente del SO che fornisce i meccanismi di accesso e memorizzazione delle informazioni allocate in memoria di massa.

Realizza i concetti astratti

- di file: unità logica di memorizzazione
- di direttorio: insieme di file (e direttori)
- di partizione: insieme di file associato ad un particolare dispositivo fisico (o porzione di esso)

Organizzazione del file system

La struttura di un file system può essere rappresentata da un insieme di componenti organizzate in vari livelli:



- Struttura logica: presenta alle applicazioni una visione astratta delle informazioni memorizzate, basata su file, directory, partizioni, ecc.. Realizza le operazioni di gestione di file e directory: copia, cancellazione, spostamento, ecc.
- Accesso: definisce e realizza i meccanismi per accedere al contenuto dei file; in particolare:
  - Definisce l'unità di trasferimento da/verso file: record logico
  - Realizza i metodi di accesso (sequenziale, casuale, ad indice)
  - Realizza i meccanismi di protezione
- Organizzazione fisica: rappresentazione di file e directory sul dispositivo:
  - Allocazione dei file sul dispositivo (unità di memorizzazione = blocco): mapping di record logici su blocchi. Vari metodi di allocazione.
  - Rappresentazione della struttura logica sul dispositivo.
- Dispositivo Virtuale: presenta una vista astratta del dispositivo, che appare come una sequenza di blocchi, ognuno di dimensione data costante.

Struttura logica

È un insieme di informazioni; ad es.:

- programmi
- dati (in rappresentazione binaria)
- dati (in rappresentazione testuale)

Ogni file è individuato da (almeno) un nome simbolico mediante il quale può essere riferito e un insieme di attributi. Solitamente:

- tipo: stabilisce l'appartenenza a una classe (eseguibili, testo, musica, non modificabili, ...)
- indirizzo: puntatore/i a memoria secondaria

- dimensione: numero di byte contenuti nel file
- data e ora (di creazione e/o di modifica)
- In SO multiutente anche:
  - utente proprietario
  - protezione: diritti di accesso al file per gli utenti del sistema

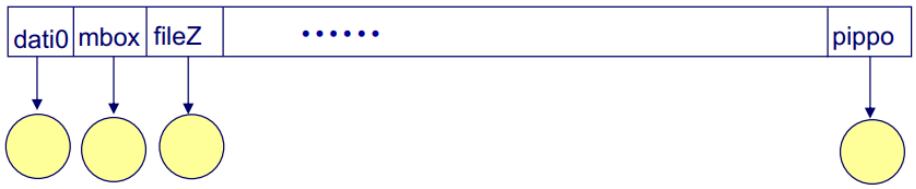
Il descrittore del file è la struttura dati che contiene gli attributi del file.

Directory

Directory (o direttorio) è lo strumento per organizzare i file all'interno del file system, è realizzata mediante una struttura dati che associa al nome di ogni file le informazioni che consentono di localizzarlo in memoria di massa.

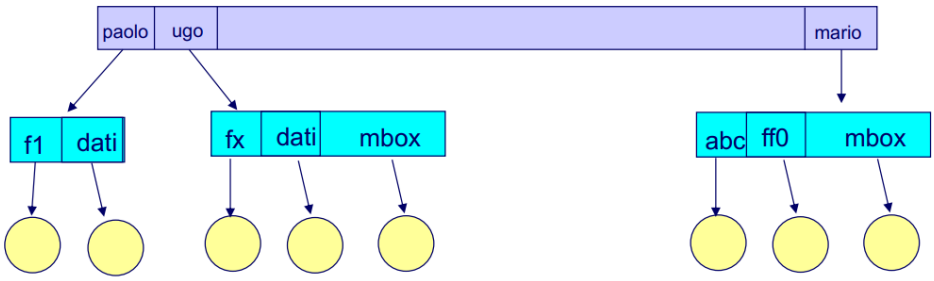
La struttura logica delle directory può variare a seconda del SO. Schemi più comuni sono:

- a un livello: una sola directory per ogni file system.

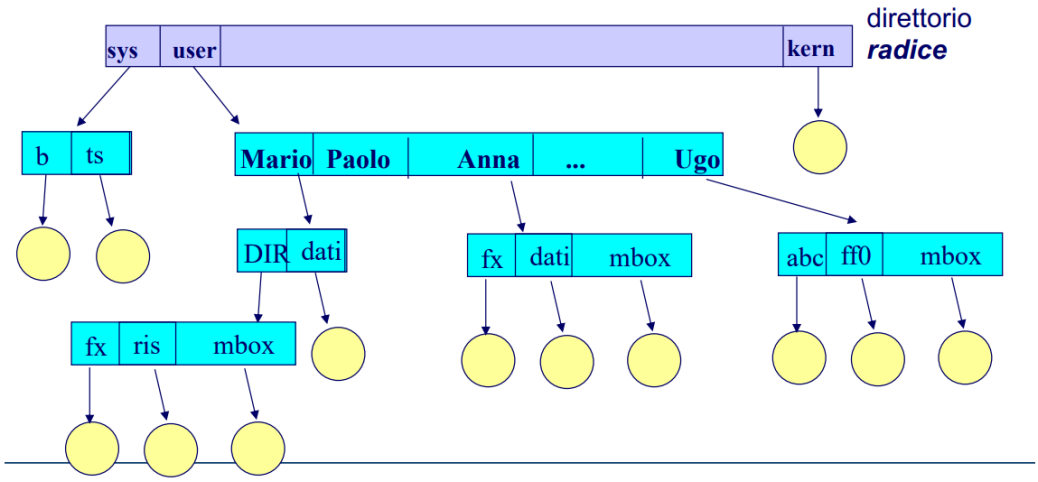


  I problemi di un tale schema sono l'unicità dei nomi e la multiutenza (come separare i file dei diversi utenti).

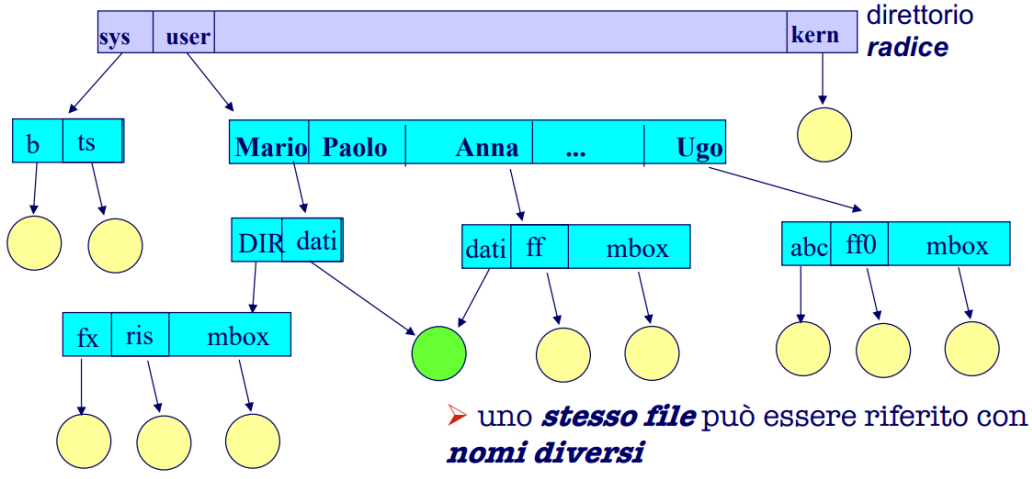
- a due livelli: il primo livello contiene una directory per ogni utente del sistema, mentre il secondo livello è rappresentato dalla directory dell'utente, la quale è a un livello.



- ad albero: organizzazione gerarchica a N livelli. Ogni direttorio può contenere file e altri direttori.



- a grafo aciclico: estende la struttura ad albero con la possibilità di inserire link differenti allo stesso file.



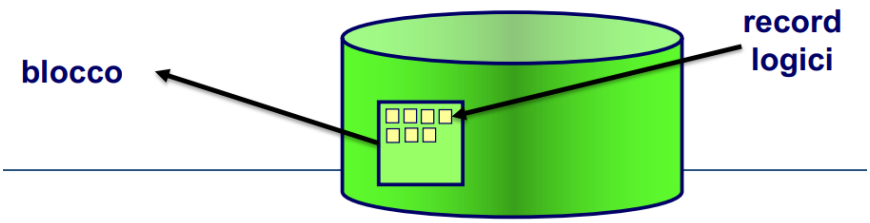
Una singola unità di memorizzazione secondaria (es. Disco) può contenere più partizioni.

Accesso

Compito del SO è consentire l'accesso on-line ai file: ogni volta che un processo modifica il contenuto di un file, tale cambiamento è immediatamente visibile a tutti gli altri processi.

Per migliorare l'efficienza il SO mantiene in memoria una struttura che registra i file attualmente in uso e viene fatto il «memory mapping» dei file aperti, ovvero i file aperti (o porzioni di essi) vengono temporaneamente copiati in memoria centrale.

Ogni dispositivo di memorizzazione secondaria viene partizionato in blocchi (o record fisici). Blocco: unità di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Sempre di dimensione fissa. Le applicazioni vedono il contenuto di ogni file come un insieme di record logici: Record logico: unità di trasferimento logico nelle operazioni di accesso al file (es. lettura, scrittura di blocchi). Di dimensione variabile. I record logici vengono impacchettati all'interno di blocchi.



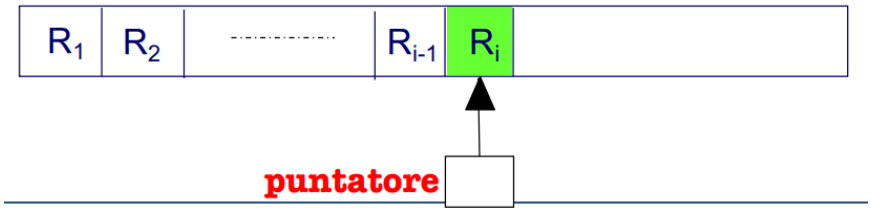
Metodi di accesso

L'accesso a file può avvenire secondo varie modalità:

- accesso sequenziale

  Il file è una sequenza [R1, R2, ..., RN] di record logici: per accedere ad un particolare record logico Ri , è necessario accedere prima agli (i-1) record che lo precedono nella sequenza. Le operazioni di accesso sono del tipo: readnext: lettura del prossimo record logico della sequenza, writenext: scrittura del prossimo record logico.

È necessario registrare la posizione corrente: puntatore al file (I/O pointer).



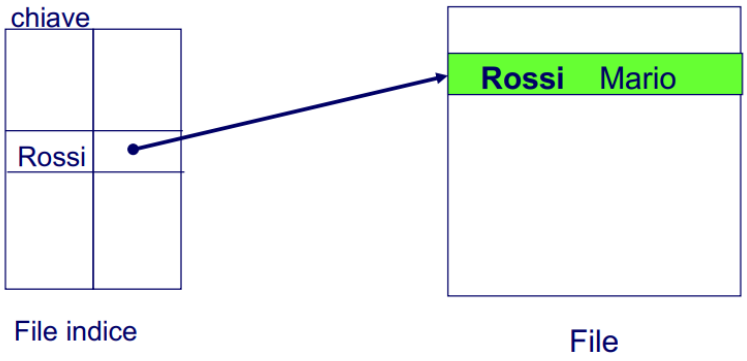
UNIX prevede questo tipo di accesso.

- accesso diretto

  Il file è un insieme {R1, R2, ..., RN} di record logici numerati: si può accedere direttamente a un particolare record logico specificandone il numero. operazioni di accesso sono del tipo read i: lettura del record logico i, write i: scrittura del record logico i.

- accesso a indice

  Ad ogni file viene associata una struttura dati contenente l'indice delle informazioni contenute.



Protezione

Il proprietario/creatore di un file dovrebbe avere la possibilità di controllare quali azioni sono consentite sul file e da parte di chi.

Per ogni file ci sono 3 classi di utenti: owner access, group access e public access.

L’amministratore può creare gruppi (con nomi unici) e inserire/eliminare utenti in/da quel gruppo.

Organizzazione fisica

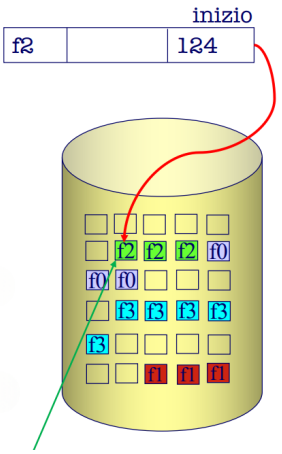
SO si occupa anche della realizzazione del file system sui dispositivi di memorizzazione secondaria.

Quali sono le tecniche più comuni per l’allocazione dei blocchi sul disco

- allocazione contigua

  Ogni file è mappato su un insieme di blocchi fisicamente contigui.

  - Vantaggi: costo della ricerca di un blocco, possibilità di accesso sequenziale e diretto
  - Svantaggi: individuazione dello spazio libero per l’allocazione di un nuovo file, frammentazione esterna: man mano che si riempie il disco, rimangono zone sempre più piccole, a volte inutilizzabili → Necessità di azioni di compattazione.

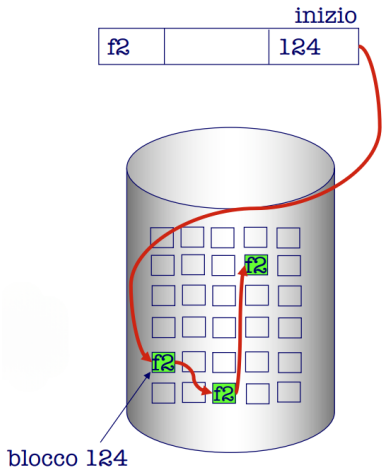


- allocazione a lista

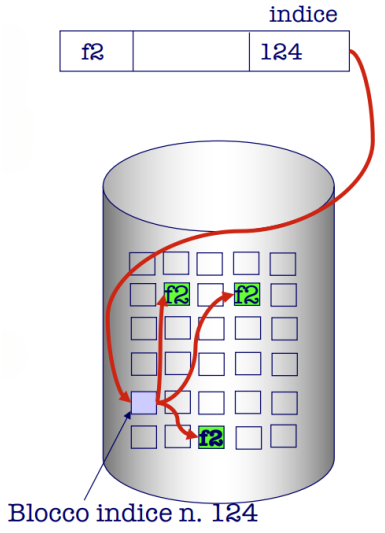
  I blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata.

  - Vantaggi: non c’è frammentazione esterna, minor costo di allocazione.

- Svantaggi: possibilità di errore se link danneggiato, maggior occupazione (spazio occupato dai puntatori), difficoltà di realizzazione dell'accesso diretto, costo della ricerca di un blocco.



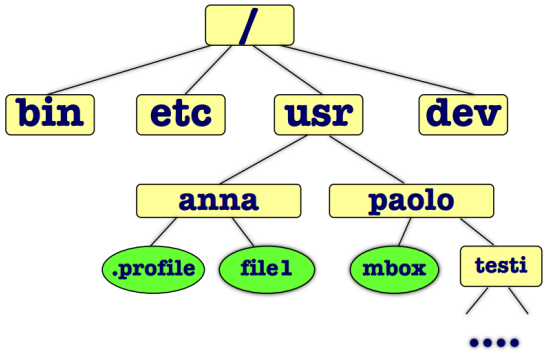
- allocazione a indice: tutti i puntatori ai blocchi utilizzati per l'allocazione di un determinato file sono concentrati in un unico blocco per quel file (blocco indice). A ogni file è associato un blocco (indice) in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file.



  - Vantaggi: stessi della lista + possibilità di accesso diretto, maggiore velocità di accesso.
  - Svantaggi: possibile scarso utilizzo dei blocchi indice.

▼ 3.2 - Il file system di UNIX

Organizzazione logica



Nel file system di UNIX tutto è un file. Esistono 3 categorie di file:

- File ordinari
- Direttori
- Dispositivi fisici (nel direttorio /dev)

Ad ogni file possono essere associati uno o più nomi simbolici ma ad ogni file è associato uno ed un solo descrittore (i-node), univocamente identificato da un intero (i-number).

Organizzazione fisica

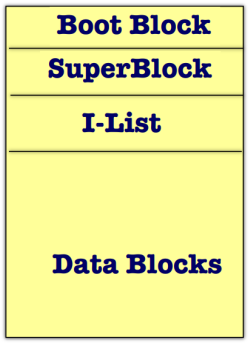
Il metodo di allocazione utilizzato in Unix è ad indice.

La superficie del disco File System è partizionata in 4 regioni/blocchi:

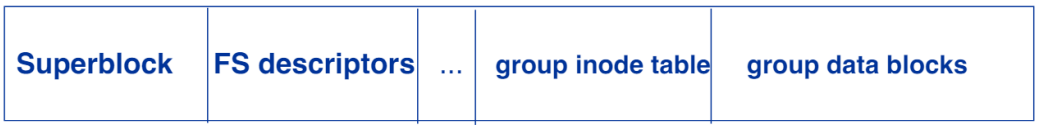
- boot block: contiene le procedure di inizializzazione del sistema
- super block: fornisce
  - i limiti delle 4 regioni
  - il puntatore a una lista dei blocchi liberi
  - il puntatore a una lista degli i-node liberi
- i-list: contiene la lista di tutti i descrittori (inode) dei file.

  Tra gli attributi contenuti nell'i-node vi sono:

    - tipo di file:
        - ordinario
        - direttorio
        - file speciale, per i dispositivi.
    - proprietario, gruppo (user-id, group-id)
    - dimensione
    - data
    - 12 bit di protezione
    - numero di links
    - 13 -15 indirizzi di blocchi
- data blocks: l'area del disco effettivamente disponibile per la memorizzazione dei file



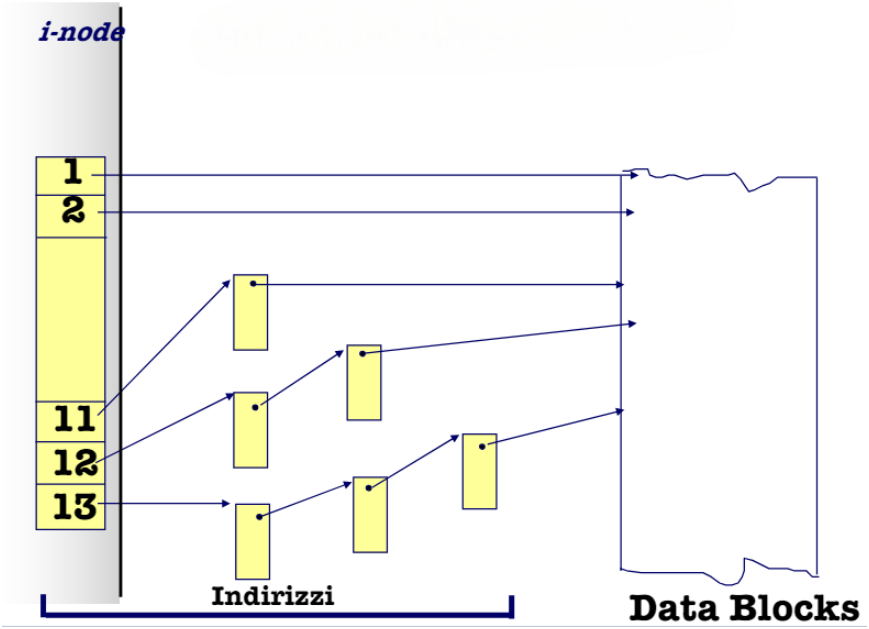
Ogni blocco ha la seguente struttura:



Indirizzamento

Il metodo di allocazione utilizzato in Unix è ad indice (a più livelli di indirizzamento). Nell'i node sono contenuti puntatori a blocchi (ad esempio 13), dei quali:

- i primi 10 indirizzi: riferiscono blocchi di dati (indirizzamento diretto)
- 11° indirizzo : indirizzo di un blocco contenente a sua volta indirizzi di blocchi dati (1 livello di indirettezza)
- 12° indirizzo: due livelli di indirettezza
- 13° indirizzo: tre livelli di indirettezza

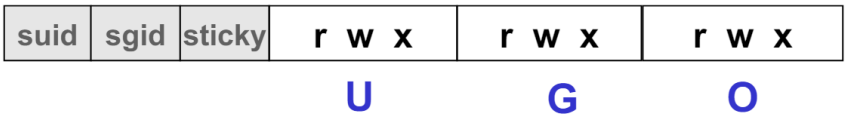


In base alle informazioni che abbiamo calcoliamo la dimensione massima di un file. Sappiamo che ogni blocco ha una dimensione di 512 byte e che ogni blocco ha la possibilità di contenere 128 indirizzi. Sappiamo dunque che 10 blocchi * 512 byte = 5KB sono accessibili direttamente, 128 blocchi * 512 byte = 64KB accessibili a 1 livello di indirettezza (indirizzo 11), 128 * 128 blocchi * 512 byte = 8MB accessibili a 2 livelli di indirettezza (indirizzo 12) e 128 * 128 * 128 blocchi * 512 byte = 1GB accessibili a 3 livelli di indirettezza (indirizzo 13). La dimensione massima del file è dell'ordine del Gigabyte (1GB + 8MB + 64KB + 5KB).

Si utilizza questa tecnica per rendere più veloce l'accesso a file di piccole dimensioni.

Protezione

Ad ogni file sono associati 12 bit di protezione (nell'i-node).



- U: utente proprietario
- G: utenti del gruppo
- O: tutti gli altri utenti

Al processo che esegue un file eseguibile è associato dinamicamente uno User-ID e Group-ID. Di default questi assumono il valore dell'utente che lancia il processo. È possibile però modificare questa impostazione settando suid e sgid a 1, i quali, se settati, associano al processo che esegue il file lo User-Id e Group-Id del proprietario del file (chi lancia il processo assume temporaneamente l'identità del proprietario).

Il bit Save-Text-Image (Sticky) consente di stabilire se l'immagine del processo deve essere mantenuta in area di swap anche dopo che il processo è terminato. Questo consente una maggior velocità di (ri) avvio.

In Unix utenti, gruppi e password sono registrati nel file /etc/passwd, accessibile in scrittura solo dal proprietario. Il comando passwd permette di modificare il file /etc/passwd, "impersonando" il superutente (root).

Per modificare i bit di protezione si può utilizzare invece il comando chmod o la system call chmod().

Direttorio

Anche il direttorio è rappresentato nel file system da un file. Ogni file-direttorio contiene un insieme di record logici con la seguente struttura:

ogni record rappresenta un file appartenente al direttorio.

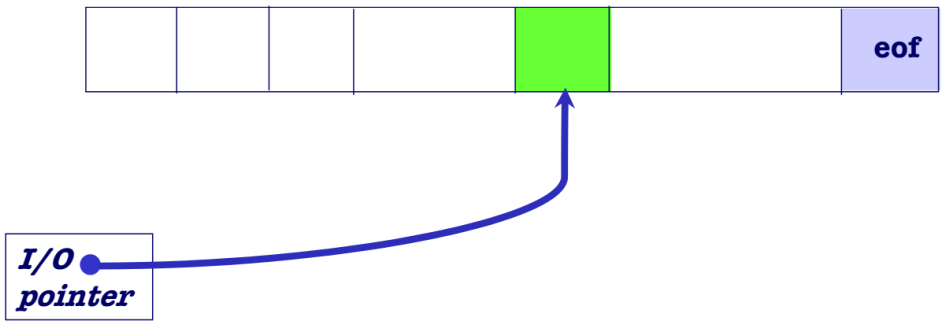
Virtual file system

Linux implementa di default il file system Ext, ma prevede l'integrazione con filesystem diversi da Ext, grazie al Virtual File System.

Accesso al file system

Accesso a file

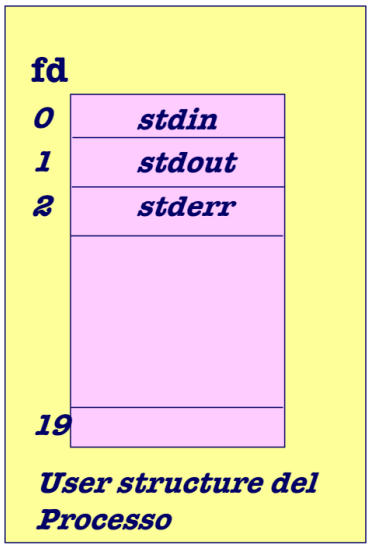
Il file in Unix è visto come un insieme di byte. L'accesso ad esso è di tipo sequenziale, e il puntatore al file (I/O pointer) registra la posizione corrente.



L'accesso ad un file deve sempre avvenire tra l'apertura e la chiusura di esso, e può essere di diverso tipo (lettura, scrittura, lettura e scrittura).

Strutture dati di accesso

A ogni processo è associata una tabella dei file aperti. Ogni elemento della tabella (file descriptor) rappresenta un file aperto dal processo ed è individuato da un indice intero. I file descriptor 0,1,2 individuano rispettivamente standard input, output, error (aperti automaticamente). La tabella dei file aperti del processo è allocata nella sua user structure.

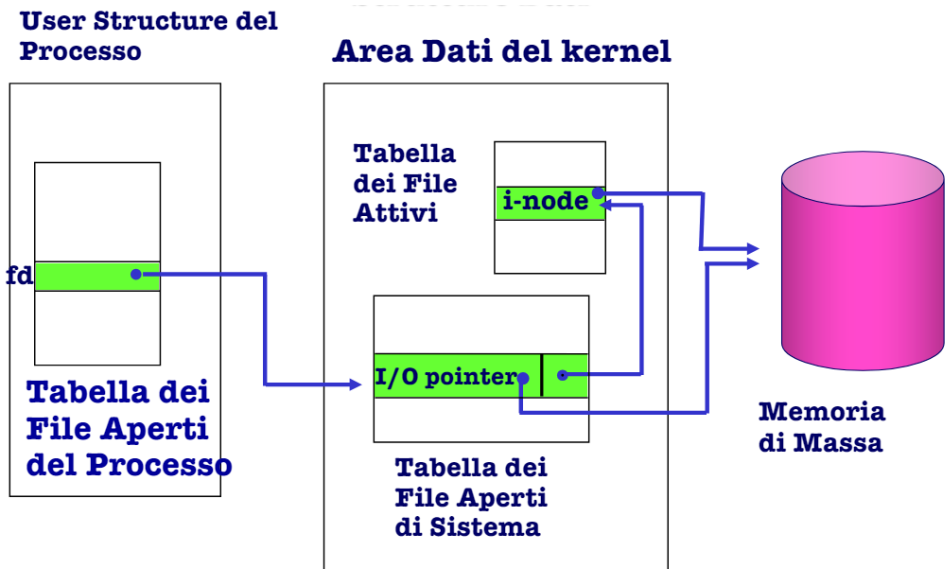


Strutture dati del kernel

Per realizzare l'accesso ai file, il sistema operativo utilizza due strutture dati globali, allocate nell'area dati del kernel:

- la tabella dei file attivi: per ogni file aperto, contiene una copia del suo i-node.
- la tabella dei file aperti di sistema: ha un elemento per ogni operazione di apertura relativa a file aperti (e non ancora chiusi); ogni elemento contiene:
  - l'I/O pointer, che indica la posizione corrente all'interno del file
  - un puntatore all'i-node del file nella tabella dei file attivi

  Più processi possono infatti accedere contemporaneamente allo stesso file, ma con I/O pointer distinti.



System call

Unix permette ai processi di accedere a file, mediante un insieme di system call, tra le quali:

- apertura/creazione: open, creat
- chiusura: close
- lettura: read
- scrittura: write
- accesso «diretto»: lseek

```
int open(char nomefile[], int flag, [int mode]);
```

Flag esprime il modo di accesso:

- O_RDONLY (= 0), accesso in lettura
- O_WRONLY(= 1), accesso in scrittura
- O_APPEND (= 2), accesso in scrittura, append (I/O pointer posizionato alla fine del file)
- O_RDWR, accesso in lettura e scrittura

Inoltre, è possibile abbinare ai tre modi precedenti, altri modi (mediante il connettore |); ad esempio:

- O_CREAT, per accesso in scrittura: se il file non esiste, viene creato
- O_TRUNC, per accesso in scrittura: la lunghezza del file viene troncata a 0

Mode è un parametro richiesto soltanto se l'apertura determina la creazione del file (flag O_CREAT): in tal caso, mode specifica i bit di protezione (ad esempio, codifica ottale).

Il valore restituito dalla open è il file descriptor associato al file, o -1 in caso di errore.

```
int creat(char nomefile[], int mode);
```

Mode specifica i 12 bit di protezione per il nuovo file.

Il valore restituito dalla creat è il file descriptor associato al file, o -1 in caso di errore.

Se la creat ha successo, il file viene aperto in scrittura.

```
int close(int fd);
```

Fd è il file descriptor del file da chiudere.

Restituisce l'esito della operazione (0, in caso di successo, <0 in caso di insuccesso).

```
int read(int fd,char *buf,int n);
```

Fd è il file descriptor del file. Buf è l'area in cui trasferire i byte letti. N è il numero di caratteri da leggere.

In caso di successo, restituisce un intero positivo (<=n) che rappresenta il numero di caratteri effettivamente letti. È previsto un carattere di End-Of-File che marca la fine del file (da tastiera:^D).

```
int write(int fd,char *buf,int n);
```

Buf è l'area da cui trasferire i byte scritti. N è il numero di caratteri da scrivere.

In caso di successo, restituisce un intero positivo che rappresenta il numero di caratteri effettivamente scritti.

```
lseek(int fd, int offset,int origine);
```

Offset è lo spostamento (in byte) rispetto all'origine.

Origine può valere:

- 0: inizio file (SEEK_SET)
- 1: posizione corrente (SEEK_CUR)
- 2 :fine file (SEEK_END)

In caso di successo, restituisce un intero positivo che rappresenta la nuova posizione.

Gestione dei file

System call

I processi possono gestire i file tramite alcune system call, tra le quali:

- cancellazione: unlink
- linking: link
- verifica dei diritti di accesso: access
- verifica degli attributi: stat
- modifica diritti di accesso: chmod
- modifica proprietario: chown

```
int unlink(char *name);
```

Ritorna 0, se OK, altrimenti -1.

In generale, l'effetto della system call unlink è decrementare di 1 il numero di link del file dato (nell'i-node); nel caso in cui il numero dei link risulti 0, allora il file viene cancellato.

```
int link(char *oldname, char * newname);
```

- oldname è il nome del file esistente
- newname è il nome associato al nuovo link

Per aggiungere un link a un file esistente. Ritorna 0, in caso di successo; -1 se fallisce.

```
int access (char * pathname, int amode);
```

- Il parametro amode esprime il diritto da verificare e può essere:
  - 04 read access
  - 02 write access
  - 01 execute access
  - 00 existence

access restituisce il valore 0 in caso di successo (diritto verificato), altrimenti -1.

nota: access verifica i diritti dell'utente, cioè fa uso del real uid del processo (non usa effective uid).

```
int stat(const char *path, struct stat *buf);
```

- il parametro buf è il puntatore a una struttura di tipo stat, nella quale vengono restituiti gli attributi del file (definito nell'header file <sys/stat.h>).

  ```
  struct stat {
      dev_t st_dev; /* ID of device containing file */
      ino_t st_ino; /* i-number */
  ```

```
    mode_t st_mode; /* protection & file type */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system
    blkcnt_t st_blocks; /* number of blocks allocated
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last status change */
};
```

Per interpretare il valore di st_mode, sono disponibili alcune costanti e macro (ad esempio S_ISREG(mode): è un file regolare?, S_ISDIR(mode): è una directory?, S_ISCHR(mode): è un dispositivo a caratteri?, S_ISBLK(mode): è un dispositivo a blocchi (file speciale)? (flag S_IFBLK)). Si utilizzano nel seguente modo:

```
S_ISREG(sb.st_mode)
```

Per leggere gli attributi di un file.

Ritorna 0 in caso di successo, -1 in caso di errore.

```
int chmod (char *pathname, char *newmode);
```

- newmode contiene i nuovi diritti

```
int chown(char *pathname, int owner, int group);
```

- owner è l'uid del nuovo proprietario
- group è il gid del gruppo

Gestione dei direttori

System call

- chdir: per cambiare direttorio (v. Comando cd)
- opendir, closedir: apertura e chiusura di direttori
- readdir: lettura di direttorio

```
int chdir (char *nomedir);
```

Ritorna 0 in caso di successo e -1 in caso di fallimento.

```
DIR *opendir (char *nomedir);
```

La funzione restituisce un valore di tipo puntatore a DIR (NULL in caso di insuccesso).

```
int closedir (DIR *dir);
```

Ritorna 0 in caso di successo e -1 in caso di fallimento.

```
dirent *readdir(DIR *dir);
```

- dir è il puntatore al direttorio da leggere (valore restituito da opendir)

La funzione restituisce un puntatore a dirent diverso da NULL se la lettura ha avuto successo, altrimenti restituisce NULL.

```
struct dirent {
    long d_ino; /* i-number */
    off_t d_off; /* offset del prossimo */
    unsigned short d_reclen; /* lunghezza del record */
    unsigned short d_namelen; /* lunghezza del nome */
    char *d_name; /* nome del file */
}
```

```
int mkdir (char *pathname, int mode);
```

- mode esprime i bit di protezione.

Restituisce il valore 0 in caso di successo, altrimenti un valore negativo.

Interazione tra processi

I processi Unix non possono condividere memoria (modello ad ambiente locale).

L'interazione tra processi può avvenire:

- mediante la condivisione di file.
- attraverso specifici strumenti di Inter Process Communication:
  per la comunicazione tra processi sulla stessa macchina:
  - pipe (tra processi della stessa gerarchia)
  - fifo (qualunque insieme di processi)

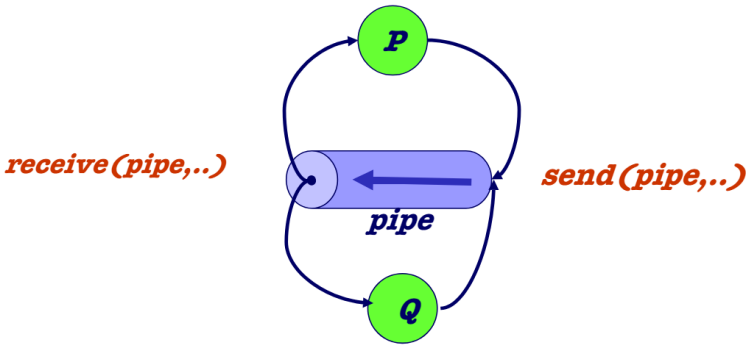
  per la comunicazione tra processi in nodi diversi della stessa rete:
  - socket

Pipe

La pipe è un canale di comunicazione tra processi:

- unidirezionale: accessibile ad un estremo in lettura ed all'altro in scrittura
- molti-a-molti:
  - più processi possono spedire messaggi attraverso la stessa pipe
  - più processi possono ricevere messaggi attraverso la stessa pipe
- FIFO

Uno stesso processo può sia depositare messaggi nella pipe (send), mediante il lato di scrittura che prelevare messaggi dalla pipe (receive), mediante il lato di lettura.




```
int pipe(int fd[2]);
```

- fd è il puntatore a un vettore di 2 file descriptor, che verranno inizializzati dalla system call in caso di successo:
  - fd[0] rappresenta il lato di lettura della pipe
  - fd[1] è il lato di scrittura della pipe

  Ogni lato di accesso alla pipe è visto dal processo in modo omogeneo al file (file descriptor), si può dunque accedere alla pipe mediante le system call di accesso a file: read, write.

La system call pipe restituisce 0 se ha successo e un valore negativo in caso di fallimento.

Il canale (la pipe) ha capacità limitata, dunque read e write possono essere sospensive nel caso di pipe vuota e piena.

Siccome i file descriptor della pipe vengono assegnati al processo che l'ha creata e viene ereditata dai suoi figli, soltanto i processi appartenenti a una stessa gerarchia (cioè, che hanno un antenato in comune) possono scambiarsi messaggi mediante pipe.

Ogni processo può chiudere un estremo della pipe con una close. Un estremo della pipe viene effettivamente chiuso (cioè, la comunicazione non è più possibile) quando tutti i processi che ne avevano visibilità hanno compiuto una close.

Ridirezione di I/O e Piping

Ridirezione e piping di comandi

La shell di unix prevede l'uso degli operatori:

- di ridirezione (>,<,>>)
- di piping (|)

```
comando > F
```

l'output del Comando viene scritto nel file F (invece che sul dispositivo di standard output).

```
comando >> F
```

l'output del comando viene aggiunto in coda (append) al contenuto del file F (invece che sul dispositivo di standard output).

```
comando < F
```

l'input del comando viene acquisito dal file F (invece che dal dispositivo di standard input).

```
comando1 | comando2 | comando3
```

l'output di comando1 viene ridiretto nell'input di comando2; → l'output di comando2 viene ridiretto nell'input di comando3.

Ridirezione in output

È possibile effettuare la ridirezione in output tramite codice nel seguente modo:

1. chiudo il file descriptor 1 (standard output): close(1); → libero l'elemento di indice 1 nella tab. dei file aperti
2. apro il file dato : fd=open("pippo", O_WRONLY); → il primo elemento libero della tab. dei file aperti è quello di indice 1, pertanto fd vale 1; quindi ogni write sul file descriptor 1 scriverà nel file pippo
3. eseguo il comando : execlp(argv[1],argv[1],(char *)0); → se il comando scrive sullo std.output, le informazioni corrispondenti verranno scritte

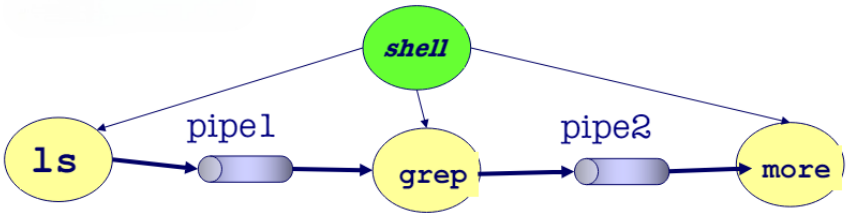
nel file pippo

Ridirezione in input

1. chiudo il file descriptor 0 (standard input): close(0); → libero l'elemento di indice 0 nella tab. dei file aperti
2. apro il file dato : fd=open("testo.txt", O_RDONLY); → il primo elemento libero della tab. dei file aperti è quello di indice 0, pertanto fd vale 0; quindi ogni read dal file descriptor 0 leggerà dal file testo.txt
3. eseguo il comando : execlp("grep", "grep", "ciao", (char *)0); → se comando grep legge dal file di standard input le informazioni contenute nel file testo.txt.

Piping di comandi

Per ogni comando viene creato un processo, il quale comunica con il successivo tramite la pipe.



Per far sì che venga utilizzata la pipe come input e output si utilizza la system call dup:

```
int dup(int fd)
```

la quale copia l'elemento fd della tabella dei file aperti nella prima posizione libera della tabella. Restituisce il nuovo file descriptor (del file aperto copiato), oppure -1 (in caso di errore).

Mediante la dup è possibile modificare l'input e l'output chiudendo, prima di richiamarla sul lato di lettura e scrittura della pipe, i dispositivi di standard input e output rispettivamente.

```
close(1); /* chiudo disp. stdout*/
dup(fd[1]); /* ridirigo stdout sul lato di scrittura della
```

```
close(0);
dup(fd[0]); /* ridirigo stdin sulla pipe */
```

Fifo

La pipe ha due svantaggi:

- consente la comunicazione solo tra processi in relazione di parentela.
- non è persistente: viene distrutta quando terminano tutti i processi che la usano.

Per realizzare la comunicazione tra una coppia di processi non appartenenti alla stessa gerarchia è possibile utilizzare un'altra struttura dati, la FIFO. La FIFO è un canale unidirezionale gestito con politica First-In-First-Out, rappresentata da un file nel file system.

Per creare una fifo si usa il seguente comando:

```
int mkfifo(char* pathname, int mode);
```

- pathname è il nome della fifo
- mode esprime i permessi

Restituisce 0, in caso di successo, un valore negativo in caso contrario.

La FIFO è aperta e acceduta con le stesse system call dei file.

▼ 4.0 - Gestione memoria

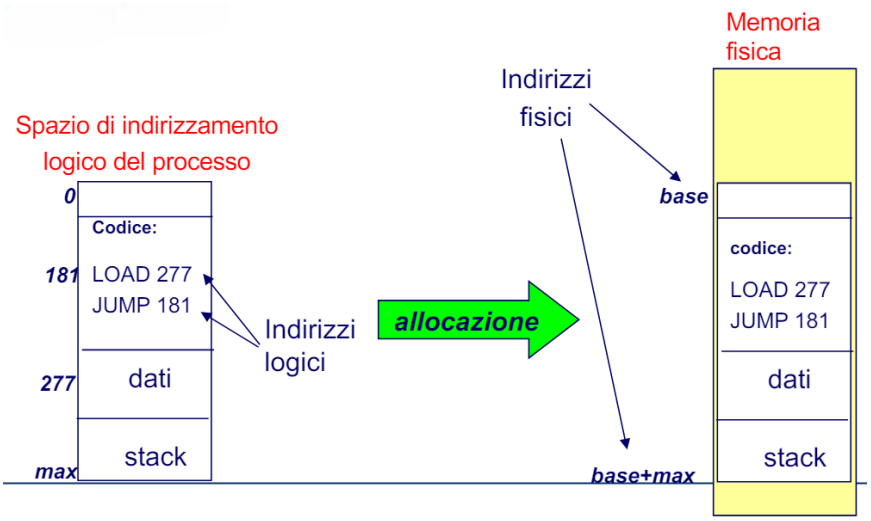
▼ 4.1 - Introduzione alla gestione della memoria

In un sistema multiprogrammato l'esecuzione concorrente dei processi determina la necessità di mantenere più processi in memoria centrale: il SO deve gestire la memoria in modo da consentire la presenza contemporanea di più processi.

Indirizzi

Ogni processo “vede” di un proprio spazio di indirizzamento logico [0,max] che contiene il codice e i dati necessari per la sua esecuzione:

Esecuzione di un programma: lo spazio di indirizzamento logico del processo deve essere allocato nella memoria fisica.



Indirizzi simbolici: ogni programma in forma sorgente contiene indirizzi simbolici, ovvero riferimenti a dati/istruzioni mediante nomi simbolici (es: nomi di variabili, funzioni ecc.). Il processo di compilazione trasforma ogni indirizzo simbolico in un indirizzo logico.

Ad ogni indirizzo logico/simbolico viene fatto corrispondere un indirizzo fisico: l’associazione tra indirizzi relativi e indirizzi assoluti viene detta binding. Il Binding può essere effettuato:

- staticamente
  a tempo di compilazione o di caricamento, durante i quali vengono generati indirizzi assoluti.
- dinamicamente
  a tempo di esecuzione. Durante l'esecuzione lo spazio di indirizzamento di un processo (o parti di esso) può essere spostato (riallocato) nella memoria fisica.

Caricamento/collegamento dinamico

In alcuni casi è possibile caricare in memoria una funzione/procedura a runtime solo quando avviene la chiamata, questo viene chiamato caricamento dinamico.

Loader di collegamento rilocabile: carica e collega dinamicamente la funzione al programma che la usa. La funzione potrebbe essere usata da più processi simultaneamente. Compito SO è concedere/controllare l'accesso di un processo allo spazio di un altro processo.

▼ 4.2 - Tecniche di allocazione in memoria centrale

Tecniche di allocazione memoria centrale

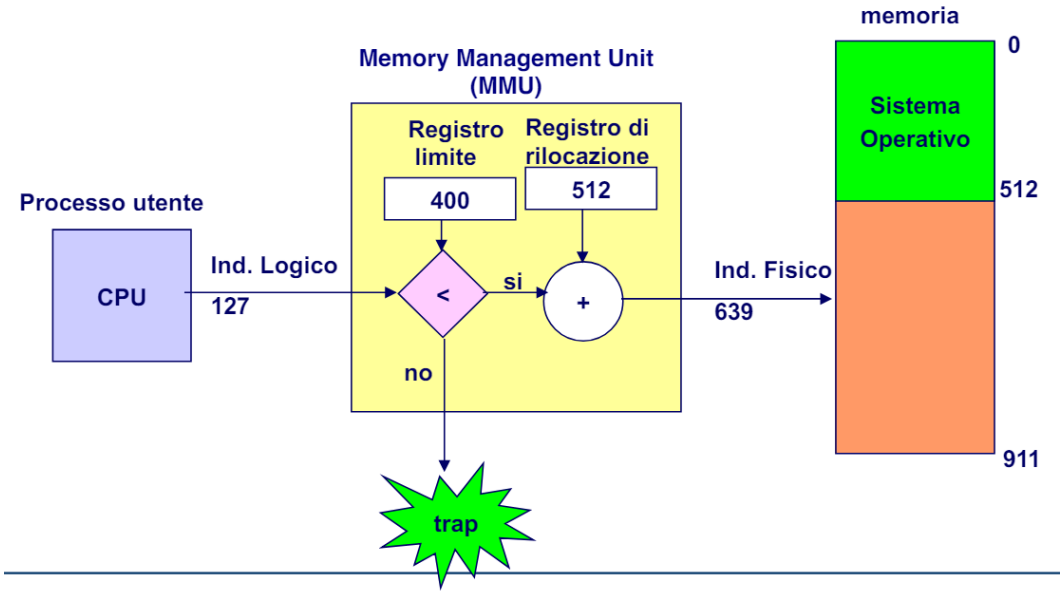
Le tecniche di allocazione stabiliscono come vengono allocati codice e dati dei processi in memoria centrale. Queste possono essere:

- Allocazione Contigua:
  - a partizione singola
  - a partizioni multiple
- Allocazione non contigua:
  - paginazione
  - segmentazione

Allocazione contigua a partizione singola

Primo approccio molto semplificato: la parte di memoria disponibile per l'allocazione dei processi di utente non è partizionata: un solo processo

alla volta può essere allocato in memoria: non c'è multiprogrammazione.



Allocazione contigua a partizioni multiple

Partizioni multiple vengono usate per la multiprogrammazione: ad ogni processo caricato viene associata un'area di memoria distinta (partizione). Due casi:

- partizioni fisse

  La memoria fisica disponibile per l'allocazione dei processi è suddivisa a priori in un numero prefissato di partizioni. La dimensione di ogni partizione è fissata a priori.

  Problemi:

  - frammentazione interna: sottoutilizzo della partizione
  - grado di multiprogrammazione limitato al numero di partizioni
  - dimensione massima dello spazio di indirizzamento di un processo limitata dalla dimensione della partizione più estesa

- partizioni variabili

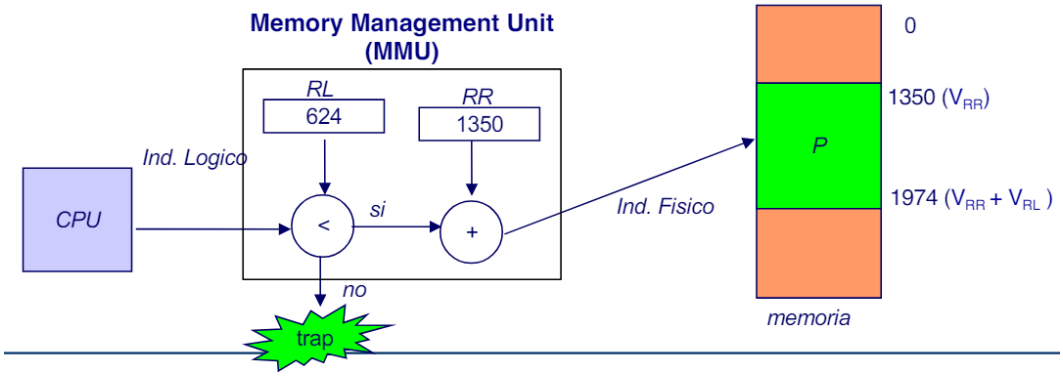
  Ogni partizione è creata dinamicamente e dimensionata in base alla dimensione del processo da allocare: quando un processo viene

caricato, SO cerca un'area sufficientemente grande per allocarvi dinamicamente la partizione associata.

Problemi:

- scelta dell'area in cui allocare: varie politiche possibili. Es: best fit, worst fit, first fit, ...
- frammentazione esterna - man mano che si allocano nuove partizioni, la memoria libera è sempre più frammentata → necessità di compattazione periodica

Ad ogni processo P è associata una coppia di valori $< V_{RR}, V_{RL} >$. Quando P viene schedulato, dispatcher carica RR e RL con i valori associati al processo.



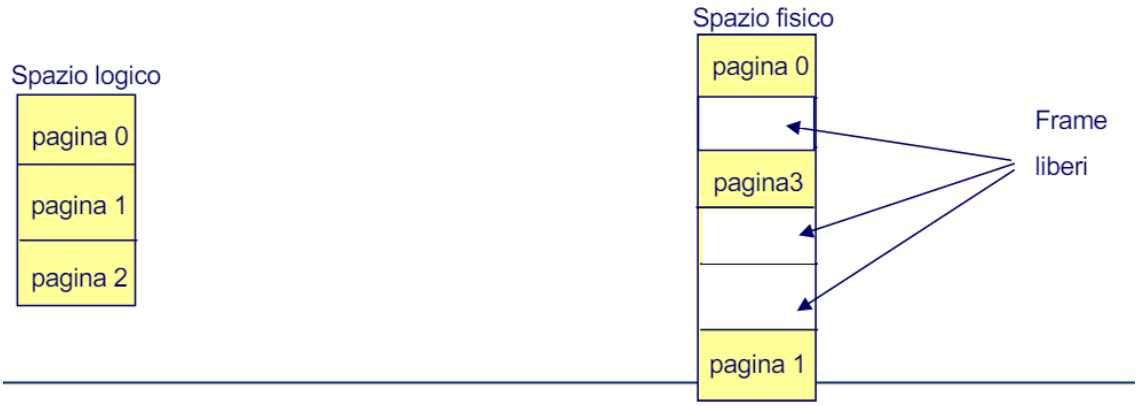
Paginazione

Allocazione contigua a partizioni multiple: il problema principale è la frammentazione esterna; per risolverlo: Allocazione non contigua → paginazione.

Idea di base: partizionamento spazio fisico di memoria in pagine (frame) di dim costante e limitata (ad es. 4KB) sulle quali mappare porzioni dei processi da allocare.

- Spazio fisico: insieme di frame di dim D_f costante prefissata
- Spazio logico: insieme di pagine di dim uguale a D_f

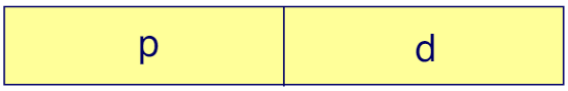
Ogni pagina logica di un processo viene mappata su una pagina fisica in memoria centrale.



Vantaggi:

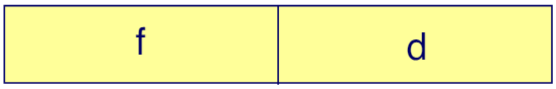
- Pagine logiche contigue possono essere allocate su pagine fisiche non contigue → non c'è frammentazione esterna
- Le pagine sono di dimensione limitata: per ogni processo la frammentazione interna è limitata dalla dimensione del frame
- È possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo (vedi memoria virtuale nel seguito)

Struttura dell'indirizzo logico:



- p numero di pagina logica
- d offset della cella rispetto all'inizio della pagina

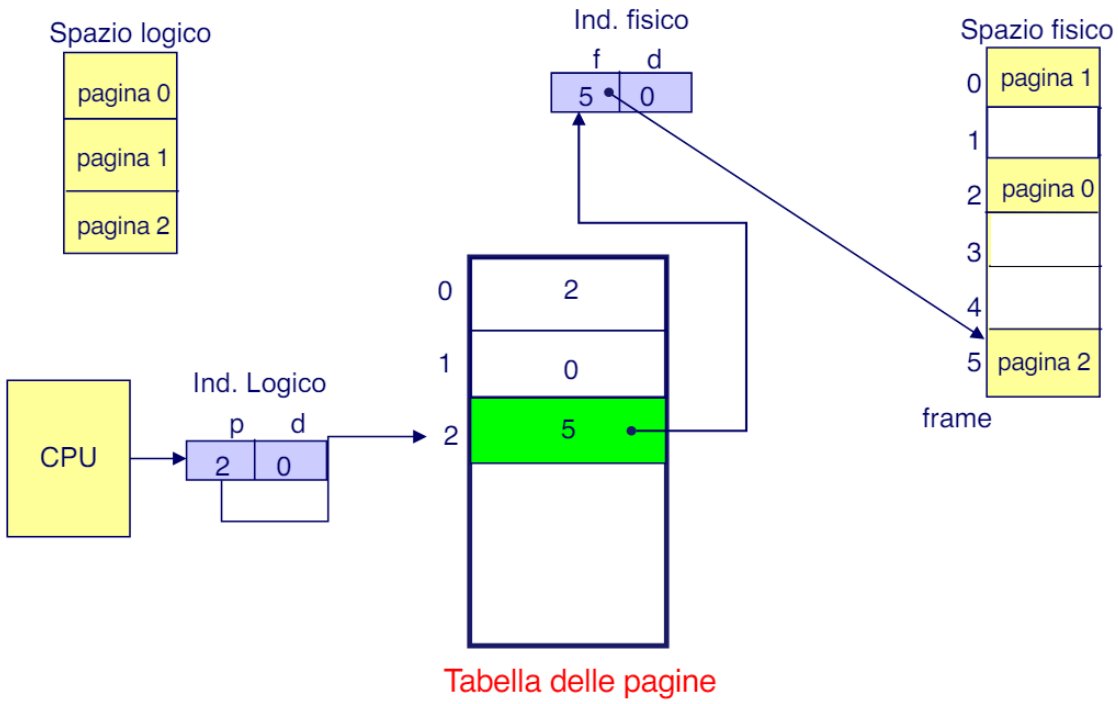
Struttura dell'indirizzo fisico:



- f numero di frame (pagina fisica)
- d offset della cella rispetto all'inizio del frame

Il Binding tra indirizzi logici e fisici può essere realizzato mediante una struttura dati, detta tabella delle pagine.

Ogni tabella delle pagine è associata ad un particolare processo: ogni elemento significativo della tabella associa a ogni pagina logica p la pagina fisica f nella quale è p allocata [L'offset è invariato].



Realizzazione della tabella delle pagine

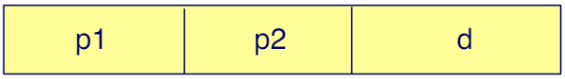
Problemi da considerare: tabella può essere molto grande e la traduzione (ind.logico → ind. fisico) deve essere il più veloce possibile.

Una soluzione può essere quella di utilizzare memoria centrale + cache: (Translation Look-aside Buffers, TLB) per memorizzare la tabella delle pagine e velocizzare l'accesso. Ciò comporta che la tabella delle pagine è allocata in memoria centrale, e una parte di essa (di solito, quella relativa alle pagine accedute più di frequente o più di recente) è copiata in cache. Se la coppia (p,f) è già presente in cache l'accesso è veloce; altrimenti SO deve trasferire la coppia richiesta dalla tabella delle pagine (in memoria centrale) in TLB.

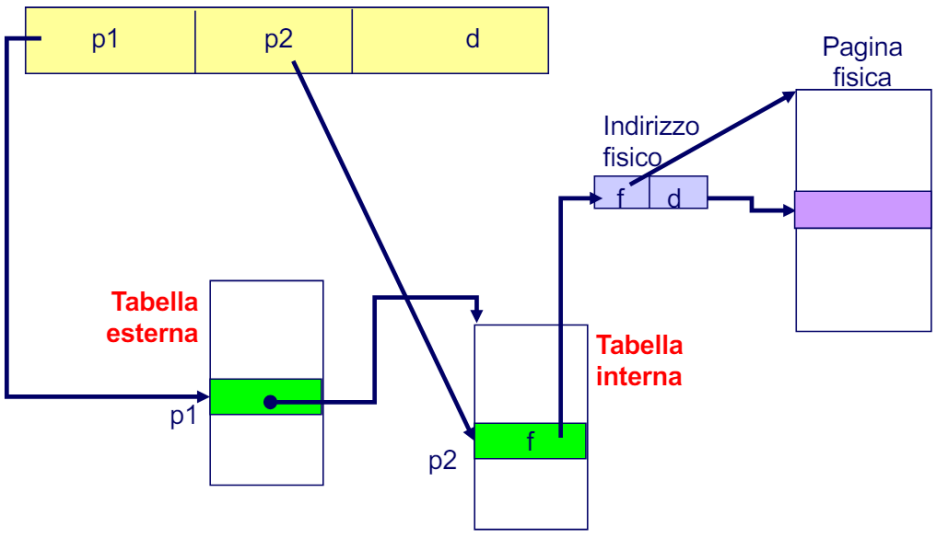
La tabella delle pagine ha dimensione fissa e non sempre viene utilizzata completamente, dunque per distinguere gli elementi significativi da quelli non utilizzati si possono utilizzare due metodi:

- Bit di validità: ogni elemento contiene un bit. se è a 1, entry valida (pagina appartiene allo spazio logico del processo), se è 0, entry non valida.
- Page Table Length Register: registro che contiene il numero degli elementi validi nella tabella delle pagine.

Lo spazio logico di indirizzamento di un processo può essere molto esteso: elevato numero di pagine e tabella delle pagine di grandi dimensioni. Sì può dunque utilizzare la paginazione a più livelli: si applica ancora la tecnica di paginazione alla tabella della pagine. Struttura dell'indirizzo logico:



- p1 indice di «pagina» nella tabella esterna
- p2 offset nella tabella interna
- d offset cella all'interno della pagina fisica



Vantaggi:

- possibilità di indirizzare spazi logici di dimensioni elevate riducendo i problemi di allocazione delle tabelle

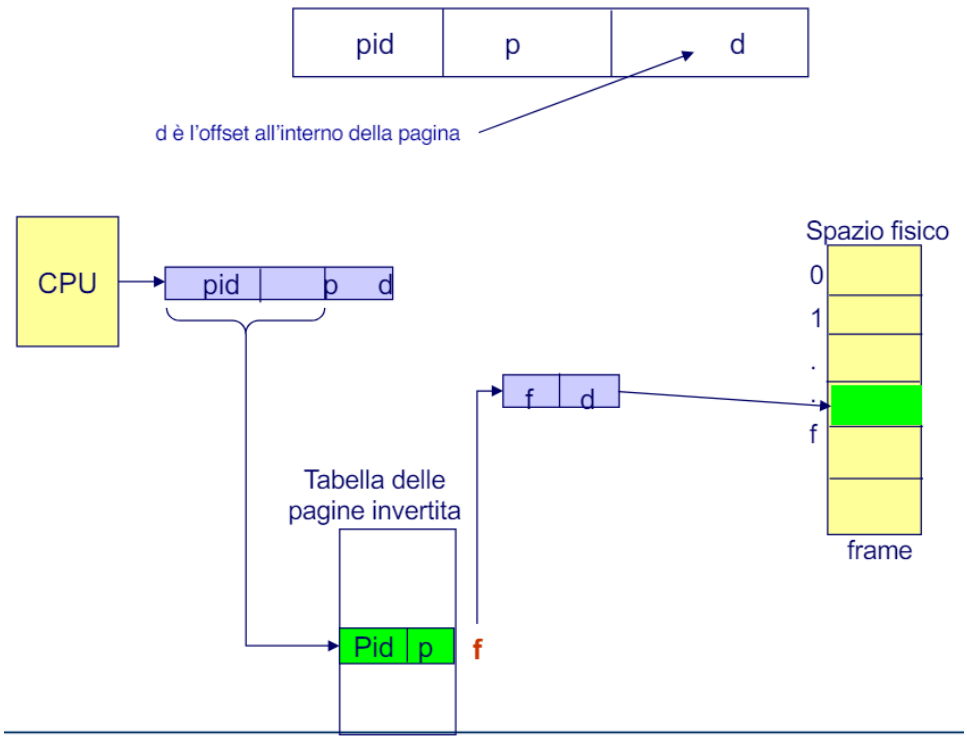
- possibilità di mantenere in memoria soltanto le tabelle interne (secondo livello) che servono

Svantaggi:

- tempo di accesso più elevato: per tradurre un indirizzo logico sono necessari più accessi in memoria

Un altra possibile implementazione della tabella delle pagine è la tabella delle pagine invertita, ovvero un'unica struttura dati globale che ha un elemento per ogni frame (indirizzo pari alla posizione nella tabella) e, in caso di frame allocato, contiene:

- pid: identificatore del processo a cui è assegnato il frame
- p: numero di pagina logica



Segmentazione

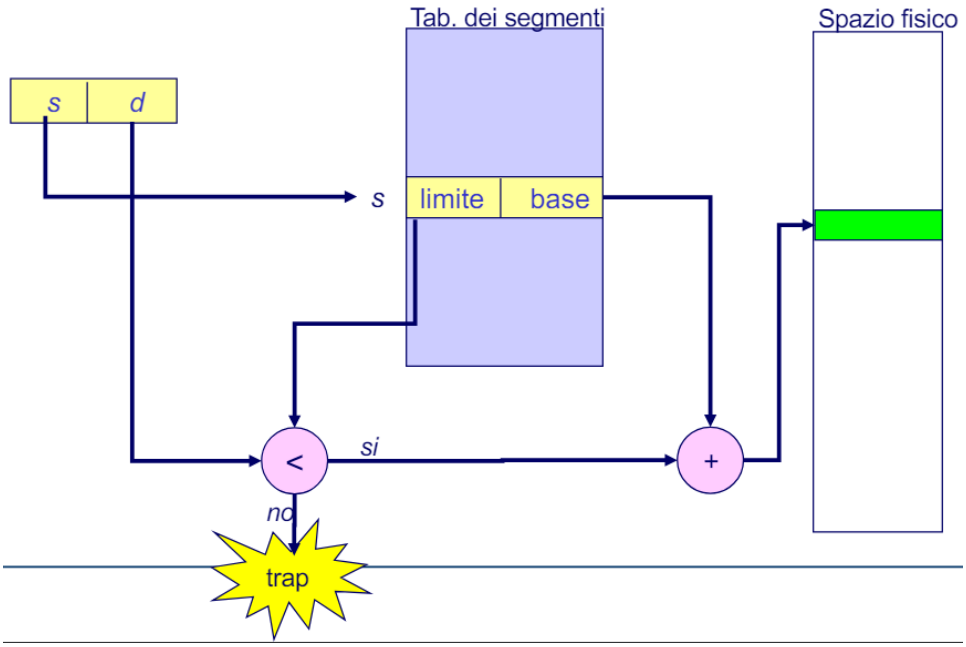
La segmentazione è una tecnica di allocazione della memoria centrale che si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti (segmenti), ognuna caratterizzata da nome e lunghezza

propri. Ogni segmento è allocato in memoria in modo contiguo. Un esempio può essere quello di dividere lo spazio logico in codice, dati, stack e heap.

Ogni indirizzo è costituito dalla coppia: <segmento, offset>.

Tabella dei segmenti

La tabella dei segmenti è un supporto hardware alla segmentazione la quale ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica mediante la coppia <base, limite>, dove base indica l'indirizzo della prima cella del segmento nello spazio fisico, mentre limite indica la dimensione del segmento.



Come nel caso delle partizioni variabili, la segmentazione porta a frammentazione esterna. Una soluzione può essere quella dell'allocazione dei segmenti utilizzando alcune tecniche, come la best fit e la worst fit, oppure utilizzando la compattazione.

Segmentazione paginata

Segmentazione e paginazione possono essere combinate. In questo modo lo spazio logico viene segmentato (specialmente per motivi di protezione) e ogni segmento suddiviso in pagine.

Vantaggi:

- eliminazione della frammentazione esterna
- non necessario mantenere in memoria l'intero segmento, ma è possibile caricare soltanto le pagine necessarie

Come strutture dati occorre mantenere una tabella dei segmenti e una tabella delle pagine per ogni segmento.

Overlay

In generale, la memoria disponibile può non essere sufficiente ad accogliere codice e dati di un processo. Una possibile soluzione è l'overlay: codice e dati di un processo vengono suddivisi dal programmatore in parti separate che vengono caricate e scaricate dinamicamente (dal gestore di overlay).

Un esempio può essere quello dell'utilizzo di un assembler a 2 passi, il quale produce l'eseguibile di un programma assembler mediante 2 fasi sequenziali:

1. Creazione della tabella dei simboli (passo 1)
2. Generazione dell'eseguibile (passo 2)

Memoria virtuale

La dimensione della memoria può rappresentare un vincolo importante, riguardo a

- dimensione dei processi
- grado di multiprogrammazione

Per evitare questi vincoli è possibile utilizzare un sistema a memoria virtuale. Con le tecniche viste finora l'intero spazio logico di ogni processo veniva allocato in memoria, oppure si allocava/deallocava parti dello spazio di indirizzi tramite l'overlay, ma ciò veniva fatto dal programmatore. Con un sistema a memoria virtuale il SO consente l'esecuzione di processi non completamente allocati in memoria, in modo completamente trasparente per il programmatore e per i processi stessi.

Paginazione su richiesta

Di solito la memoria virtuale è realizzata mediante tecniche di paginazione su richiesta:

- tutte le pagine di ogni processo risiedono in memoria di massa («backing store»);
- durante l'esecuzione alcune di esse vengono trasferite, all'occorrenza, da backing store a memoria centrale, e viceversa.

Esecuzione di un processo può richiedere swap-in del processo

- pager: gestisce i trasferimenti di singole pagine
- swapper: gestisce i trasferimenti di interi processi

Quindi, in generale, una pagina dello spazio logico di un processo può essere allocata in memoria centrale o in memoria secondaria. Per distinguere i due casi ogni elemento della tabella delle pagine contiene 1bit di validità, il cui valore indica:

- valore 1: la pagina è presente in memoria centrale
- valore 0: la pagina è in memoria secondaria oppure è invalida, cioè non appartiene allo spazio logico del processo. Se si tenta di accedere ad una pagina con bit di validità = 0 viene generata un interruzione al SO (page fault).

  Quando il kernel del SO riceve l'interruzione dovuta al page fault:

  1. Salvataggio del contesto di esecuzione del processo
  2. Verifica del motivo del page fault
     • riferimento illegale (violazione delle politiche di protezione) → terminazione del processo
     • riferimento legale: la pagina è in memoria secondaria
  3. Copia della pagina in un frame libero
  4. Aggiornamento della tabella delle pagine
  5. Ripristino del contesto del processo: esecuzione dell'istruzione interrotta dal page fault

Sovrallocazione

In seguito a un page fault, se è necessario caricare una pagina in memoria centrale, può darsi che non ci siano frame liberi. Si ricade dunque nella sovrallocazione.

La soluzione è quella di sostituzione di una pagina P_{vitt} («vittima») allocata in memoria con la pagina P_{new} da caricare:

In generale, la sostituzione di una pagina può richiedere 2 trasferimenti da/verso il disco:

1. per scaricare la vittima
2. per caricare la pagina nuova.
   E' possibile che la vittima non sia stata modificata rispetto alla copia residente sul disco. In questo caso la copia della vittima sul disco può essere evitata. Per questo viene introdotto il bit di modifica (dirty bit). Se settato, la pagina ha subìto almeno un aggiornamento da quando è caricata in memoria, se a 0, la pagina non è stata modificata. L'algoritmo di sostituzione può esaminare il bit di modifica della vittima ed esegue swap-out della vittima solo se il dirty bit è settato.

Per la scelta della pagina vittima si possono utilizzare diversi algoritmi:

- LFU (Least Frequently Used): sostituita la pagina che è stata usata meno frequentemente (in un intervallo di tempo prefissato). È necessario associare un contatore degli accessi ad ogni pagina, la vittima è quella con minimo valore del contatore.
- FIFO: sostituita la pagina che è da più tempo caricata in memoria. È necessario memorizzare la cronologia dei caricamenti in memoria: timestamping o gestione di una coda in cui ogni elemento rappresenta una pagina caricata in memoria.
- LRU (Least Recently Used): di solito preferibile per principio di località; viene sostituita la pagina che è stata usata meno recentemente. È necessario registrare la sequenza degli accessi alle pagine in memoria.

Thrashing

La paginazione su domanda pura, carica una pagina soltanto se strettamente necessaria. Questo porta a possibilità di thrashing, ovvero il

sistema impiega più tempo per la paginazione che per l'esecuzione.

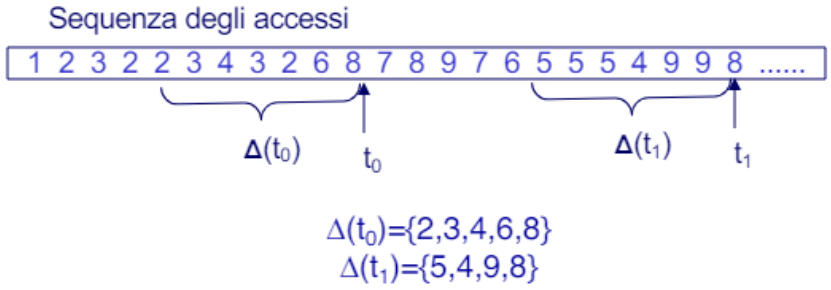
Per contrastare il thrashing si usano tecniche di gestione della memoria che si basano su pre-paginazione, ovvero si prevede il set di pagine di cui il processo da caricare ha bisogno per la prossima fase di esecuzione: "working set".

Si è osservato che un processo, in una certa fase di esecuzione usa solo un sottoinsieme relativamente piccolo delle sue pagine logiche, stimabile utilizzando due principi:

- Località spaziale: alta probabilità di accedere a locazioni vicine a locazioni appena accedute (ad esempio, elementi di un vettore, codice sequenziale, ...)
- Località temporale: alta probabilità di accesso a locazioni accedute di recente (ad esempio cicli)

Utilizzando la località temporale, dato un intero Δ, il working set di un processo P (nell'istante t) è l'insieme di pagine $\Delta(t)$ indirizzate da P nei più recenti Δ riferimenti.

Ad esempio, per $\Delta = 7$:



All'istante t vengono dunque mantenute le pagine usate dal processo nell'ultima finestra $\Delta(t)$, le altre pagine (esterne a $\Delta(t)$) possono essere sostituite.

▼ 4.3 - Esempi di gestione della memoria nei SO

Esempi di gestione della memoria nei SO

Unix

In UNIX lo spazio logico è segmentato. Nelle prime versioni vi era un allocazione contigua dei segment e non c'era memoria virtuale. In caso di difficoltà di allocazione dei processi avveniva lo swapping dell'intero spazio degli indirizzi tramite tecnica first fit.

Da BSDv3 in poi Unix usa la segmentazione paginata con l'utilizzo di memoria virtuale tramite paginazione su richiesta. Viene utilizzata la core map, ovvero una struttura dati interna al kernel che descrive lo stato di allocazione dei frame e che viene consultata in caso di page fault.

L'algoritmo di sostituzione in caso di page-fault è un LRU modificato o "algoritmo di seconda chance": ad ogni pagina viene associato un bit di uso che al momento del caricamento è inizializzato a 0. Quando la pagina è acceduta, viene settato a 1. Nella fase di ricerca di una vittima, vengono esaminati i bit di uso di tutte le pagine in memoria:

- se una pagina ha il bit di uso a 1, viene posto a 0
- se una pagina ha il bit di uso a 0, viene selezionata come vittima

L'algoritmo di sostituzione viene eseguito dal pager pagedaemon (pid=2).

Alcuni parametri da considerare per il funzionamento dello swapper sono i seguenti:

- lotsfree: numero minimo di frame liberi per evitare sostituzione di pagine
- minfree: numero minimo di frame liberi necessari per evitare swapping dei processi
- desfree: numero desiderato di frame liberi

Lo Scheduler attiva l'algoritmo di sostituzione se il numero di frame liberi < lotsfree. Se numero di frame liberi < minfree oppure il numero medio di frame liberi nell'unità di tempo < desfree, lo scheduler attiva swapper.

Linux

In Linux invece si fa uso di un'allocazione basata su segmentazione paginata a più livelli. Viene utilizzata la memoria virtuale senza working set.

MS Windows XP

In MS Windows XP viene utilizzata la segmentazione paginata con clustering delle pagine: in caso di page fault, viene caricato tutto un gruppo di pagine “attorno” a quella mancante (page cluster). Ogni processo ha infatti un working set minimo (numero minimo di pagine sicuramente mantenute in memoria) e un working set massimo (massimo numero di pagine mantenibile in memoria). Qualora la memoria fisica libera scenda sotto una soglia, SO automaticamente ristabilisce la quota desiderata di frame liberi (working set trimming), eliminando pagine appartenenti a processi che ne hanno in eccesso rispetto a working set minimo

▼ 5.0 - Shell Unix

Introduzione alla shell

La shell è un interprete comandi evoluto, la quale interpreta ed esegue comandi da standard input o da file comandi e che offre un potente linguaggio di scripting.

La shell non è unica, un sistema può metterne a disposizione varie (Bourne shell (standard), C shell, Korn shell, ecc.). L’implementazione della bourne shell in Linux è bash (/bin/bash) Ogni utente può indicare la shell preferita (v. comando chsh).

Accesso al sistema: shell di login

Per ogni utente connesso viene generato un processo dedicato (che esegue la shell). La shell di login è quella che richiede inizialmente i dati di accesso all’utente. SO verifica le credenziali dell’utente e manda in esecuzione la sua shell di preferenza.

Uscita dal sistema: logout

Per uscire da una shell qualsiasi si può utilizzare il comando exit (che invoca la system call exit() per quel processo). Per uscire dalla shell di login invece si può utilizzare il comando logout.

Esecuzione di comandi

L’esecuzione di comandi della shell può avvenire in due modi:

- comandi builtin: definiti internamente ed eseguiti dentro il processo shell, come una chiamata a funzione. Il comando help stampa la lista di tutti i comandi builtin.
- comandi external: la shell crea una shell figlio, dedicata all'esecuzione del comando.

`type <comando>` stampa il tipo (builtin/external).

Ridirezione di input e output

Possibile ridirigere input e/o output di un comando facendo sì che non si legga da stdin (e/o non si scriva su stdout) ma da file.

Ridirezione dell'input: `comando < file_input` .

Ridirezione dell'output:

- Aperto in scrittura (nuovo o sovrascritto): `comando > file_output`.
- Scrittura in append: `comando >> file_output`.

Con il piping l'output di un comando può esser rediretto e diventare l'input di un altro comando: `comando1 | comando2`.

Linguaggio della shell

Metacaratteri

Shell riconosce caratteri speciali (wild card):

- `*` una qualunque stringa di zero o più caratteri in un nome di file
- `?` un qualunque carattere in un nome di file
- `[zfc]` un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. E' possibile esprimere intervalli di valori: `[a-d]`.
- `#` commento fino alla fine della linea.
- `\` escape: segnala di non interpretare il carattere successivo come speciale

Variabili

In ogni shell è possibile definire un insieme di variabili, trattate come stringhe, con nome e valore. I riferimenti ai valori delle variabili si fanno con il carattere

speciale `$` (`$nomevariabile`):

- `VAR` nome della variabile (l-value)
- `$VAR` valore della variabile VAR (r-value)

Assegnamento: `nomevariabile=valore` (senza spazi).

Ambiente di esecuzione e variabili di ambiente

Ogni comando esegue nell'ambiente (cioè l'insieme di variabili di ambiente definite) associato alla shell che esegue il comando. Ogni shell eredita l'ambiente dalla shell che l'ha creata.

Nell'ambiente ci sono variabili alle quali il comando può fare riferimento. Per vedere tutte le variabili di ambiente e i valori loro associati si può utilizzare il comando `set`.

Valutazione di espressioni: comando expr

Per valutare un'espressione: il comando expr stampa sullo standard output il risultato dell'espressione specificata mediante gli argomenti (expr prevede degli operatori, si veda il man).

Espressioni

`` (backtick o backquote) indicano alla shell di valutare la stringa racchiusa da ` come un comando: il comando tra backquote viene eseguito e sostituito con il suo output.

```
utente~$ echo risultato: ‘expr 5 + 1‘
risultato: 6
```

Shell scripting

E' possibile creare file (file comandi o shell script) che contengono comandi da eseguire in sequenza.

Prima dell'esecuzione:

- La shell prima eventualmente prepara e collega i comandi come filtri: ridirezione e piping di ingresso uscita

- Successivamente, se trova caratteri speciali, produce delle sostituzioni (passo di espansione)
    1. Sostituzione dei comandi comandi contenuti tra ` (backquote) sono eseguiti e sostituiti dall'output prodotto
    2. Sostituzione delle variabili e dei parametri nomi delle variabili ($nome) sono espansi nei valori corrispondenti
    3. Sostituzione dei metacaratteri * ? nei nomi di file secondo un meccanismo di pattern matching

    In alcuni casi è necessario privare i caratteri speciali del loro significato, considerandoli come caratteri normali.

    - \ (backslash) carattere successivo è considerato come un normale carattere
    - ' ' (apici singoli) proteggono da qualsiasi tipo di espansione
    - " " (doppi apici) proteggono dalle espansioni, con l'eccezione di $ \

I comandi contenuti in ogni file comandi vengono eseguiti in sequenza da una shell dedicata; per ogni comando, la shell crea un processo dedicato alla sua esecuzione.

Scelta della shell

La prima riga di un file comandi deve specificare quale shell si vuole utilizzare con la notazione:

```
#!<shell voluta>
Esempio:
#!/bin/bash
```

Assegnamento di variabili

```
var=value
```

Nessuno spazio prima e dopo il carattere "=".

Passaggio di argomenti

L’esecuzione di un file comandi può essere fatta nel seguente modo:

```
./nomefilecomandi arg1 arg2 ... argN
```

Gli argomenti sono variabili posizionali nella linea di invocazione contenute nell’ambiente della shell:

- $0 rappresenta il comando stesso
- $n rappresenta l’n-esimo argomento

Il comando shift fa scorrere tutti gli argomenti verso sinistra ($0 non va perso, solo gli altri sono spostati).

	$0	$1	$2
prima di shift	nomefilecom	-w	/usr/bin
dopo shift	nomefilecom	/usr/bin	

Il comando set riassegna gli argomenti $1 … $n.

Altre variabili notevoli

- `$*` insieme di tutte le variabili posizionali, che corrispondono ad argomenti del comando: $1, $2, ecc.
- `$#` numero di argomenti passati ($0 escluso!)
- `$$` id numerico del processo in esecuzione (pid)
- `$?` valore (int) restituito dall'ultimo comando eseguito. Ogni comando restituisce un valore di stato, che ne indica l’esito: successo o fallimento. Lo stato vale 0 in caso di successo e >0 in caso di errore.

Input e output

```
read var1 var2 var3
```

Le stringhe in ingresso vengono attribuite alle variabili secondo corrispondenza posizionale.

```
echo var1 contiene $var1 e var2 contiene $var2
```

La stringa specificata di seguito a echo, opportunamente espansa ($var.. vengono sostituiti con i relativi valori) viene scritta su stdout.

Valutazione espressione

Comando per la valutazione di una espressione

```
test expression
```

Restituisce uno stato:

- valore zero → true
- valore non-zero → false

In alternativa a test è possibile utilizzare le parentesi quadre singole:

```
[ expression ]
```

Gli spazi tra le parentesi e l'espressione sono significativi.

In alternativa è possibile utilizzare le parentesi quadre doppie:

```
[[ expression ]]
```

Vantaggi di quest'ultima alternativa:

- prestazioni
- si possono usare operatori logici && ||
- si possono usare > e < ma solo per confrontare stringhe (non vengono interpretati come redirezioni dell'I/O)
- si possono usare i metacaratteri per fare pattern-matching all'interno della condizione stessa.

  Esempio: [[$a = res*]] → restituisce 0 (=vero) se $a contiene una stringa che inizia per res seguito da zero o più caratteri.

[$a = res*] → avrebbe invece espanso res* con tutti i nomi di file nel dir corrente che fanno mach con "res" seguito da zero o più caratteri, dunque se ci sono più di 1 file che fanno match il programma da errore.

Regular expression

Per confrontare regular expressions complesse è opportuno utilizzare l'operatore =~.

Esempio:

```
[[ $a =~ ^[0-9]+$ ]]
```

Restituisce vero se $a contiene una stringa corrispondente a un intero positivo.

Strutture di controllo: if e case

```
if <condizioni>
then
    <comandi>
[elif <condizioni>
then
    <comandi>]
[else
    <comandi>]
fi
```

Alternativa:

```
case <var> in
    <pattern-1>)
        <comandi>;;
    ...
    <pattern-i> | <pattern-j> | <pattern-k>)
        <comandi>;;
    ...
    <pattern-n>)
```

```
        <comandi> ;;
esac
```

Nell'alternativa multipla si possono usare metacaratteri per fare pattern-matching.

Cicli enumerativi: for

```
for <var> [in <list>] #list=lista di stringhe
do
    <comandi>
done
```

Scrivendo solo for i si itera con valori di i in $*. Cioè: si itera sui parametri in ingresso $1 $2 $3 ...

Cicli non enumerativi: while e until

```
while <condizione o lista-comandi>
do
    <comandi>
done
```

```
until <condiz. o lista-comandi>
do
    <comandi>
done
```

Until è come while, ma inverte la condizione.

- [ ] Comandi specifici

▼ 6.0 - Programmazione concorrente

▼ 6.1 - Introduzione alla programmazione concorrente

Se la macchina concorrente è organizzata secondo il modello ad ambiente globale (o modello a memoria comune), un processo corrisponde a un thread, e la comunicazione tra processi avviene attraverso dati/risorse

condivisi. Vi è dunque la necessita` di sincronizzare gli accessi agli oggetti condivisi.

Ogni applicazione concorrente può essere rappresentata da un insieme di componenti, suddiviso in due sottoinsiemi disgiunti:

- processi (componenti attivi)
- risorse (componenti passivi)

  Qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito.

  Le risorse possono essere private o comuni, a seconda di quanti processi possono operare su di esse. Nel modello a memoria comune i processi interagiscono esclusivamente operando su risorse comuni.

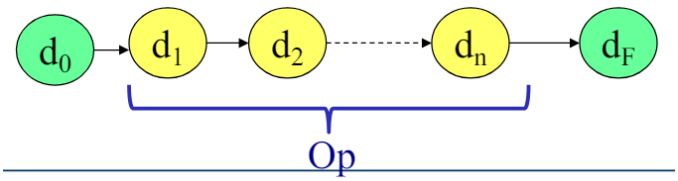
La mutua esclusione

La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo.

L'esecuzione contemporanea della stessa risorsa da parte di più processi può portare ad un uso scorretto di questa e/o ad errori.

Operazioni indivisibili

Data un'operazione Op(d), che opera su un dato comune a più processi, essa è indivisibile, se, durante la sua esecuzione da parte di un processo, il dato non è accessibile ad altri processi. Il processo può dunque operare una serie di trasformazioni sul valore del dato fino a giungere allo stato finale; siccome l'operazione è indivisibile gli stati intermedi non possono essere rilevati da altri processi concorrenti.



Sezione critica

La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di sezione critica.

La regola di mutua esclusione stabilisce che: Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

Deadlock

Quando due o piu processi competono per l'uso esclusivo di risorse comuni, è possibile incorrere in situazioni di stallo (deadlock, o blocco critico).

Un insieme di processi è in deadlock se ogni processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo dell'insieme.

Grafi di allocazione delle risorse

Per descrivere il deadlock, si utilizzano i grafi di allocazione delle risorse:

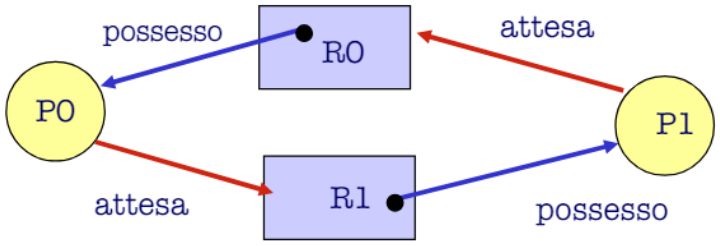
- processi (cerchi)
- risorse: (rettangoli)
- archi:
  - risorsa → processo: possesso della risorsa
  - processo → risorsa: attesa della risorsa

Condizioni per il deadlock

Dato un insieme di processi, essi si trovano in uno stato di deadlock se e solo se sono
verificate le seguenti 4 condizioni:

1. mutua esclusione: le risorse sono utilizzate in modo mutuamente esclusivo.
2. possesso e attesa: ogni processo che possiede una risorsa può richiederne un'altra.
3. impossibilità di prelazione: una volta assegnata ad un processo, una risorsa non può essere sottratta al processo.

4. attesa circolare: esiste un gruppo di processi {P0, P1..PN} in cui P0 attende una risorsa posseduta da P1, P1 attende una risorsa posseduta da P2,.. e PN attende una risorsa posseduta da P0.



Trattamento del deadlock

Le situazioni di blocco critico vanno evitate o risolte. Approcci:

- prevenzione: si evita a-priori il verificarsi di situazioni di blocco critico:
  - prevenzione statica: si impongono vincoli sulla struttura dei programmi, garantendo a priori che almeno una delle condizioni necessarie per il deadlock non sia verificata
  - prevenzione dinamica: la prevenzione è attuata in modo dinamico: quando un processo P richiede una risorsa R , il sistema operativo verifica se l'allocazione di R a P può portare a situazioni di blocco critico:
    - possibilità di deadlock (sequenza non salva): attesa del processo.
    - altrimenti (sequenza salva): allocazione della risorsa R a P.
- rilevazione/ripristino del deadlock: non c'è prevenzione, ma il S.O. rileva la presenza di deadlock → algoritmo di ripristino

Prevenzione statica

Si impone che almeno 1 delle 4 condizioni necessarie non sia verificata:

1. mutua esclusione: utilizzare risorse condivisibili (in certi casi non è possibile, dipende dalla natura della risorsa)
2. possesso e attesa: si impedisce ad ogni processo di possedere una risorsa mentre ne richiede un'altra.

3. preemption: possibilità di sottrarre la risorsa al processo (in certi casi non è possibile)
4. attesa circolare: si stabilisce un rigido ordinamento nell'acquisizione delle risorse da parte di ogni processo (es. ogni processo Pi non può acquisire Rj se è già in possesso di Rk, con k>j).

Prevenzione dinamica

Il sistema operativo può prevenire il deadlock, mediante specifici algoritmi; ad esempio, l'Algoritmo del Banchiere.

A tempo di esecuzione: quando un processo P richiede una risorsa R, il sistema operativo verifica se l'allocazione di R a P può portare a situazioni di blocco critico. Obiettivo: individuazione di una sequenza di processi salva, tra tutte le possibili sequenze di esecuzione. Analisi delle possibili sequenze di esecuzione: una sequenza di processi {P0, P1, .. Pn} è salva se per ogni processo Pi le richieste di risorse che Pi può ancora effettuare possono essere soddisfatte con le risorse attualmente libere, più le risorse allocate a tutti i processi Pj (j<i).

Soluzioni al problema della mutua esclusione

Una corretta soluzione al problema della mutua esclusione deve soddisfare i seguenti requisiti:

1. Sezioni critiche della stessa classe devono essere eseguite in modo mutuamente esclusivo.
2. Quando un processo si trova all'esterno di una sezione critica non può impedire l'esecuzione della stessa sezione (o di sezioni della stessa classe) ad altri processi.
3. Assenza di deadlock.

Schema generale

Prologo: ogni processo prima di entrare in una sezione critica esegue una serie di istruzioni che gli garantiscono l'accesso esclusivo alla risorsa, se questa è libera, oppure ne impediscono l'accesso se questa è già occupata.

Epilogo: al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la risorsa.

```
<prologo>
<sezione critica>
<epilogo>
```

Soluzioni possibili

1. Soluzioni Algoritmiche: la soluzione non richiede particolari meccanismi di sincronizzazione ma sfrutta solo la possibilità di condivisione di variabili.
2. Hardware-based: il supporto è fornito direttamente dall'architettura HW.
3. Soluzioni basate su strumenti di sincronizzazione: prologo ed epilogo sfruttano strumenti software per la sincronizzazione realizzati dal nucleo del sistema operativo che consentono l'effettiva sospensione dei processi in attesa di eseguire sezioni critiche.

Soluzioni algoritmiche

- L'algoritmo di Dekker è stata la prima soluzione corretta al problema della mutua esclusione.

```
int busy1 = 0;
int busy2 = 0;
int turno = 1; /*dominio {1,2}*/

/* processo P1: */
main() {
    busy1 = 1;
    while (busy2 == 1) {
        if (turno == 2) {
            busy1 = 0;
            while(turno != 1);
            busy1 = 1;
        }
```

```
    }
    <sezione critica A>;
    turno = 2;
    busy1 = 0;
}

/* processo P2: */
main() {
    busy2 = 1;
    while (busy1 == 1) {
        if (turno == 1) {
            busy2 = 0;
            while(turno != 2);
            busy2 = 1;
        }
    }
    <sezione critica B>;
    turno = 1;
    busy2 = 0;
}
```

- Un’altra soluzione corretta è quella dell’algoritmo di Peterson:

```
int busy1=0;
int busy2=0;
int turno=1; /*dominio {1,2}*/

/* processo P1: */
main() {
    busy1 = 1;
    turno = 2;
    while(busy2 && turno == 2);
    <sezione critica A>;
    busy1 = 0;
}
```

```
/* processo P2: */
main() {
    busy2 = 1;
    turno = 1;
    while(busy1 && turno == 1);
    <sezione critica B>;
    busy2 = 0;
}
```

Soluzioni hardware

Tutte le soluzioni algoritmiche realizzano l'attesa dei processi con cicli che si ripetono fino a che la condizione di ripetizione fallisce. Durante l'attesa ogni processo usa inutilmente la CPU: attesa attiva (busy waiting).

- Una possibile soluzione hardware che evita questo spreco è quella della disabilitazione delle interruzioni durante le sezioni critiche.

  ```
  /* struttura processo: */
  main() {
      <disabilitazione delle interruzioni>;
      <sezione critica A>;
      <abilitazione delle interruzioni>;
  }
  ```

  La soluzione è parziale in quanto è valida solo per sezioni critiche che operano sullo stesso processore e rende insensibile il sistema ad ogni stimolo esterno per tutta la durata di qualunque sezione critica.

- Molti processori prevedono istruzioni che consentono di esaminare e modificare il contenuto di una parola in un unico ciclo. L'istruzione test and set è un'istruzione macchina che consente la lettura e la modifica di una parola in memoria in modo indivisibile, cioè in un solo ciclo di memoria.

```
int test-and-set(int *a) {
    int R;
    R = *a;
    *a = 0;
    return R;
}
```

Tramite questa istruzione è possibile realizzare un meccanismo hardware-based (lock/unlock) per la soluzione del problema di mutua esclusione.

Sia x una variabile logica associata ad una classe di sezioni critiche inizializzata al valore 1 (libera): x = 0 risorsa occupata, x = 1 risorsa libera. Definiamo su x le seguenti operazioni atomiche:

```
void lock(int *x) {
    while (!test-and-set(x));
}

void unlock(int *x) {
    *x = 1;
}
```

La soluzione al problema della mutua esclusione mediante lock/unlock è dunque la seguente:

```
int x = 1;

/* processo P1: */
main() {
    lock(&x);
    <sezione critica A>;
    unlock(&x);
}

/* processo P2: */
```

```
main() {
    lock(&x);
    <sezione critica B>;
    unlock(&x);
}
```

I requisiti della mutua esclusione sono soddisfatti, vi è però attesa attiva vista il ciclo presente nella lock, la quale è però accettabile nel caso di sezioni critiche molto brevi. Si può applicare in ambiente multiprocessore.

Soluzioni software

La mutua esclusione si può ottenere attraverso strumenti software di sincronizzazione realizzati dalla macchina concorrente (es. kernel del SO) come ad esempio i semafori.

Un semaforo è un dato astratto rappresentato da un intero non negativo, al quale è possibile accedere soltanto tramite le due operazioni atomiche p e v, definite nel seguente modo:

- p(s)

  ```
  while (s == 0);
  s--;
  ```

  L'operazione p ritarda il processo fino a che il valore del semaforo s diventa maggiore di 0 e quindi decrementa tale valore di 1.

  Invocando l'operazione p() su un semaforo s:

  - Se il valore di s è 0: l'esecuzione di p provocherà l'attesa del processo che la invoca;
  - Se il valore di s è maggiore di 0: la chiamata di p provocherà il decremento di S e la continuazione dell'esecuzione.

- v(s)

  ```
  s++;
  ```

L’operazione v incrementa di 1 il valore del semaforo s.

Le due operazioni p e v sono atomiche (cioè, sezioni critiche). Il valore del semaforo viene modificato da un solo processo alla volta.

La mutua esclusione viene risolta utilizzando i semafori nel seguente modo:

```
/* mutex è un semaforo di «mutua esclusione»: il suo valor
e potrà assumere solo i valori {0, 1}.
Se mutex=0 -> risorsa occupata; se mutex=1 -> risorsa libe
semaphore mutex;
mutex.value = 1;

// P1
p(&mutex);
<Sezione critica 1>;
v(&mutex);

// P2
p(&mutex);
<Sezione critica 2>;
v(&mutex);
```

Per far sì che il semaforo gestisca più di due processi contemporaneamente è possibile realizzarlo nel seguente modo:

```
typedef struct{
    unsigned int value;
    queue Qs;
} semaphore;

void p(semaphore *s) {
    if (s.value == 0)
        <il processo viene sospeso ed il suo
        descrittore inserito in s.Qs>
    else s.value--;
```

```
}

void v(semaphore *s) {
    if (<s.Qs non e` vuota>)
        <il descrittore del primo processo viene rimosso
        dalla coda ed il suo stato modificato in pronto>
    else s.value++;
}
```

Per realizzare l'atomicità delle operazioni p e v si può fare uso di lock/unlock:

```
typedef struct{
    int value;
    queue Qs;
    int lock; // inizializzato a 1
} semaphore;

void p(semaphore *s) {
    lock(s.lock);
    if(s.value == 0){
        unlock(s.lock); //uscita sez. critica
        <sospensione processo in s.Qs>
        lock(s.lock); //entrata sez. critica
    } else s.value--;
    unlock(s.lock);
}

void v(semaphore *s) {
    lock(s.lock);
    if (<s.Qs non e` vuota>)
        <il descrittore del primo processo viene
        rimosso dalla coda ed il suo stato
        modificato in pronto>
    else s.value++;
```

```
    unlock(s.lock);
}
```

È possibile utilizzare semafori anche per la cooperazione tra processi concorrenti, ovvero lo scambio di messaggi generati da un processo e consumati da un altro o lo scambio di segnali temporali che indicano il verificarsi di dati eventi.

Per far rispettare un vincolo di precedenza (l'operazione A in P0 deve essere eseguita prima dell'operazione B in P1) è possibile utilizzare i semafori nel seguente modo:

```
semaphore si;
si.value = 0;

// P0
main() {
    <A (da eseguire prima..)>
    v(&si);
}

// P1
main() {
    p(&si);
    <B (..da eseguire dopo)>
}
```

Per la comunicazione tra due processi tramite un buffer occorre invece utilizzare due semafori al fine di controllare anche che il buffer non sia pieno/vuoto:

```
semaphore spazio_disp, msg_disp;
spazio_disp.value = n;
msg_disp.value = 0;

/* Processo produttore:*/
```

```
main() {
    <produzione messaggio>;
    p(&spazio_disp);
    <deposito messaggio>;
    v(&msg_disp);
}

/* Processo consumatore:*/
main() {
    p(&msg_disp);
    <prelievo messaggio>;
    v(&spazio_disp);
    <consumo messaggio>;
}
```

Nel caso di più produttori e più consumatori è necessario garantire che non più di un produttore alla volta acceda al buffer e che non più di un consumatore alla volta acceda al buffer. Per fare ciò si aggiungono due semafori, mutex1 e mutex2, i quali fanno rispettivamente i due controlli.

In generale i semafori sono uno strumento molto potente, in quanto è possibile adottarli per molte politiche di gestione delle risorse differenti (nelle slide sono presenti i casi di più processi che devono lavorare su più risorse e quello di più lettori e scrittori che devono operare su un'unica risorsa).

▼ 6.2 - Thread in Java

Thread

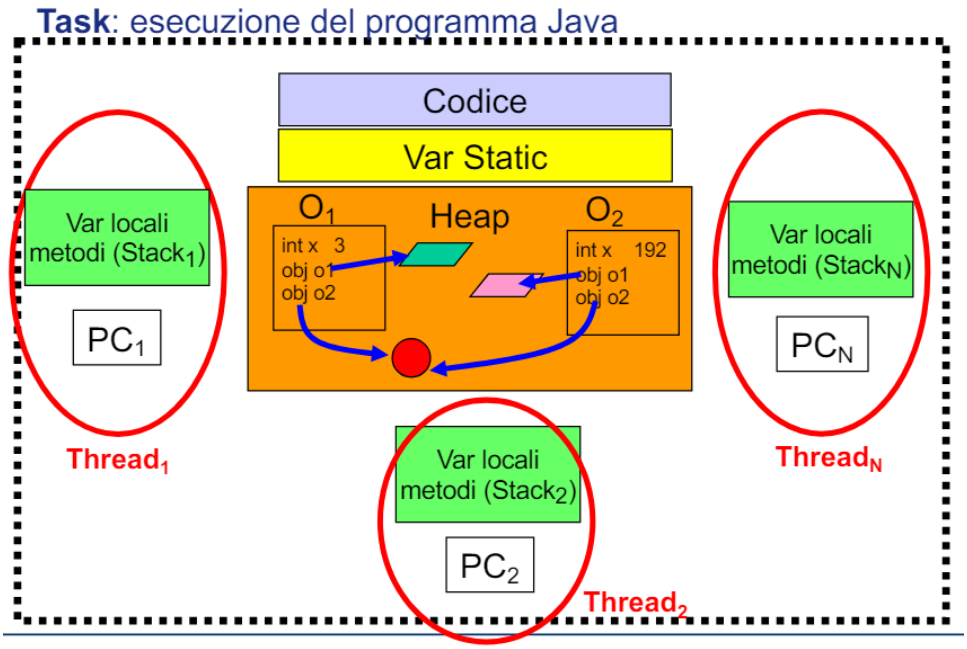
Un thread è un singolo flusso sequenziale di controllo all'interno di un processo (task). Un thread (o processo leggero) è un'unità di esecuzione che condivide codice e dati con altri thread ad esso associati.
Un thread:

- NON ha spazio di memoria riservato per dati e heap: tutti i thread appartenenti allo stesso processo condividono lo stesso spazio di indirizzamento

- Ha stack e program counter privati

Thread in Java

All'esecuzione di ogni programma Java corrisponde un task che contiene almeno un thread, corrispondente all'esecuzione del metodo main() sulla JVM. E' possibile creare dinamicamente ulteriori thread attivando concorrentemente le loro esecuzioni all'interno del programma.



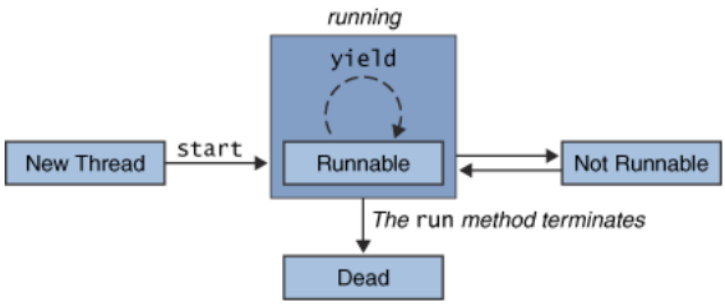
Thread come oggetti di sottoclassi della classe Thread

I thread sono oggetti che derivano dalla classe Thread.

In ogni sottoclasse derivata da Thread il metodo run deve essere ridefinito (override) specificando all'interno di esso cosa far eseguire ai thread di quella classe. Il metodo run() definisce l'insieme di istruzioni Java che ogni thread eseguirà.

Per creare un thread, si deve creare (tramite new) un'istanza della classe che lo definisce; dopo la new il thread esiste, ma non è ancora attivo. Per attivare un thread si deve invocare il metodo start().

Il ciclo di vita un di thread è il seguente:



- New Thread: subito dopo l'istruzione new, il costruttore alloca e inizializza le variabili di istanza.
- Runnable: il thread è eseguibile ma potrebbe non essere in esecuzione
- Not Runnable: il thread non può essere messo in esecuzione perchè è in attesa di
  un evento (attesa della terminazione di un'operazione di I/O oppure è bloccato a causa di sincronizzazione).
- Dead: il thread giunge a questo stato per "morte naturale" o perché un altro thread ha invocato il suo metodo stop().

Thread come classi che implementano Runnable

I thread sono oggetti che implementano l'interfaccia Runnable.

In ogni classe che implementa l'interfaccia Runnable il metodo run deve essere ridefinito (override).

Sincronizzazione di thread in java

Siccome differenti thread condividono lo stesso spazio di memoria (heap) occorre utilizzare algoritmi di sincronizzazione per far sì che più thread possano accedere contemporaneamente ad uno stesso oggetto senza che si verifichino problemi.

È possibile ad esempio sincronizzare il thread corrente con la terminazione di un altro thread utilizzando il metodo join():

```
T.join();
```

dove T è il thread del quale si attende la terminazione.

Mutua esclusione

Ad ogni oggetto Java associa un oggetto «object lock» che rappresenta lo stato di utilizzo dell'oggetto (libero/occupato). L'associazione del lock ad ogni oggetto viene fatta in modo
automatico e implicito dalla JVM.

E' possibile denotare alcune sezioni di codice che operano su un oggetto come sezioni critiche tramite la parola chiave synchronized.

Per ogni parte di codice synchronized il compilatore inserisce un prologo in testa alla sezione critica per l'acquisizione dell'object lock dell'oggetto (v.lock()) e un epilogo alla fine della sezione critica per rilasciare l'object lock (v.unlock()).

```
synchronized (oggetto x) {<sequenza di statement>;}
```

La keyword synchronized permette l'esecuzione del blocco mutuamente esclusiva rispetto ad altre esecuzioni dello stesso blocco e all'esecuzione di altri blocchi synchronized sullo stesso oggetto.

Semafori

La classe Semaphore consente di creare semafori, sui quali è possibile operare tramite i metodi:

- acquire(); // implementazione di p()
- release(); // implementazione di v()

Stop e suspend

- stop() forza la terminazione di un thread, tutte le risorse utilizzate vengono immediatamente liberate (lock compresi).

  Se il thread interrotto stava compiendo un insieme di operazioni da eseguirsi in maniera atomica, l'interruzione può condurre ad uno stato inconsistente del sistema. Per questo motivo il metodo stop() è "deprecated".

- suspend() blocca l'esecuzione di un thread, in attesa di una successiva invocazione di resume(). Non libera le risorse impegnate dal thread (non rilascia i lock)

Se il thread sospeso aveva acquisito una risorsa in maniera esclusiva (ad esempio sospeso durante l'esecuzione di un metodo synchronized), tale risorsa rimane bloccata. Per questo motivo il metodo suspend() è "deprecated".

Altri metodi di interesse per Java thread

- sleep(long ms): sospende thread per il # di ms specificato
- interrupt(): invia un evento che produce l'interruzione di un thread
- interrupted()/isInterrupted(): verificano se il thread corrente è stato interrotto
- isAlive(): true se thread è stato avviato e non è ancora terminato
- yield(): costringe il thread a cedere il controllo della CPU

▼ 6.3 - Il monitor

Il monitor è un costrutto sintattico che associa un insieme di operazioni ad una struttura dati comune a più processi, tale che le operazioni entry siano le sole operazioni permesse su quella struttura e tali operazioni siano mutuamente esclusive.

Struttura del monitor

La struttura generale del monitor è la seguente:

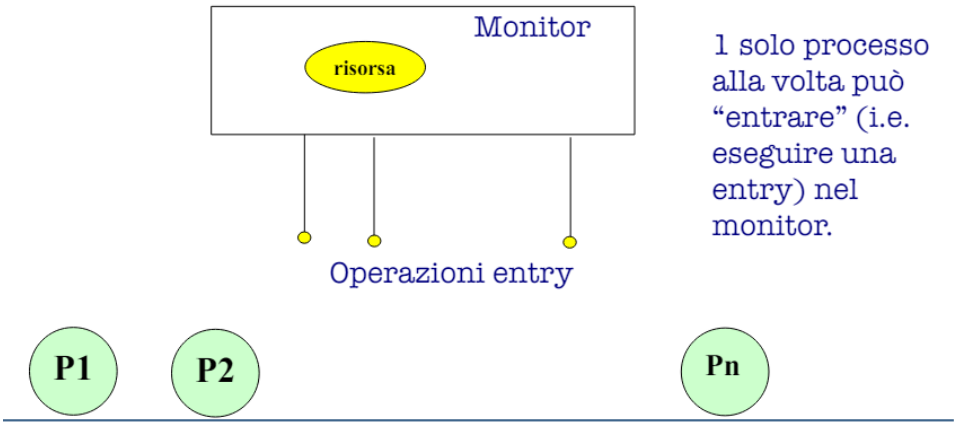
```
monitor tipo_risorsa {
    <dichiarazioni variabili locali>;
    <inizializzazione variabili locali>;

    entry void op1 ( ) {
        <corpo della operazione op1 >;
    }
    …
    entry void opn ( ) {
        <corpo della operazione opn>;
    }
```

```
    <eventuali operazioni non entry>
}
```

Le variabili locali sono accessibili solo all'interno del monitor.
Le operazioni entry (o public) sono le sole operazioni che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in modo mutuamente esclusivo.
Le operazioni non dichiarate entry non sono accessibili dall'esterno. Sono invocabili solo all'interno del monitor (dalle funzioni entry e da quelle non entry).

Uso del monitor

Solitamente, al monitor è associata una risorsa. Lo scopo del monitor è controllare l'accesso alla risorsa da parte processi concorrenti, in accordo a determinate politiche.



L'accesso alla risorsa avviene tramite il monitor, che garantisce due livelli di sincronizzazione:

1. Il primo garantisce che un solo processo alla volta possa aver accesso al monitor. Ciò è ottenuto mediante le operazioni entry che, per definizione, sono eseguite in mutua esclusione (eventuale sospensione dei processi nella coda entry queue).

   Tale livello di sincronizzazione viene realizzato direttamente dal linguaggio: ogni primitiva entry è sempre mutuamente esclusiva.

2. Il secondo controlla l'ordine con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica che assicura l'ordinamento. Se la condizione logica non è soddisfatta, il processo viene posto in attesa ed il monitor viene liberato.

   Questo livello di sincronizzazione viene realizzato dal programmatore in base alle politiche di accesso date sfruttando la variabile condizione, che vedremo in seguito.

Variabili condizione

Una variabile condizione rappresenta una coda nella quale i processi possono sospendersi (se la condizione di sincronizzazione non è verificata).

Definizione di una variabile di tipo condizione:

```
condition cond;
```

Su ogni variabile condizione è possibile richiamare i seguenti due metodi:

- wait(cond): sospende il processo, introducendolo nella coda individuata dalla variabile cond, e il monitor viene liberato.
- signal(cond): riattiva un processo in attesa nella coda individuata dalla variabile cond; se non vi sono processi in coda, non produce effetti.

Un esempio classico di utilizzo delle operazioni wait e signal in un monitor è il seguente:

```
monitor Risorsa() {
    boolean risorsa_libera = true;
    condition C;
    int turno = ..; // determina l'ordine di accesso

    entry void acquisizione(int id) {
        while (turno != id || risorsa_libera == false)
            C.wait();
```

```
        risorsa_libera = false;
    }

    entry void rilascio(int id) {
        risorsa_libera = true;
        <attribuzione nuovo valore a turno>
        C.signal();
    }
}
```

Il controllo nell'accesso al monitor viene esercitato tramite la sospensione dei processi in alcune code:

- Primo livello: se un processo che vuole accedere al monitor (tramite un'operazione entry) lo trova occupato, esso viene sospeso nella entry queue.
- Secondo livello: se la condizione di sincronizzazione di un processo che esegue nel monitor (tramite un'operazione entry) non è soddisfatta, esso viene sospeso nella condition associata alla condizione di sincronizzazione (condition queue).

Semantiche dell'operazione signal

Come conseguenza della signal, entrambi i processi (quello «segnalante» Q e quello «segnalato» P), possono proseguire la loro esecuzione. Il monitor, per definizione, limita a 1 il numero dei processi che possono eseguire al suo interno. A chi dare la precedenza?

Possibili strategie (o semantiche):

- signal_ and_ wait: P riprende immediatamente l'esecuzione ed il processo Q viene sospeso nella entry queue.

  Il primo processo ad operare nel monitor dopo la signal è certamente P.

- signal_ and_ continue: Q prosegue la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver riattivato il processo P, il quale si sospende nella entry queue.

Poiché altri processi possono entrare nel monitor prima di P, questi potrebbero modificare la condizione di sincronizzazione (lo stesso potrebbe fare Q). E' pertanto necessario che, al rientro nel monitor, il processo segnalato P verifichi di nuovo la condizione di sincronizzazione:

```
while(!B) wait(cond);
<accesso alla risorsa>
```

- signal_and_urgent_wait. E` una variante della signal_and_wait: Q ha la priorità rispetto agli altri processi che aspettano di entrare nel monitor. Viene quindi sospeso in una coda interna al monitor (urgent queue). Quando P ha terminato la sua esecuzione (o si è nuovamente sospeso), trasferisce il controllo a Q senza liberare il monitor.

E' possibile anche risvegliare tutti i processi sospesi sulla variabile condizione utilizzando la signal_all, che è una variante della signal_and_continue. Tutti i processi risvegliati vengono messi nella entry_queue dalla quale, uno alla volta potranno rientrare nel monitor.

La semantica della signal dipende dal linguaggio di programmazione utilizzato. Ad esempio il linguaggio Java implementa la variabile condizione con semantica signal&continue.

Realizzazione del costrutto monitor tramite semafori

Il kernel offre il semaforo come strumento "primitivo" di sincronizzazione. Per questo motivo, per un linguaggio concorrente che offre il costrutto monitor, il compilatore di tale linguaggio implementa il costrutto linguistico monitor mediante i semafori.

Per ogni istanza di un monitor il compilatore del linguaggio concorrente prevede:

- un semaforo mutex inizializzato a 1 per la mutua esclusione delle operazioni entry del monitor: la richiesta di un processo di eseguire un'operazione entry equivale all'esecuzione di una p(mutex). Alla fine di ogni operazione entry viene eseguita una v(mutex).

- per ogni variabile di tipo condition: un semaforo condsem inizializzato a 0 sul quale il processo si può sospendere tramite una p(condsem). un contatore condcount inizializzato a 0 per tenere conto dei processi sospesi su condsem.

Implementazione con semantica signal_and_continue

```
Prologo di ogni operazione entry: P(mutex);
Epilogo di ogni operazione entry: V(mutex);

wait(cond) {
    condcount++;
    V(mutex);
    P(condsem);
    P(mutex);
}

signal(cond) {
    if (condcount > 0) {
        condcount--;
        V(condsem);
    }
}
```

Implementazione con semantica signal_and_wait

```
Prologo di ogni operazione entry P(mutex);
Epilogo di ogni operazione entry V(mutex);

wait(cond) {
    condcount++;
    V(mutex);
    P(condsem);
}

signal(cond) {
```

```
    if (condcount > 0) {
        condcount-- ;
        V(condsem);
        P(mutex);
    }
}
```

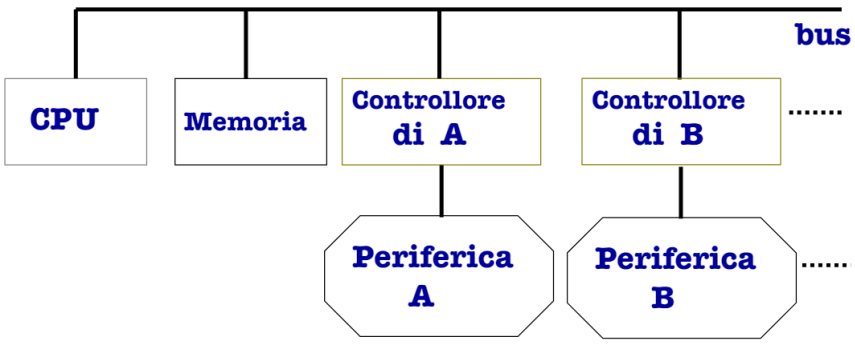
▼ **7.0 - Gestione dei dispositivi di I/O**

▼ **7.1 - Compiti del sottosistema di I/O**

Compiti del sottosistema di I/O

I compiti che il sottosistema di I/O deve rispettare sono:

1. Nascondere al programmatore i dettagli delle interfacce hardware dei dispositivi



2. Omogeneizzare la gestione di dispositivi diversi

   Esistono differenti tipologie di dispositivi:

   - Dispositivi a carattere (es. tastiera, stampante, mouse,...)
   - Dispositivi a blocchi (es. dischi, nastri, ..)
   - Dispositivi speciali (es. timer)

   Ogni dispositivo ha inoltre la sua velocità di trasferimento.

3. Gestire i malfunzionamenti che si possono verificare durante un trasferimento di dati

Possono presentarsi differenti tipologie di guasti:

- Guasti transitori (es. disturbi elettromagnetici durante un trasferimento dati)
- Guasti permanenti (es. rottura di una testina di lettura/scrittura di un disco)
- Eventi eccezionali (es. mancanza di carta sulla stampante, end-of-file)

4. Definire lo spazio dei nomi (naming) con cui vengono identificati i dispositivi

   Si tratta sia di definire nomi simbolici da parte dell'utente, che l'uso di indentificatori unici (es: valori numerici interi) all'interno del sistema per identificare in modo univoco i dispositivi.

5. Garantire la corretta sincronizzazione tra ogni processo applicativo che ha attivato un trasferimento dati e l'attività del dispositivo

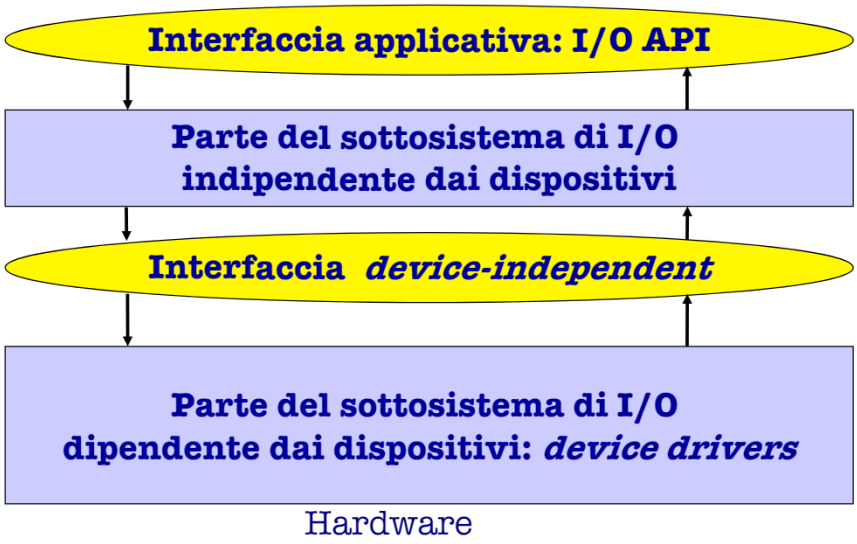
   Gestione sincrona dei trasferimenti: un processo applicativo attiva un dispositivo e si blocca fino al termine del trasferimento.

   Gestione asincrona dei trasferimenti: un processo applicativo attiva un dispositivo e prosegue senza bloccarsi.

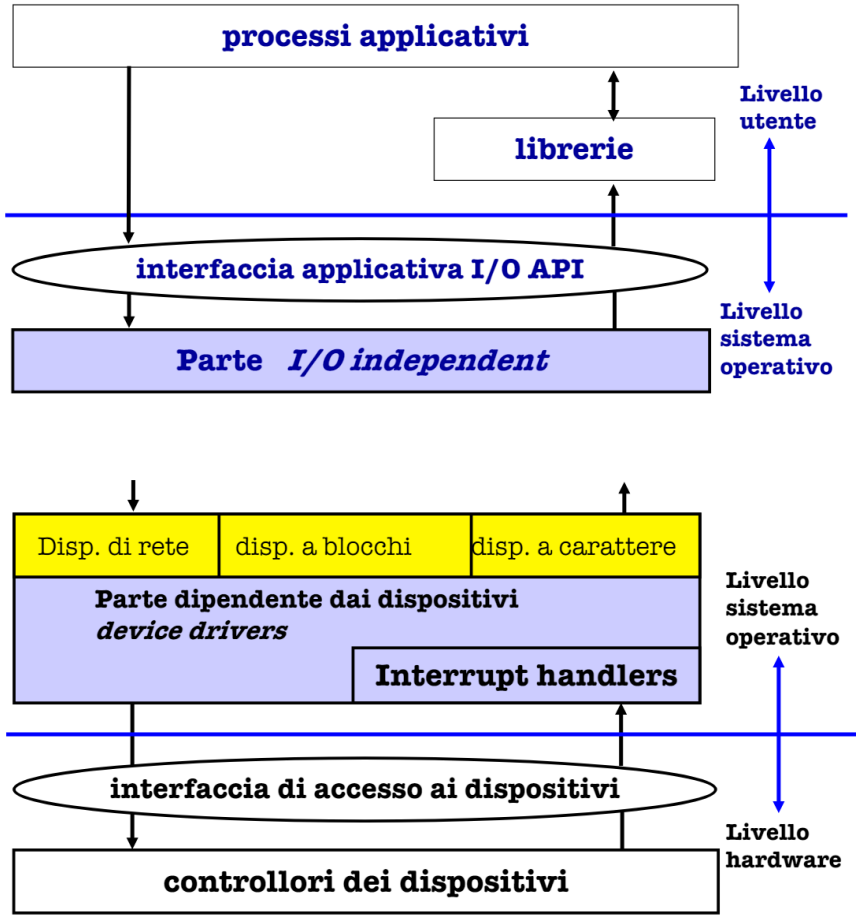
▼ **7.2 - Architettura del sottosistema di I/O**

Architettura del sottosistema di I/O

L'architettura generale del sottosistema di I/O è la seguente:



Analizzandola più nello specifico abbiamo:



Livello indipendente dai dispositivi

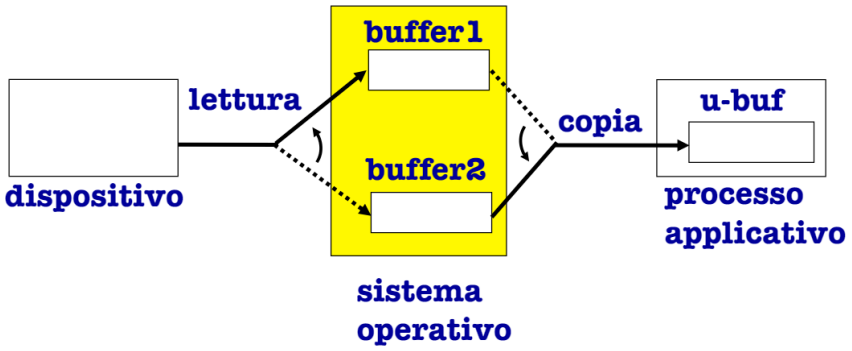
Tale livello svolge le seguenti funzioni:

- Naming
- Buffering

  Per ogni operazione di I/O il sistema operativo riserva un'area di memoria "tampone" (buffer), per contenere i dati oggetto del trasferimento.

  Un operazione di lettura con singolo buffer avviene nel seguente modo:

  È possibile utilizzare anche un doppio buffer, uno per la lettura dal dispositivo e uno per la copia dei dati nel buffer del processo applicativo. In questo modo si ottengono alcuni vantaggi, come quello di poter leggere dal dispositivo e copiare nel processo applicativo allo stesso tempo.



- Gestione malfunzionamenti

  I malfunzionamenti possono essere eventi generati a questo livello (es. tentativo di accesso a un dispositivo inesistente), oppure eventi

propagati dal livello inferiore (es. guasto HW temporaneo o permanente).

La gestione di tali eventi può consistere nella risoluzione diretta del problema mascherando l'evento anomalo, oppure nella gestione parziale e propagazione a livello applicativo.

- Allocazione dei dispositivi ai processi applicativi

  In base alla tipologia di allocazione si può ricadere nei seguenti tre casi:

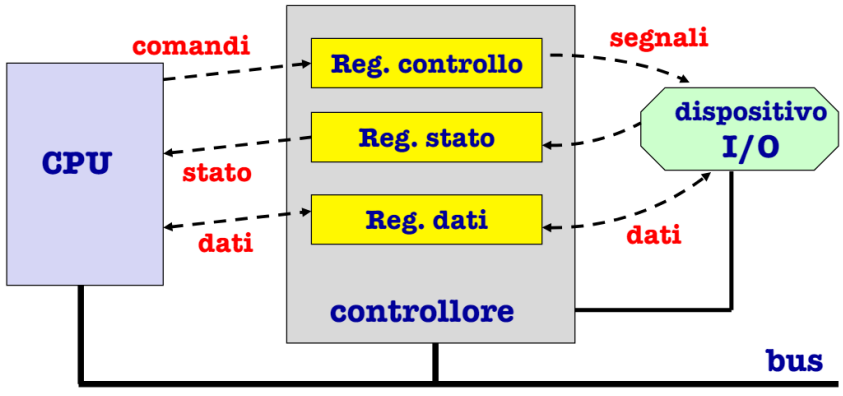
  - Dispositivi condivisi da più processi, da utilizzare in mutua esclusione
  - Dispositivi dedicati ad un solo processo (server) a cui i processi client possono inviare messaggi di richiesta di servizio
  - Tecniche di spooling (dispositivi virtuali)

Livello dipendente dai dispositivi

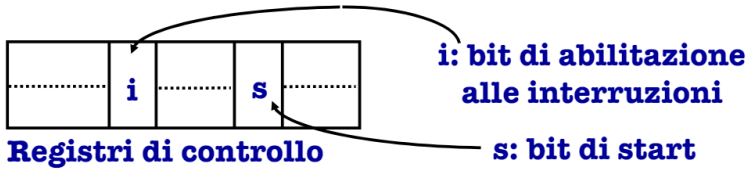
Tale livello svolge le seguenti funzioni:

- Fornire i gestori dei dispositivi (device drivers)

  Per l'interazione di dispositivi e CPU viene utilizzato un controllore I/O, il quale ha il seguente schema semplificato:

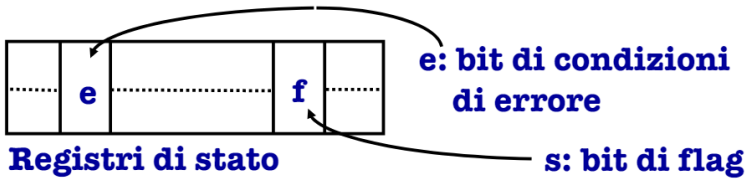


  - Registro di controllo: questo registro viene scritto dalla CPU per impartire comandi alla periferica.



La periferica viene attivata tramite il settaggio del bit di start.

- Registro di stato: la periferica usa il registro di stato per comunicare l'esito di ogni comando eseguito ed, eventualmente, per notificare eventuali errori occorsi.



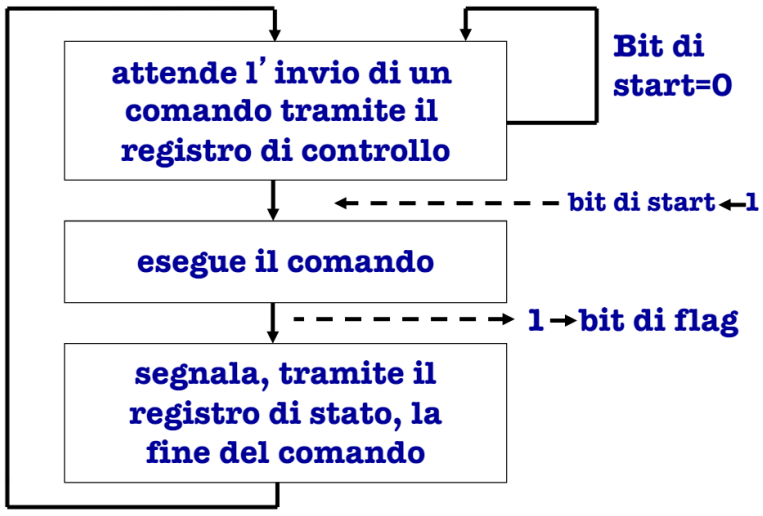
Il termine del comando viene notificato tramite il bit di flag.

- Registro Dati: viene utilizzato per il trasferimento dei dati letti/scritti dal dispositivo.

Nella gestione di ogni dispositivo, la sincronizzazione tra CPU e periferica può avvenire secondo 2 modelli alternativi:

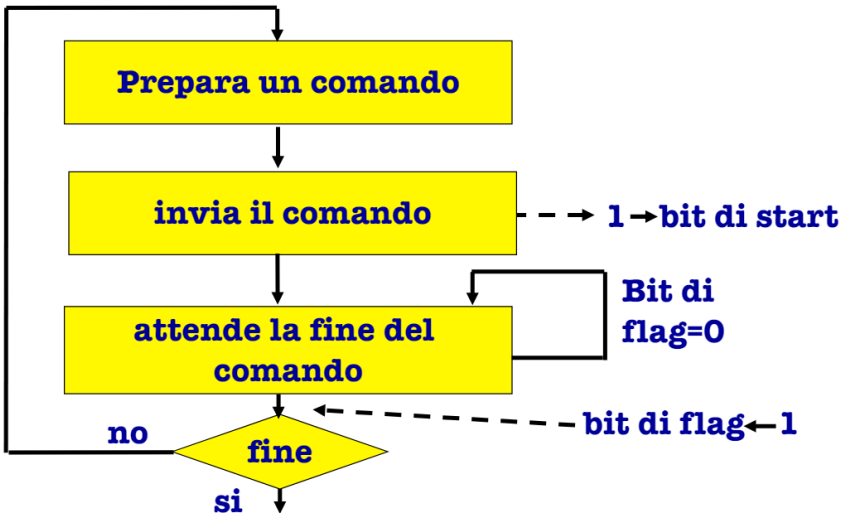
- Gestione a controllo di programma (o polling)

  In questo modello l'attività del dispositivo è del seguente tipo:



```
processo esterno { // descrive l'attività del dispos
    while (true) {
        do{;} while (start == 0) //stand-by
        <esegue il comando>;
        <registra l'esito del comando ponendo flag =
    }
}
```

L'attività del processo applicativo ha invece il seguente schema:




```
processo applicativo {
    for (int i = 0; i++; i<n) {
        <prepara il comando>;
        <invia il comando>;
        do{;} while (flag == 0)
            //ciclo di attesa attiva
        <verifica l'esito>;
    }
}
```

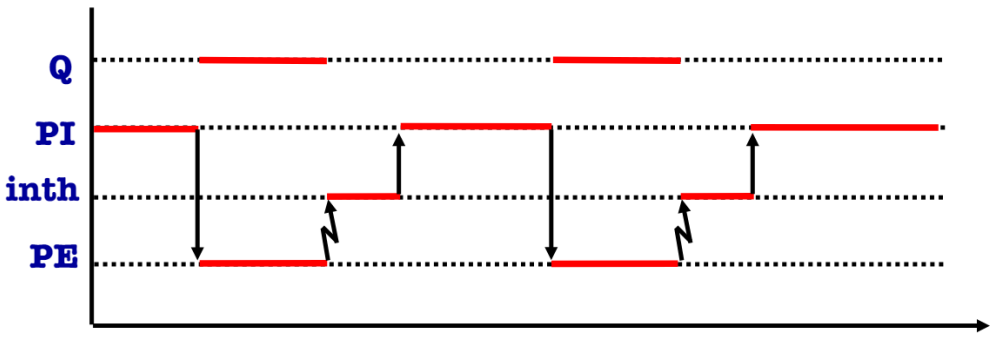
- Gestione basata su interruzioni

Lo schema “a controllo di programma” non è adatto per sistemi multiprogrammati, a causa dei cicli di attesa attiva.

È possibile evitare l’attesa attiva riservando per ogni dispositivo un semaforo e attivare il dispositivo abilitandolo a interrompere (ponendo nel registro di controllo il bit di abilitazione a 1).

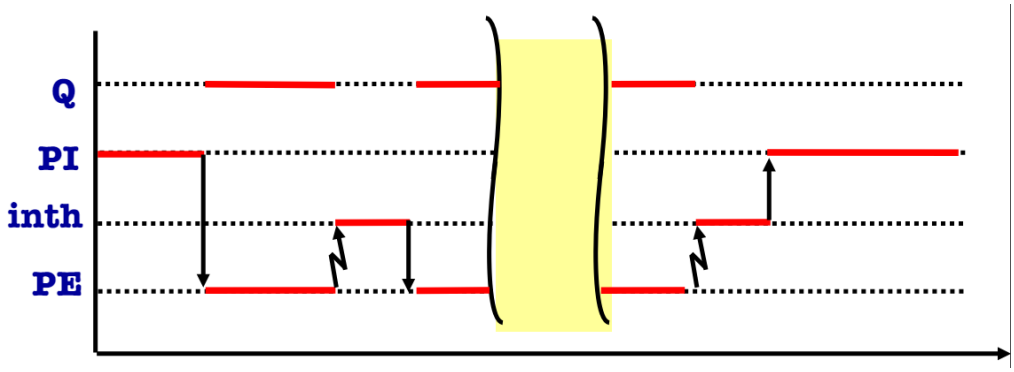
```
processo applicativo {
    for (int i = 0; i++; i<n) {
        <prepara il comando>;
        <invia il comando>;
        p(dato_disponibile);
        <verifica l’esito>;
    }
}

Interrupt_handler {
    v(dato_disponibile);
}
```

Il diagramma temporale di una gestione basata su interruzioni è del seguente tipo:



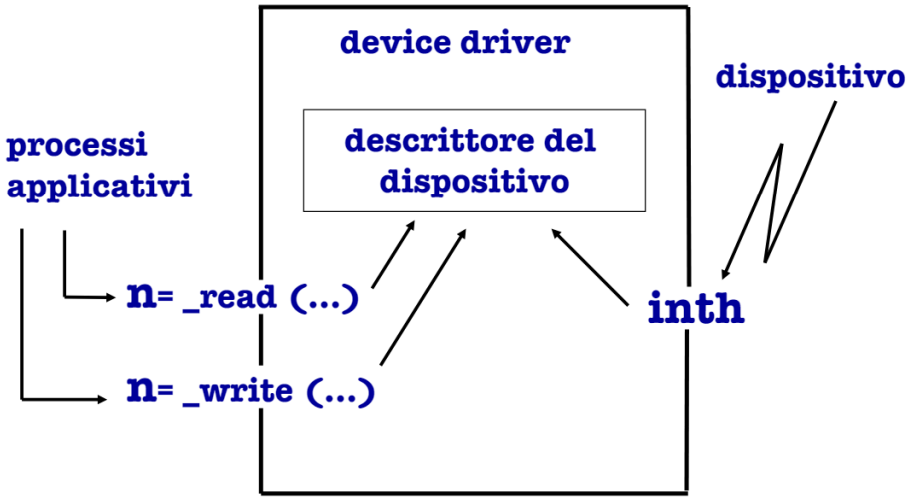
PI: processo applicativo (“interno”) che attiva il dispositivo
PE: processo esterno
Inth: routine di gestione interruzioni
Q: altri processi applicativi

Solitamente è preferibile uno schema in cui il processo applicativo (PI) che ha attivato un dispositivo per trasferire n dati venga risvegliato solo alla fine dell'intero trasferimento:



- Offrire al livello superiore l'insieme delle funzioni di accesso ai dispositivi (tramite un'interfaccia "device-independent").

  L'interfaccia devide-independent è caratterizzata dalla seguente astrazione.



Il descrittore del dispositivo è caratterizzato dal seguente contenuto:

Un esempio di funzione di accesso ad un dispositivo di lettura è la seguente:

```
int _read(int disp, char *buf, int cont)
```

dove disp è il nome unico del dispositivo, buf è l'indirizzo del buffer in memoria e cont è il numero di dati da leggere.

La funzione restituisce -1 in caso di errore o il numero di caratteri letti se tutto va bene.

Il contenuto della funzione _read è il seguente:

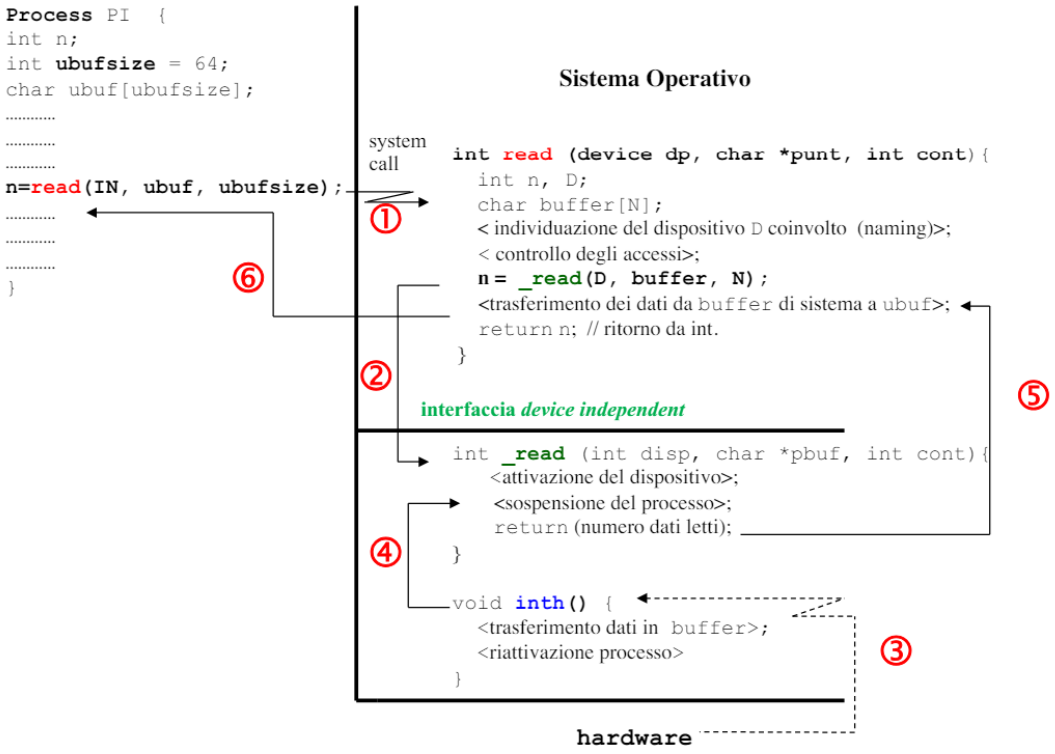
```
int _read(int disp,char *buf,int cont) {
    descrittore[disp].contatore = cont;
    descrittore[disp].puntatore = buf;
    <attivazione dispositivo>;
    p(descrittore[disp].dato_disponibile);
    if (descrittore[disp].esito == <cod.errore>) return
    return (cont-descrittore[disp].contatore);
}
```

L'interrupt handler è invece il seguente:

```
void inth() { // interrupt handler
    char b;
```

```
    <legge il valore del registro di stato>;
    if (<bit di errore> == 0) {
        <ramo normale della funzione>
    } else {
        <ramo eccezionale della funzione>
    }
    return //ritorno da interruzione
}

// ramo normale della funzione
{
    <b = registro dati>;
    *(descrittore[disp].puntatore) = b;
    descrittore[disp].puntatore++;
    descrittore[disp].contatore--;
    if (descrittore[disp].contatore != 0)
        <riattivazione dispositivo>;
    else {
        descrittore[disp].esito = <codice di terminazio
        <disattivazione dispositivo>;
        v(descrittore[disp].dato_disponibile);
    }
}

// ramo eccezionale della funzione
{
    <routine di gestione errore>;
    if (<errore non recuperabile>) {
        descrittore[disp].esito = <codice errore>;
        v(descrittore[disp].dato_disponibile);
    }
}
```

Il flusso di controllo durante un trasferimento è invece il seguente:



Un esempio di funzione utilizzabile su di un dispositivo di tipo temporizzatore è la delay. Un dispositivo temporizzatore (timer o clock) permette la modalità di servizio a divisione di tempo. Questo, il quale funzionamento consiste nel generare interruzioni periodiche a frequenze stabilite, consente tramite una gestione software di ottenere alcuni servizi come l'aggiornamento della data, la gestione del quanto di tempo nei sistemi time-sharing, valutazione dell'impegno della CPU di un processo, gestione della system call ALARM, gestione del time-out (delay).

Il funzione di una funzione delay è la seguente: il controllore del timer contiene, oltre ai registri di controllo e di stato, un registro contatore nel quale la CPU trasferisce un valore intero che viene decrementato dal timer. Quando il registro contatore raggiunge il valore zero il controllore lancia un segnale di interruzione. Nel descrittore della periferica timer sono presenti:

- un array di N semafori (fine_attesa[N]) inizializzati a zero. Ciascun semaforo viene utilizzato per bloccare il corrispondente processo che chiama la delay.

- un array di interi (ritardo[N]) utilizzato per mantenere aggiornato il numero di quanti di tempo che devono ancora passare prima che un processo possa essere riattivato.

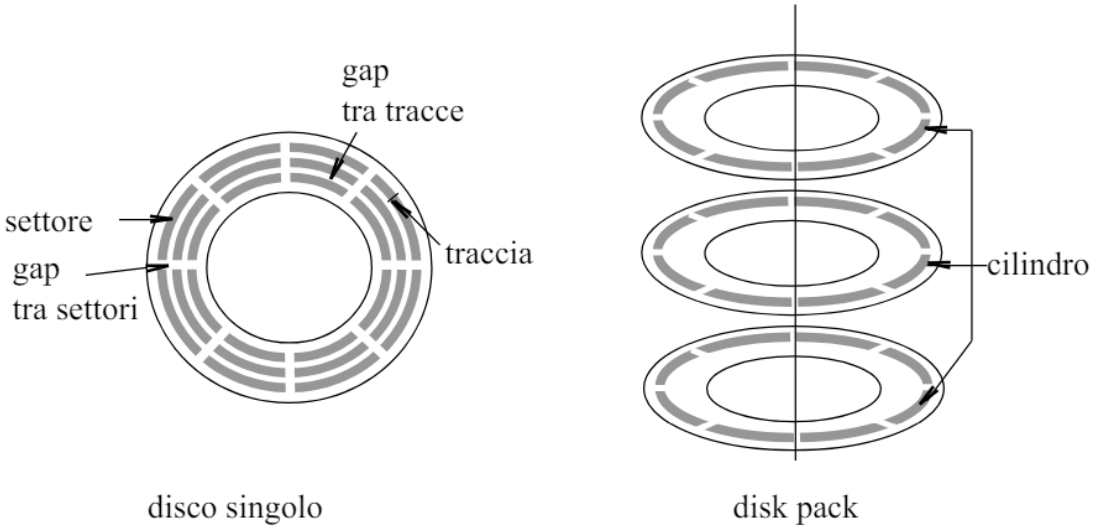
La struttura delle funzioni delay e dell'interrupt handler è la seguente:

```
void delay (int n) {
    int proc;
    proc = <indice del processo in esecuzione>;
    descrittore.ritardo[proc] = n;
    // sospensione del processo:
    descrittore.fine_attesa[proc].p();
}

void inth() {
    for(int i = 0; i < N, i++) {
        if (descrittore.ritardo[i]!=0) {
            descrittore.ritardo[i]--;
            if (descrittore.ritardo[i] == 0)
                descrittore.fine_attesa[i].v();
        }
    }
}
```

▼ **7.3 - Gestione e organizzazione dei dischi**

Gestione e organizzazione dei dischi

Organizzazione fisica



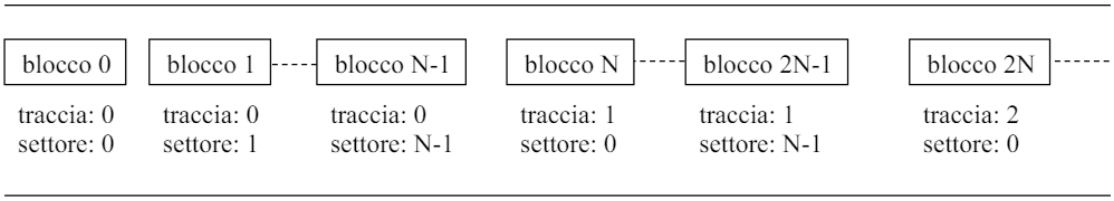
Indirizzo di un settore

L'indirizzo di un settore viene indicato con:

$$(f, t, s)$$

dove f è il numero della faccia, t è il numero della traccia e s è il numero del settore all'interno della faccia.

Tutti i blocchi che compongono un disco (o un pacco di dischi), possono essere trattati come un array lineare di blocchi.



Indicando dunque con M il numero di tracce per faccia e N il numero di settori per traccia otteniamo che un blocco di coordinate (f, t, s) viene rappresentato nell'ambito dell'array con l'indice:

$$i = f \cdot M \cdot N + t \cdot N + s$$

Scheduling delle richieste di trasferimento

Calcoliamo il tempo medio di trasferimento di un blocco:

$$TF = TA + TT$$

- TF: tempo medio di trasferimento di un blocco (per leggere o scrivere un blocco)
- TA: tempo medio di accesso (per posizionare la testina di lettura/scrittura all'inizio del blocco considerato)
- TT: tempo di trasferimento dei dati del blocco. Corrisponde al tempo necessario per far transitare sotto la testina l'intero blocco. Indicando con t il tempo necessario per compiere un giro, s il numero di settori per traccia, si ha:

$$TT = t/s$$

Possiamo inoltre scomporre TA in:

$$TA = ST + RL$$

- ST: tempo di seek (per posizionare la testina sopra la traccia contenente il blocco considerato)
- RL: rotational latency, ovvero il tempo necessario perché il settore, ruotando, si posizioni sotto la testina

Il tempo medio di trasferimento TF dipende sostanzialmente dal tempo medio di accesso TA. RL è un parametro che dipende dalle caratteristiche fisiche del dispositivo, mentre ST dipende da come il disco viene gestito. Per ridurre il tempo medio di trasferimento occorre dunque scegliere una modalità di gestione mirata alla riduzione del valore di ST.

Per fare ciò è possibile agire in due modi:

- Scegliere il giusto metodo di allocazione dei file, ovvero il modo con cui i blocchi sono memorizzati su disco.
- Scegliere la giusta politica di scheduling delle richieste, ovvero il criterio con cui servire le richieste di accesso.

Le richieste in coda ad un dispositivo possono essere servite secondo diverse politiche:

- First-Come-First-Served (FCFS)

  Le richieste sono servite rispettando il tempo di arrivo. Si evita il problema della starvation, ma non risponde ad alcun criterio di ottimalità.

- Shortest-Seek-Time-First (SSTF)

  Seleziona la richiesta con tempo di seek minimo a partire dalla posizione attuale della testina; può provocare situazioni di starvation

- SCAN algorithm

  La testina si porta ad una estremità del disco e si sposta verso l'altra estremità, servendo le richieste man mano che vengono raggiunte le tracce indicate, fino all'altra estremità del disco. Quindi viene invertita la direzione. → Minimizzazione del ST medio

- C-SCAN (Circular-SCAN)

  L'algoritmo SCAN non considera il tempo di attesa dei processi. Quando viene invertita la direzione per iniziare una nuova scansione, le richieste più "vecchie" riguardano tracce più lontane, che quindi verranno servite alla fine della scansione. CSCAN è una variante dello SCAN, nella quale l'insieme delle tracce viene scansionato sempre nella stessa direzione (non c'è inversione di direzione): arrivata all'ultima traccia, la testina ritorna immediatamente all'inizio del disco per una nuova scansione a partire dalla prima traccia. Rispetto allo SCAN, fornisce un tempo di attesa medio più basso.

Todo

- [ ] Creare chiavetta con file utili

Da sapere

Random

```
#include <time.h>

srand(time(NULL)); // inizializzazione generatore
x=rand()%MAX; // x è un numero compreso tra 0 e MAX - 1
```

```
Random r = new Random(System.currentTimeMillis());
r.nextInt(max) // da 0 (compreso) a max (non compreso)
```

Concatenare due stringhe in c

```
char *str1 = "str1";
char *str2 = "str2";

char str3[20];
strcpy(str3, str1);
strcat(str3, str2);
```

Lista dei segnali unix

```
kill -l
```

Risposte