SISTEMI OPERATIVI

Introduzione Sistemi Operativi

Cosa si intende per sistema operativo?

E' un programma che agisce come intermediario tra l'utente e l'hardware del computer:

- Fornisce un ambiente di sviluppo e di esecuzione per i programmi applicativi
- Fornisce una visione astratta dell'HW
- gestisce efficientemente le risorse del sistema di calcolo

Il sistema operativo puo' quindi essere visto come interfaccia tra cio' che offre l'hardware e il livello applicativo. Quello che deve fare il sistema operativo, tra gli altri compiti, e' quello di mappare le risorse hardware fisiche con delle risorse logiche, che offrono come funzionalita' la possibilita' di essere piu' facili da gestire e fornire a livello applicativo.
In particolare:

- La capacita' di processamento della CPU viene offerta tramite l'astrazione dei **processi**
- L'accesso ai dischi fisici viene fornito attraverso l'astrazione del **file system**
- L'accesso alla memoria, in particolare la memoria virtuale, viene offerta grazie al concetto di memoria virtuale.

Nel sistema operativo, l'utente e' l'utilizzatore dei programmi applicativi.
Possiamo quindi vedere l'architettura hardware e software di un elaboratore come un'architettura a strati in cui al centro troviamo l'hardware e le risorse fisiche, il sistema operativo che ah come compito di fornire una visione astratta e facilitata alle risorse dell'hardware, i programmi applicativi che grazie al sistema operativo e all'astrazione che offre sono in grado di accedere e utilizzare le risorse hardware e infine gli utenti che accedono ai programmi applicativi.

Un sistema operativo e' un programma che gestisce risorse del sistema di calcolo in modo corretto ed efficiente e le alloca ai programmi/utenti che ne necessitano. E' quindi un programma che ha come un risultato di innalzare il livello di astrazione con cui le risorse vengono sfruttate.

I sistemi operativo possono differire rispetto a molteplici aspetti:

- **Struttura**: come e' organizzato un SO (monoblocco o modulare);
- **Condivisione**: quali risorse vengono condivise tra utenti e programmi e in che modo;
- **Efficienza**: come massimizzare l'utilizzo delle risorse disponibili e anche che cosa massimizzare;
- **Affidabilita'**: come reagisce un SO a malfunzionamenti (SW/HW)
- **Estendibilita'**: e' possibile aggiungere funzionalita' al sistema?;

- **Protezione e sicurezza**: SO deve impedire interferenze tra programmi/utenti diversi;
- **Conformita' Standard**: portabilita', estendibilita', apertura.

Evoluzione SO

- Prima generazione (anni '50):
  - Linguaggio macchina
  - dati su schede perforate
- Seconda generazione ('55-'65):
  - sistemi batch semplici
  - linguaggio di alto livello (fortran)
  - input mediante schede perforate
  - aggregazione di programmi **lotti** (batch) con esigenze simili

Un sistema batch e' essenzialmente un insieme di programmi (job) che possono eseguire in modo sequenziale. L'esecuzione termina quando l'ultimo dei job e' arrivato a terminazione.
In questo caso il sistema operativo viene chiamato **monitor**, ovvero offre funzionalita' di trasferimento e controllo da un job all'altro. Si parla quindi di batch semplici.

Caratteristiche dei sistemi batch semplici:

- **SO residente in memoria (monitor)**;
- **assenza di interazione tra utente e job**;
- **scarsa efficienza**: durante l'I/O del job corrente, la CPU rimane inattiva (lentezza dei dispositivi I/O meccanici). Questo perche' in memoria centrale veniva caricato al piu' un solo job.

A causa della scarsa efficienza dei sistemi batch semplici e per migliorare l'utilizzo della CPU, e' stato introdotto il meccanismo di **Spooling (Simultaneous Peripheral Operation On Line)**. Tale meccanismo permetteva di caricare su disco i programmi e i dati quando la CPU era ancora in utilizzo per altri job, era quindi possibile sovrapporre l'uso della CPU per un job con la parte I/O dei job successivi.

Problemi:

- finche' il job corrente non e' terminato, il successivo non può iniziare l'esecuzione;
- se un job si sospende in attesa di un evento, la CPU rimane inattiva
- non c'e' iterazione con l'utente.

Sistemi batch multiprogrammati

Per ovviare a questi problemi si e' passati a sistemi batch multiprogrammati, in questo caso abbiamo sempre un pool di job che possono eseguire, ma in questo caso contemporaneamente e i job che possono eseguire sono tutti presenti su disco. In questo caso l'SO evolve e ha due compiti principali che nei sistemi batch semplici non aveva:

- SO seleziona un sottoinsieme di job appartenenti al pool da caricare in memoria centrale;
- mentre un job e' in attesa di un evento, il sistema operativo assegna CPU a un altro job. Si ha quindi una riduzione dei tempi di esecuzione dei job.

E' quindi l'SO in grado di portare avanti l'esevuzione di diversi job contemporaneamente, in ogni caso un solo job alla volta puo' usare la CPU e quindi e' in effettiva esecuzione. Mentre molteplici job possono essere pronti ad essere eseguiti e quindi attendono che il sistema operativo li assegni alla CPU.
Il sistema operativo non e' piu' un monitor come nei sistemi batch semplici ma si evolve verso un sistema operativo di tipo **scheduling**, ovvero ha il compito di scegliere quali dei job pronti per l'esecuzione deve essere effettivamente messo in esecuzione, e deve anche supportare tutte le funzionalita' necessare per fermare un job, selezionarne un altro e metterlo in memoria centrale.

Le scelte importanti che l'SO deve fare sono due:

- quali job caricare in memoria centrale: **scheduling del job** (long-term scheduling)
- a quale job assegnare la CPU: **scheduling della CPU** (short-term scheduling)

In memoria centrale, ad ogni istante, possono essere caricati piu' job. Dato che nella memoria centrale c'e' il sistema operativo, e' necessario che il sistema operativo assicuri che ciascun job possa intervenire solo nello spazio di memoria ad esso associato. Serve quindi protezione.

Sistemi time-sharing (Multics, 1965)

Nascono dalla necessita' di:

- **interattivita'** con l'utente, l'utente e' quindi in grado di interagire con i job tramite il sistema operativo, non solo all'inizio e alla fine del job, ma anche durante l'esecuzione del job;
  Per garantire un'accettabile velocita' di "reazione" alle richieste dei singoli utenti, SO **interrompe l'esecuzione** di ogni job dopo un intervallo di tempo prefissato (**quanto di tempo** o **time slice**), assegnango la CPU a un altro job.
- **multi-utenza**: piu' utenti interagiscono contemporaneamente con SO. Il sistema presenta ad ogni utente una **machina virtuale completamente dedicata** in termini di
  - utilizzo della CPU
  - utilizzo delle risorse, ad es. file system

I sistemi time-sharing sono sistemi in cui:

- attivita' della **CPU e' dedicata a job diversi** che si alternano **ciclicamente** nell'uso della risorsa

- frequenza di commutazione della CPU e' tale da fornire l'illusione ai vari utenti di una macchina completamente dedicata (**macchina virtuale**)

Cambio di contesto (context switch): operazione di trasferimento del controllo da un job al successivo --> costo aggiuntivo (overhead)

Requisiti del time-sharing:

- **Gestione/protezione** della memoria:
  - trasferimento memoria-disco
  - **separazione degli spazi** assegnati ai diversi job
  - molteplicita' job + limitatezza della memoria
  - a ogni job e' assegnata una cella di memoria virtuale (esso puo' accedere solo alla memoria a lui assegnata)
- **Scheduling CPU:** il sistema operativo deve poter schedulare l'assegnamento della CPU ai job anche in relazione al tempo di esecuzione di ciascun job
- **Sincronizzazione/comunicazione** tra job (job diversi possono avere necessita' di scambiarsi informazioni tra di loro o di accedere a risorse condivise):
  - interazione
  - prevenzione/trattamento di blocchi critici (**deadlock**)
- **Interattività'**: accesso on-line al file system per permettere agli utenti di accedere semplicemente a codice e dati

Esempi di SO attuali:

- **MSDOS**: monoprogrammato, monoutente
- **Windows 95/98**, molti SO attuali per dispositivi portabili (**Symbian, PalmOS**): multprogrammato (time sharing), tipicamente monoutente
- **Windows NT/2000/XP**: multiprogrammato, "multiutente"
- **MacOSX**: multiprogrammato, multiutente
- **UNIX/LINUX**: multiprogrammato, multiutente

Hardware di un sistema di elaborazione

Funzionamento a interruzioni:

- le varie componenti (HW e SW) sono in grado di interagire col sistema operativo tramite degli **interrupt (interruzioni asincrone)**

- ogni interruzione e' causata da un **evento**, ad es:
  - richiesta di servizio al SO
  - completamento di I/O
  - accesso non consentito alla memoria
- ad ogni interruzione e' associata una **routine di servizio (handler)** per la **gestione dell'evento**

Le interruzioni possono essere di due tipi:

- **Interruzioni Hardware**: dispositivi inviano segnali per richiedere l'esecuzione di servizi di SO
- **Interruzioni Software**: i programmi in esecuzione sono in grado di generare interruzioni SW
  - quando tentano l'esecuzione di **operazioni non lecite** (ad es. divisione per 0): **trap**
  - quando richiedono l'esecuzione di servizi al SO - **system call**

Gestione delle interruzioni

Alla ricezione di un'interruzione, SO:

1. interrompe la sua esecuzione --> **salvataggio dello stato** in memoria (locazione fissa, stack di sistema, ...)
2. attiva la **routine di servizio all'interruzione** (handler)
3. **ripristina lo stato** salvato

Per individuare la routine di servizio, SO puo' utilizzare un **vettore delle interruzioni**, vettore che metta in associazione ciascun possibile interrupt HW/SW con un'opportuna routine di servizio che sia in grado di gestire quel particolare interrupt.

Come avviene l'I/O in un sistema di elaborazione?

Avviene tramite dei controller, che fungono da interfaccia tra l'hardware e il bus d sistema che permette di mettere in comunicazione diverse periferiche, memoria centrale e CPU. Ogni controller e' dotato di:

- **un buffer** (con la funzione di memorizzare temporaneamente le informazioni da leggere o scrivere)
- alcuni **registri speciali**, ove memorizzare le specifiche delle operazioni di I/O da eseguire e in quale area di memoria leggere o scrivere

Quando un job richiede un'operazione di I/O (ad esempio, lettura da un dispositivo):

- CPU scrive nei registri speciali del dispositivo le specifiche dell'operazione da eseguire
- controller esamina i registri e provvede a trasferire i dati richiesti dal dispositivo al buffer
- invio di interrupt alla CPU (complemento del trasferimento)
- CPU esegue l'operazione di I/O tramite la routine di servizio (trasferimento dal buffer del controller alla memoria centrale)

2 tipi di I/O:

- **Sincrono**: il job viene sospeso finche' l'operazione di I/O non viene terminata
- **Asincrono**: il sistema restituisce immediatamente il controllo al job (molto piu' efficiente ma anche molto piu' complesso)
    - e' necessario predisporre delle funzionalita' di blocco in attesa di completamento dell'I/O
    - e' possibile avere piu' I/O pendenti
        - tabella di stato dei dispositivi

Direct Memory Access

Il trasferimento tra memoria e dispositivo viene effettuato direttamente, **senza intervento della CPU**, questo rende piu' efficiente il trasferimento di grandi quantita' di dati.

Introduzione di un dispositivo HW per controllare I/O: **DMA controller**.

- **driver di dispositivo**: componente del SO che
    - copia nei registri del DMA controller i dati relativi al trasferimento da effettuare
    - invia comando di richiesta al DMA controller
- **Interrupt** della CPU (inviato dal DMA controller) solo alla fine del trasferimento dispositivo --> memoria, usualmente di grandi quantita' di dati.

Protezione HW degli accessi a risorse

Nei sistemi che prevedono multiprogrammazione e multiutenza e' necessario adottare dei sistemi di protezione che si basano anche su meccanismi HW.
Dato che le risorse, l'accesso ai dispositivi, l'accesso alla memoria e l'accesso alla CPU sono condivisi tra i diversi job in esecuzione, e' necessario che l'accesso a tali risorse condivise sia mediato tramite opportuni meccanismi che ne impediscano l'accesso illecito da parte di programmi e utenti.
Ad esempio: accesso a locazioni esterne allo spazio di indirizzamento del programma.

Protezione della memoria

In un sistema **multiprogrammato** o **time-sharing**, ogni job ha un suo spazio di indirizzi:

- e' necessario impedire al programma in esecuzione di accedere al aree di memoria estere al proprio spazio (as esempio de SO oppure di altri job)
- se fosse consentito, un programma potrebbe modificare codice e dati di altri programmi, o ancor peggio, del SO

Per garantire protezione, molte architetture di CPU prevedono un duplice modo di funzionamento (**dual mode**)

- **user** mode

- **kernel** mode (supervisor, monitor code)

Realizzazione: l'architettura hardware della CPU prevede un bit di modo

- kernel: 0
- user: 1

Dual mode

Istruzioni privilegiate: sono quelle piu' pericolose e possono essere eseguite soltanto se il sistema si trova in **kernel mode**

- accesso a dispositivi I/O (dischi, schede di rete, ...)
- gestione della memoria (accesso a strutture dati di sistema per controllo e accesso alla memoria, ...)
- istruzioni di **shutdown** (arresto del sistema)
- ...

L'intero sistema operativo esegue in modalita' kernel in quanto deve poter accedere alle risorse hardware a disposizione.
Ogni programma utente invece esegue in **user** mode:

- quando un programma utente tenta l'esecuzione di una istruzione privilegiata, viene generato un **trap**
- se necessita di **operazioni privilegiate** --> chiamata a system call

La system call serve per ottenere l'esecuzione di istruzioni privilegiate, un programma i utente deve chiamare una system call:

- invio di un'interruzione software al SO
- salvataggio dello stato (PC, registri, bit di modo, ...) del programma chiamante e trasferimento del controllo a SO
- SO esegue in modo **kernel** l'operazione richiesta
- al termine dell'operazione, il controllo ritorna al programma chiamante (ritorno al modo **user**)

Introduzione all'organizzazione dei Sistemi Operativi

Le principali componenti di un SO sono:

- gestione dei processi
- gestione della memoria centrale
- gestione della memoria secondaria e file system
- gestione dell'I/O

- protezione e sicurezza
- interfaccia utente/programmatore

Processi

Processo = programma in esecuzione.
Il programma e' un'entita' passiva (un insieme di byte contenente le istruzioni che dovranno essere eseguite).
Il processo, invece, e' un entita' attiva, ovvero e' l'unita' di lavoro/esecuzione all'interno del sistema.
Ogni attivita' del SO e' rappresentata da un processo. Essenzialmente, un processo e' l'istanza di un programma che viene messo in esecuzione.

Processo = programma + contesto di esecuzione (PC, registri, ...)

Gestione dei Processi

In un sistema multiprogrammato piu' processo possono essere simultaneamente presenti nel sistema.
Il compito cruciale del SO e' quindi quello di:

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

Gestione della memoria centrale

Dal punto di vista hardware il sistema di elaborazione e' equipaggiato con **un unico spazio di memoria** accessibile direttamente da CPU e dispositivi.
il compito cruciale del SO e' quindi quello di:

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

Gestione dei dispositivi I/O

La gestione dei dispositivi I/O rappresenta una parte di SO:

- interfaccia tra programmi e dispositivi, ovvero e' il SO che interagisce interamente coi controller dei dispositivi e col DMA
- per ogni dispositivo esiste all'interno del SO un **device driver**
    - routine pr l'interazione con un particolare dispositivo

- contiene una conoscenza specifica sul dispositivo (ad es., routine di gestione delle interruzioni)

Gestione della memoria secondaria

Tra tutti i dispositivi, la memoria secondaria riveste un ruolo particolarmente importata es. gestione del file system, ma ha anche altre funzionalita':

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

Di solito la gestione dei file usa i meccanismi di gestione della memoria secondaria. La gestione della memoria secondaria e' indipendente dalla gestione dei file.

Gestione del file system

Ogni sistema di elaborazione dispone di uno o piu' dispositivi per la memorizzazione persistente delle informazioni (memoria secondaria).
Compito di SO --> Fornire una visione logica uniforme della memoria secondaria (indipendente dal tipo e dal numero dei dispositivi):

- realizzare il concetto astratto di **file**, come unita' di memorizzazione logica
- fornire una struttura astratta per l'organizzazione dei file (**direttorio)**

inoltre il sistema operativo si deve anche occupare di:

- creazione/cancellazione di file e directory
- manipolazione di file/directory
- associazione tra file e dispositivi di memorizzazione secondaria

Spesso file, directory e dispositivi I/O vengono presentati a utenti/programmi in modo uniforme

Protezione e sicurezza

In un sistema multiprogrammato, piu' entita' (processi o utenti) possono utilizzare le risorse del sistema contemporaneamente: necessita' di protezione.

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

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

Risorse da proteggere:

- memoria
- processi
- file
- dispositivi

Interfaccia utente

Il sistema operativo deve offrire la possibilita' agli utenti di interfacciarsi con il SO e quindi alle risorse che il SO gestisce

- **interpretazione comandi (shell)**: l'interazione avviene mediante una linea di comando
- **interfaccia grafica** (graphical user interface, **GUI**): l'interazione avviene mediante interazione mouse-elementi grafici su desktop; di solito e' organizzata in finestre.

Interfaccia programmatore

L'interfaccia del SO offre anche un'interfaccia per i programmatori, ovvero SO offre un insieme di system call con cui il programmatore puo' richiedere di interagire con le risorse condivise da parte del SO.

- mediante la system call il **processo richiede a SO** l'esecuzione di un servizio
- la system call esegue **istruzioni privilegiate**: passaggio da modo user a modo kernel

Classi di system call:

- gestione dei processi
- gestione di file e dispositivi (spesso trattati in modo omogeneo)
- gestione informazioni di sistema
- comunicazione/sincronizzazione tra processi

Programma di sistema = programma che chiama system call

Struttura e organizzazione di SO

Sistema operativo = insieme di componenti

- gestione dei processi
- gestione della memoria centrale
- gestione dei file
- gestione dell'I/O
- gestione della memoria secondaria
- protezione e sicurezza

- interfaccia utente/programmatore

Sulla base di questa osservazione bisogna chiedersi come e' possibile organizzare una struttura di un sistema operativo.
Esistono 3 macro approcci:

- struttura monolitica
- struttura modulare: stratificazione
- microkernel

SO monolitici

SO e' costituito da un unico modulo contenente un insieme di procedure, che realizzano le varie componenti. L'interazione tra le componenti avviene mediante il meccanismo di chiamata a procedura. (ne sono esempi, MS-DOS e le prime versioni di LINUX)

Principale vantaggio: basso costo di interazione tra le componenti;
Svantaggio: SO e' un sistema complesso e presenta gli stessi requisiti delle applicazioni **in-the-large**;
Soluzione: organizzazione modulare.

SO modulari

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

Storicamente, il primo sistema stratificato a livelli e' il sistema operativo THE di Dijkstra (1968) ed e' costituito da vari livelli sovrapposti. Ogni livello realizza un'insieme di funzionalita':

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

Il sistema operativo THE e' composto da 5 livelli

- livello 5: programmi di utente
- livello 4: buffering dei dispositivi di I/O
- livello 3: driver della console
- livello 2: gestione della memoria
- livello 1: scheduling della CPU
- livello 0: hardware

Vantaggi:

- Astrazione: ogni livello e' un oggetto astratto, che fornisce ai livelli superiori una visione astratta del sistema (macchina virtuale), limitta alle astrazioni presente nell'interfaccia

- Modularita': relazioni tra livelli sono chiaramente esplicitate dalle interfacce e possibilita' di sviluppo, verifica, modifica in modo indipendente dagli altri livelli.

Svantaggi:

- Organizzazione gerarchica tra le componenti: non sempre e' possibile --> difficolta' di realizzazione
- scarsa efficienza (costo di attraversamento dei livelli)

Soluzione --> limitare il numero dei livelli (struttura intermedia tra una struttura a 5 livelli e una struttura monolitica).
Questo avviene tramite la realizzazione di un sistema operativo con un nucleo (modo kernel), che e' la pare del sistema operativo che esegue in modo privilegiato. Quindi il nucleo (kernel) ha come compito di eseguire utte le operazioni privilegiate.

- E' la parte piu' interna di SO che si interfaccia direttamente con l'hardware della macchina
- Le funzioni realizzate all'interno del nucleo variano a seconda del particolare SO

Per un sistema multiprogrammato a divisione di tempo, il nucleo deve, almeno:

- gestire il salvataggio/ripristino dei contesti (context-switching)
- realizzare lo scheduling della CPU
- gestire le interruzioni
- realizzare il meccanismo di chiamata system call

SO a microkernel

La struttura del nucleo e' ridotta a poche funzionalita' di base:

- gestione della CPU
- gestione della memoria
- gestione dei meccanismi di comunicazione I/O

Il resto del SO e' mappato su processi utente

Caratteristiche:

- sono piu' affidabili (separazione tra componenti)
- possibilita' di estensioni e personalizzazioni
- scarsa efficienza (molte chiamate a system call)

MS-DOS

E' stato progettato per avere **minimo footprint**

- non diviso in moduli
- sebbene abbia una qualche struttura, interfacce e livelli di funzionalita' sono ben separati

UNIX

Dati i limiti delle risorse hardware del tempo, originariamente UNIX sceglie di avere una strutturazione limitata. Consiste di due parti separabili

- programmi di sistema
- kernel
  - costituito da tutto cio' che e' sotto l'interfaccia delle system-call interface e sopra hardware fisico
  - fornisce funzionalita' di file system, CPU scheduling, gestione memoria, ...; molte funzionalita' tutte allo stesso livello

Principi di progettazione e vantaggi:

- progetto snello, pulito e modulare
- scritto in linguaggio di alto livello (C)
- disponibilita' codice sorgente
- potenti primitive di SO su una piattaforma a basso prezzo
- progettato per essere time-sharing
- user interface semplice (Shell), anche sostituibile
- file system con direttori organizzati ad albero
- concetto unificante di file, come sequenza non strutturata di byte
- supporto semplice a processi multipli e concorrenza
- supporto ampio allo sviluppo di programmi applicativi e/o di sistema

02 - Shell Comandi

Shell

E' un programma che permette di far interagire l'utente (interfaccia testuale) con il sistema operativo tramite comandi ed e' uno strumento alternativo all'interfacia grafica. La shell e' un programma che resta sempre in attesa di un comando e viene mandato in esecuzione tramite la pressione del tasto <ENTER>.
La shell e' un interprete di comandi evoluto

- potente linguaggio di scriping
- interpreta ed esegue comandi da standard input o da file comandi

Esistono differenti tipi di shell: **Bourne shell** (standard), C shell, Korn shell, ...
L'implementazione della Bourne shell in Linux e' Bash (/bin/bash). Ogni utente puo' specificare la

propria shell preferita andando a modificare un file contenente le informazioni di tutti gli utenti del sistema che si trova nella directory /etc/passw.

La shell di login e' quella che richiede inizialmente i dati di accesso all'utente, per ogni utente connesso viene genereato un processo dedicato (che esegue la shell).

Accesso al sistema: login

Per accedere al sistema bisogna possedere una coppia **username e password.**
Il sistema operativo verifica le credenziali dell'utente e manda in esecuzione la sua shell di preferenza, posizionandolo in un direttorio di partenza (entrambe le informazioni si trovano in /etc/passw).
E' possibile cambiare la propria password di utente mediante il comando passwd. Se ci si dimentica della password, bisogna chiedere all'amministratore di sistema (utente *root*).

Uscita dal sistema: logout

Per uscire da una shell qualsiasi si puo' utilizzare il comando **exit** (che invoca la system call **exit()** per quel processo).
Per uscire dalla shell di login:

- logout
- CTRL+D (che corrisponde al carattere <EOF>)
- CTRL+C

Per rientrare nel sistema bisogna effettuare un nuovo login.

Esecuzione di un comando

Ogni comando richiede al SO l'esecuzione di una particolare azione. I comandi principali del sistema operativo si trovano all'interno della directory /bin. E' anche possibile realizzare nuovi comandi creando dei file di scripting all'interno del quale esistono una serie di comandi/righe di esecuzione e per ogni comando la shell genera un nuovo processo figlio (tramite l'esecuzione di una fork) e questo nuovo processo, separato dalla shell iniziale ha il compito di eseguire il comando richiesto. Alla fine dell'esecuzione del comando tale processo termina la sua esecuzione e viene quindi eliminato dal sistema operativo. Il processo padre ha due possibilita':

- attendere la terminazione del comando (foreground)
- proseguire in parallelo (background)

Comandi e input/output

I comandi UNIX si comportano come **filtri**. Un filtro e' un programma che riceve in ingresso da input e produce il risultato su uno o piu' output

Manuale

Per ogni comando offerto da Linux esiste un manuale on-line (man). il manuale indica:

- il formato del comando (input) e il risultato atteso (output)
- descrizione delle opzioni
- possibili restrizioni
- file di sistema interessati dal comando
- comandi correlati
- eventuali bug

Per uscire dal manuale basta digitare *q* (**quit** per editor di tipo vi)

Formato dei comandi

Tipicamente tutti i comandi Linux hanno questo formato: **nome -opzione argomenti** (es. ls -l temp.txt)

Convenzione nella rappresentazione della sintassi comandi:

- se un'opzione o un argomento possono essere omessi, si indicano tra quadre [opzione]
- se due opzioni/argomenti sono mutuamente esclusivi, vengono separati da | (arg1 | arg2)
- quando un argomento puo' essere ripetuto n volte, si aggiungono dei puntini (arg...)

Cenni pratici introduttivi all'utilizzo del file system Linux

File

Un file e' una risorsa logica costituita da una sequenza di bit a cui viene dato un nome. Tocca al sistema operativo associare il file (la nostra risorsa logica) a uno spazio su disco. Il file e' un'astrazione mlto potente che consente di trattare allo stesso modo entita' fisicamente diverse come file di testo, dischi rigidi, stampanti, directory, tastiera, video, ecc.

Esistono tre tipi di file:

- **Ordinari**: archivi di dati, comandi, programmi sorgente, eseguibili, ...
- **Directory**: gestiti direttamente solo da SO, contengono riferimenti a file (che possono essere strutturat in gerarchie)
- **Speciali**: dispositivi hardware, memoria centrale, hard disk, ecc. (essi sono gestiti come astrazione file, ma non vengono contati come file ordinari).

In aggiunta ci sono anche

- FIFO (pipe) - file per la comunicazione tra processi
- soft link - riferimenti (puntatori) ad altri file o directory

File: nomi

- E' possibile nominare un file con una qualsiasi sequenza di caratteri (max 255), a eccezione di '.' e '..'
- E' sconsigliabile utilizzare per il nomi di file dei caratteri speciali, ad es. metacaratteri e segni di punteggiatura
- Ad ogni file possono essere associati uno o piu' nomi simbolici (link) ma ad ogni file e' associato uno e un solo descrittore (i-node) identificato da un intero (i-number)

Directory

le directory sono gestite con una struttura a grafo diretto aciclico (DAG) e il punto di partenza e' sempre la radice (root) rappresentata da un singolo slash (/).

Gerarchie di directory

- All'atto del login, l'utente puo' cominciare a operare all'interno di una specifica directory (home). In seguito e' possibile cambiare directory
- E' possibile visualizzare il percorso completo attraverso il comando **pwd** (print working directory)
- Essendo i file organizzati in gerarchie di directory, SO metta disposizione dei comandi per muoversi all'interno di essi

Nomi relativi/assoluti

Ogni utente puo' specificare un file attraverso

- **nome relativo**: e' riferito alla posizione dell'utente nel file system (directory corrente)
- **nome assoluto**: e' riferito alla radice della gerarchia /

Nomi particolari:

- **.** : e' la directory corrente (visualizzato da pwd)
- **..** : e' la directory "padre"
- ~ : e' la propria home utente

Il comando **cd** permette di spostarsi all'interno del file system, utilizzando sia nomi relativi che assoluti. cd senza parametri riporta alla home dell'utente.

Link

Le informazioni contenute in uno stesso file possono essere visibili come file diversi, tramite "riferimenti" (link) allo stesso file fisico.

Il sistema operativo considera e gestisce la molteplicita' possibile di riferimenti:

- se un file viene cancellato, le informazioni sono veramente eliminate solo se non ci sono altri link a esso

- il link cambia diritti? --> meglio di no

Due tipi di link:

1. link fisici (si collegano alle strutture del file system)
2. link simbolici (si collegano solo ai nomi)

comando: ln [-s]

Gestione file: comando ls

Consente di visualizzare nomi di file. Usato all'interno di una directory elenca tutti i file presenti al suo interno.

- Varie opzioni: ls -l (per avere piu' informazioni sul file e quindi non solo il nome)
- possibilita' di usare metacaratteri (wildcard)
  - per es. se esistono i file f1, f2, f3, f4
    - ci si puo' riferire a essi scrivendo **f***
    - o piu' precisamente **f[1-4]**

Opzioni del comando **ls
***ls [-opzioni...] [file...]*

Alcune opzioni:

- **l** (long format): per ogni file una linea che contiene diritti, numero di link, proprietario del file, gruppo del proprietario, occupazione di disco (blocchi), data e ora dell'ultima modifica o dell'ultimo accesso e nome
- **t** (time): la lista e' ordinata per data dell'ultima modifica
- **u**: la lista e' ordinata per data dell'ultimo accesso
- **r** (reverse order): inverte l'ordine
- **a** (all files): fornisce una lista completa (normalmente i file il cui nome comincia con il punto non vengono visualizzati)
- **F** (classify): indica anche il tipo di file (eseguibile: *, directory:/, link simbolico: @, FIFO: |, socket: =, niente per i file regolari)

Comandi vari di gestione

- Creazione/gestione di di directory
  - **mkdir** <nomedir> creazione di un nuovo direttorio
  - **rmdir** <nomedir> cancellazione di un direttorio
  - **cd** <nomedir> cambio di direttorio
  - **pwd** stampa il direttorio corrente

    - **ls** [<nomedir>] visualizz. contenuto del direttorio
- Trattamento file
    - **ln** <vecchionome> <nuovonome> link
    - **cp** <filesorgente> <filedestinazione> copia
    - **mv** <vecchionome> <nuovonome> rinomina / sposta
    - **rm** <nomefile> cancellazione
    - **cat** <nomefile> visualizzazione

Comando ln, link

Serve per creare un novo link.
Le informazioni contenute in uno stesso file possono essere visibili come file diversi, tramite riferimenti (link) allo stesso file fisico.
Il sistema considera e tratta tutto: se un file viene cancellato, le informazioni sono veramente eliminate solo se non ci sono altri link a esso.

cat

Serve per visualizzare i contenuti all'interno di un file.

more (e less)

Il comando **cat** effettua una semplice stampa del contenuto a video di un file. Per una visualizzazione a pagine, certamente piu' comoda per file di larghe dimensioni, e' necessario utilizzare un comando di paginazione.
Il comando **more** permette di visualizzare un file una pagina per volta.
Esistono diverse alternative piu' recenti e potenti al comando more, per esempio il comando less (che permette anche di far scorrere la visualizzazione all'indietro).

echo

Il comando echo stampa a video la stringa passatagli come parametro.
Il comando echo rappresenta la primitiva fondamentale che la shell mette a disposizione per stampare informazioni a video.

sort

Il comando sort riordina le righe di un file in ordine alfabetico.
Il comando sort ha molte opzioni:

- **-o** <nomefileout> stampa su file
- **-n** interpreta le righe dei file come numeri
- **-k** <n> ordina il file secondo il contenuto della n-esima colonna

**
diff**

Stampa sullo standard output la differenza di contenuto fra i due file. Questo comando e' molto utilizzato nella gestione del codice sorgete.

wc

Il comando wc stampa il numero di righe, parole o caratteri contenuti in un file.
Esempi e opzioni:

- wc -l nome file.txt (conta le linee contenute nel file nome.txt)
- wc -w nomefile.txt (conta le parole contenute nel file nome.txt)
- wc -c nomefile.txt (conta i caratteri contenuti nel file nome.txt)

grep

Il comando grep seleziona le righe di un file che cntengono la stringa passata come paramatro e le stampa a video.
Esempi:

- grep stringa nomefile.txt (seleziona le righe del file nomefile.txt che contengono il testo "stringa" e le stampa a video)
- grep -c stringa nomefile.txt (conta le righe del file nomefile.txt che contengono il testo "stringa" e stampa il numero a video)
- grep -r stringa dir (seleziona le righe di tutti i file nella directory dir che contengono il testo "stringa" e le sta pa a video. ricorsivo sulle sotto directory)

Il comando grep ha numerosissime opzioni ed e' utilizzatissimo, specialmente nella gestione e manipolazione di file e di codice sorgente.

head

Il comando head mostra le prima righe di un file.
esempi:

- head -n 15 nomefile.txt (mostra le prime 15 righe del file)
- head -c 30 nomefile.txt (mostra i primi 30 caratteri del file)

tail

Il comando tail mostra le ultime righe di un file.
esempi:

- tail -n 15 nomefile.txt (mostra le ultime 15 righe del file)
- tail -c 30 nomefile.txt (mostra gli ultimi 30 caratteri del file)

time

Il comando time cronometra il tempo di esecuzione di un comando.

who

Il comando who mostra gli utenti attualmente collegati al sistema (ovverosia che hanno eseguito il login sulla macchina)

man

In ambiente UNIX, il comando man rappresenta l'help do sistema.
Esempio: man grep (mostra l'help del comando grep)
Oltre al comando man, le moderne distribuzioni di Linux mettono a disposizione anche il comando info.

ps

Permette di elencare tutti i processi all'interno del sistema operativo. Se il comando ha un certo nome, il processo relativo avra' lo stesso nome. Il comando ps e' molto utile quando si lancia l'esecuzione di programmi di sistema con errori di programmazione.

Alcune opzioni importanti:

- a -- mostra anche i processi degli altri utenti
- u -- fornisce il nome dell'utente che ha lanciato il processo e l'ora in cui il processo e' stato lanciato
- x -- mostra anche i processi senza terminale di constrollo (proesso demoni)

Attenzione: si ricordi di non usare il trattino (-) per specificare le opzioni del comando ps, in quanto ps adotta la sintassi del tipo "extended BSD" che non prevede l'uso di trattini.
Se si vuole un'aggiornamento periodico dello stato dei processi correnti, si usa il comando **top**.

top

Il comando top fornisce una visione dinamica in real-time del sistema corrente. Esso visualizza continuamente informazioni sull'utilizzo del sistema (memoria fisica e virtuale, CPU, ecc.) e sui processi che usano maggiore share di CPU.
Esiste una versione piu' avanzata e moderna di top, chiamata **htop**. Al contrario di top, di solito htop non e' inclusivo tra le utility di base disponibili sulle macchine Linux e va esplicitamente installato.

pgrep e pkill

Il comando pgrep restituisce il pid (process id) dei processi che corrispondono alla caratteristiche richieste (nome processo, utente, ...).
pkill rappresenta la stessa interfaccia dei comandi pgrep, ma anziche' stampare a video i pid di processi, li uccide (in realta' invia loro in segnale di SIGTERM)

Terminazione forzata di un processo

E' possibile terminare forzatamente un processo tramite il comando **kill**.
Ad esempio: **kill -9 <PID>** provoca l'invio di un segnale SIGKILL (forza la terminazione del processo che lo riceve e non puo' essere ignorato) al processo identificato da PID.

Segnali di interruzione

E' possibile interrompere un processo (purche' se ne abbiano i diritti) tramite il comando **kill -s <PID>**. Questo comando provoca un segnale (individuato dal parametro s) al processo identificato dal PID. Alcuni tra i segnali piu' comuni:

- CTRL-C -- invia un SIGINT, terminazione di un processo attualmente in foreground, kill -2
- CTRL-Z -- invia un SIGTSTP, sospensione di un processo, kill -20
- kill -l fornisce la lista dei segnali

Utenti e gruppi

- Sistema multiutente --> problemi di privacy e di possibili interferenze: necessita' di proteggere/nascondere informazione
- Concetto di gruppo (es. staff, utenti, studenti, ...)
- Ogni utente appartiene a un gruppo ma puo' far parte anche degli altri a seconda delle esgenze e coinfigurazioni
- Comandi relativi all'identita' dell'utente:
  - whoami
  - id

Protezione dei file

- Molti utenti
  - necessita' di regolare gli accessi alle informazioni
- Per un file, esistono 3 tipi di utilizzatori:
  - proprietario, **user**
  - gruppo del proprietario, **group**
  - tutti gli altri utenti, **others**
- Per ogni tipo di utilizzatore, si distinguono tre modi di accesso al file
  - lettura (r)
  - scrittura (w)
  - esecuzione (x) - per una directory significa list del contenuto
- Ogni file e' marcato con
  - User-ID e Group-ID del proprietario
  - 12 bit di protezione

SUID e SGID

SUID (Set User ID) e' un identificatore di utente effettivo. Si applica a un file di programma eseguibile solamente.
Se vale 1, fa si che l'utente che sta eseguendo quel programma venga considerato il proprietario di quel file (solo per la durata dell'esecuzione).
E' necessario per consentire operazioni di lettura/scrittura su file di sistema, che l'utente non avrebbe il diritto di leggere/modificare.

Protezione e diritti sui file

E' possibile cambiare i bit di protezione col comando:

- chmod [u g o] [+ -] [rwx] <nomefile>

I permessi possono essere concessi o negati solo dal proprietario del file.

Comandi, piping e ridirezione

Ridirezione di input e output

E' possibile ridirigere input e/o output di un comando facendo si che non si legga da stdin (e/o non si scriva su stdout) ma da file:

- Ridirezione dell'input:
  - comando < file_input (aperto in lettura)
- Ridirezione dell'output
  - comando > file_output (aperto in scrittura, nuovo o sovrascritto)
  - comando >> file_output (scrittura in append)

Piping

E' possibile utilizzare il piping per ridirigere l'output di un comando verso l'input di un altro comando.

- In DOS: realizzazione con file temporanei
- In UNIX:: pipe come costrutto parallelo (L'output del primo comando viene reso dispinibile al secondo e consumato appena possibile, non ci sono file temporanei)

Si realizza con il carattere speciale " | ".

Metacaratteri ed espansione

Metacaratteri

Shell riconosce caratteri speciali (**wild card**)

- * --> una qualunque stringa di zero o piu' caratteri in un nome di file

- ? --> un qualunque carattere in un nome file
- [zfc] --> un qualunque carattere, in un nome file, compreso tra quelli nell'insieme. Anche range di valori: **[a-d]**
- # --> commento fino a fine della linea
- \ --> escape (segnala di non interpretare il carattere successivo come speciale)

Variabili nella shell

In ogni shell e' possibile definire un insieme di variabili (trattate come stringhe) con nome e valore.

- i riferimenti a i valori delle variabili si fanno con il carattere speciale $ ($nomevariabile)
- si possono fare assegnamenti: nomevariabile=$nomevariabile

Ambiente di esecuzione

Ogni comando esegue nell'ambiente associato (insieme di variabili di ambiente definite) 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 puo' fare riferimento:
  - variabili con significato standard: PATH, USER, TERM, ...
  - variabili user-defined

Per vedere tutte le variabili di ambiente e i valori loro associati si puo' utilizzare il comando **set**.

Espressioni

Le variabili in shell sono stringhe. E' comunque possibile forzare l'interpretazione numerica di stringhe che contengono la codifica di valori numerici

- comando **expr
  **

Espansione

Prima dell'esecuzione, il comando viene scandito (parsing), alla ricerca di caratteri speciali.

- La shell prima prepara i comandi come filtri: ridirezione e piping di ingresso e uscita
- Nelle successive scansioni, se shell trova altri caratteri speciali, produce delle sostituzioni (passo di espansione)

Passi di sostituzione

Sequenza dei passi di sostituzione:

1. Sostituzione dei comandi

    - comandi contenuti tra ` ` (backquote) sono eseguiti e sostituiti dal risultato prodotto
2. Sostituzione delle variabili e dei parametri
    - nomi delle variabili ($nome) sono espansi nei valori corrispondenti
3. Sostituzione dei metacaratteri in nomi di file
    - metacaratteri * ? [ ] sono espansi nei nomi di file secondo un meccanismo di **pattern matching**

Inibizione dell'espansione

In alcuni case e' necessario privare i caratteri speciali del loro significato, considerandoli come caratteri normali.

- \ --> carattere successivo e' considerato come un normale carattere
- ' ' (apici) --> proteggono da qualsiasi tipo di espansione
- " " (doppi apici) --> proteggono dalle espansioni con l'eccezione di $ \ ` `

Comando cut

- Comando utile per la manipolazione di testo che consente di selezionare parti di una stringa (o di ciascuna riga di un file).
- cut permette di selezionare i caratteri della stringa che si trovano nelle posizioni specificate (opzione -c).
- cut puo' dividere una stringa in piu' campi (opzione -f) usando uno specifico delimitatore (opzione -d) e puo' selezionare i campi desiderati.

tr

- tr e' un comando che permette di fare semplici trasformazioni d caratteri all'interno di una stringa (o di ciascuna riga di un file), ad esempio sostituendo tutte le virgole con degli spazi
- tr esegue anche sostituzioni tra set di caratteri

seq

seq e' un comando che stampa sequenze di numeri. Puo' essere molto utile per la realizzazione di cicli negli script

find

find permette di cercare file all'interno del file system che soddisfano i requisiti richiesti dall'utente, ed eventualmente di manipolarli.

tee

- tee copia lo standard input sia nello standard output sia in un file passato come parametro

- e' molto utile quando si scrivono comandi con molte pipe, per monitorare il funzionamento di uno stadio della pipe

Script: realizzazione file comandi

La shell non e' unica. Nei moderni sistemi di Linux (e UNIX) sono disponibili diversi tipi di shell

- sh: Bourne Shell
- bash: Bourne Again Shell (versione avanzata di sh)
- zsh: Z Shell (versione molto avanzata di sh)
- ksh: Korn Shell
- csh: C Shell (sintassi simile al C)
- tcsh: Turbo C Shell (versione avanzata di csh)
- rush: Ruby Shell (Shell basata su Ruby)
- hotwire: propone un interessante e innovativo modello integrato di terminale e shell

La shell piu' usata e' sicuramente Bash, che e' molto simile alla Shell di Bourne (/bin/sh).

File Comandi

Shell e' un processore di comandi in grado di interpretare file sorgenti in formato testo e contenenti comandi --> file comandi (script).

Linguaggio di comandi (vero e proprio linguaggio di programmazione). Un file comandi comprende:

- statement per il controllo di flusso
- variabili
- passaggio dei parametri

Scelta della Shell

La prima riga di un file comandi deve specificare la hell che si vuole utilizzare:
es. #! /bin/bash

Passaggio di parametri

./nomefilecomandi arg1 arg2 ... argN

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

- $0 rappresenta il comando stesso
- $1 rappresenta il primo argomento

e' possibile far scorrere tutti gli argomenti verso sinistra con **shift**.

Altre informazioni utili

Oltre agli argomenti di invocazione del comando abbiamo:

- **$*** --> insieme di tutte le variabili posizionali, che corrispondono arg del comando ($1, $2, ecc.)
- **$#** --> numero degli argomenti passati ($0 escluso)
- **$?** --> valore (int) restituito dall'ultimo comando eseguito
- **$$** --> id numerico del processo in esecuzione (pid)

Forme di input/output

- read var1 var2 var3   (per l'input)
    - la stringa in ingresso viene attribuita alla/e variabile/i secondo la corrispondenza posizionale
- echo   var1 vale $var1 e   var2 vale $var2 (per l'output)

Strutture di controllo

Ogni comando in uscita restituisce un valore di stato che indica il suo completamento o fallimento. Tale valore e' posto nella variabile $?, se $? vale 0 il comando e' stato completato correttamente, se invece ha un valore positivo abbiamo un errore.

test

Comando per la valutazione di un espressione

- test -<opzioni> <nomefile>

Restituisce uno stato uguale o diverso da zero

- valore zero --> true
- valore non-zero --> false

Alcuni tipi di test:

- -f <nomefile> --> esistenza di file
- -d <nomefile> --> esistenza di direttori
- -r <nomefile> --> diritto di lettura su file (-w e -r)
- test <stringa1> = <stringa2> --> uguaglianza stringhe
- test <stringa1 != <stringa2> --> diversita' stringhe

Gli spazi intorno a = o a != SONO NECESSARI

03 - Processi e Thread

Concetto di processo

Il processo e' un programma di esecuzione. E' l'unita' di esecuzione all'interno del sistema operativo. Solitamente, esecuzione sequenziale (istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma).
Un SO multiprogrammato consente l'esecuzione concorrente di piu processi.

Programma = entita' passiva
Processo = entita' attiva

Il processo e' rappresentato da:

- codice (text) del programma eseguito
- dati: variabili globali
- program counter
- alcuni registri della CPU
- stack: parametri, variabili locali a funzioni/procedure

Inoltre, ad ogni processo possono essere associate delle risorse di SO. Ad esempio:

- file aperti
- connessioni di rete
- altri dispositivi di I/O in uso
- ...

Stati di un processo

Un processo, durante la sua esistenza puo' trovarsi in cari stati:

- **init**: stato transitorio durante il quale il processo viene caricato in memoria e SO inizializza i dati che lo rappresentano (stato iniziale)
- **Ready**: processo e' pronto per acquisire la CPU
- **Running**: processo che sta utilizzando la CPU
- **Waiting**: processo e' 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 di running
- piu' processi possono trovarsi negli stati ready e waiting

necessita' di strutture dati per mantenere in memoria le informazioni su processi in attesa di acquisire la CPU (ready) o di eventi (waiting). Per tutto questo e' necessario avere dei descrittori di processo (strutture dati che descrivono lo stato di ciascun processo).

Rappresentazione dei processi

- Ad ogni processo viene associata una struttura dati (descrittore): **Process Control Block (PCB)**
- PCB contiene tutte le informazioni relative al processo:
  - Stato del processo
  - Program Counter
  - Contenuto dei registri di CPU
  - Informazioni di scheduling
  - informazioni per gestore di memoria
  - Informazioni relative all'I/O
  - Informazioni di accouting
  - ...

Il sistema ha per ogni processo un PCB

Scheduling dei processi

E' l'attivita' mediante la quale un SO effettua delle scelte tra i processi, riguardo a:

- caricamento in memoria centrale
- assegnazione della CPU

In generale un sistema operativo compie tre diverse attivita' di scheduling

- scheduling a breve termine (o di CPU)
- scheduling a medio termine (o swapping)
- scheduling a lungo termine

Scheduler a lungo termine

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

- controlla il grado di multiprogrammazione (numero di processi contemporaneamente presenti nel sistema presenti nel sistema)
- e' una componente importante dei sistemi batch multiprogrammati
- nei sistemi time sharing

Interattivita': spesso e' l'utente che stabilisce direttamente il grado di multiprogrammazione --> scheduler a lungo termine non e' presente

Scheduler a medio termine

Nei sistemi operativi multiprogrammati:

- quantita' di memoria fisica puo' essere minore della somma delle dimensioni degli spazi logici di indirizzi da allocare a ciascun processo
- grado di multiprogrammazione non e', in generale, vincolato dalle esigenze di spazio dei processi

Swapping: trasferimento temporaneo in memoria secondaria di processi (o parti di processi), in modo da consentire l'esecuzione di altri processi

Scheduler a breve termine (o di CPU)

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

Nei sistemi time sharing, allo scadere di ogni quanto di tempo, SO:

- decide a quale processo assegnare la CPU (scheduling di CPU)
- effettua il cambio contesto (context switch)

Cambio di contesto

E' la fase in cui l'uso della CPU viene commutato da un processo ad un altro

Quando avviene un cambio di contesto tra un processo P_i ad un processo P_{i+1}:

- Salvataggio dello stato di P_i : SO copia Pc, registri, ... del processo deschedulato P_i nel suo PCB
- Ripristino dello stato di P_{i+1}: SO trasferisce i dati del processo P_{i+1} dal suo PCB nei registri di CPU, che puo' cosi' riprendere l'esecuzione

Il passaggio da un processo al successivo puo' richiedere onerosi trasferimenti da/verso la memoria secondaria, per allocare/deallocare gli spazi di indirizzi di processi.

Scheduler a breve termine

Lo scheduler a breve termine gestisce:

- la coda dei processi pronti: contiene i PCB dei processi che si trovano in stato Ready
- Altre strutture necessarie
  - 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

Scheduler

- **Scheduler short-term** scheduler viene invocato con alta frequenza (ms) --> deve essere molto efficiente

- **Scheduler medium-term** viene invocato a minore frequenza (sec-min) --> puo' essere anche piu' lento

Scelte ottimali di scheduling dipendono dalla tipologia di processi:

- **processi I/O-bound** - maggior parte del tempo di operazioni I/O, molti burst brevi di CPU
- **processi CPU-bound** - maggior parte del tempo in uso CPU, pochi burst CPU tipicamente molto lunghi

Code di Scheduling

Coda dei processi pronti (ready queue):

- strategia di gestione della ready queue dipende dalle politiche (Algoritmi) di scheduling adottate dal SO

Scheduling e cambio di contesto

Operazioni di scheduling determinano un costo computazionale aggiuntivo che dipende essenzialmente da:

- frequenza di cambio contesto
- dimensione PCB
- costo dei trasferimenti da/verso la memoria
    - esistono SO che prevedono processi leggeri (thread) che hanno la proprieta' di condividere codice e dati cona altri processi:
        - dimensione PCB ridotta
        - riduzione overhead

Operazione sui processi

Ogni SO multiprogrammato prvede dei meccanismi per la gestione dei processi.

Meccanismi necessari:

- creazione
- terminazione
- interazione tra processi

Sono operazioni privilegiate (esecuzione in modo kernel) --> definizione di system call

Creazione di processi

Un processo (padre) puo' chiedere la creazione di un nuovo processo (figlio). Da qui e' possibile realizzare gerarchie di processi

Relazione padre figlio

Vari aspetti/soluzioni:

- **concorrenza**
  - padre e figlio procedono in parallelo (es. UNIX), oppure
  - il padre si sospende in attesa della terminazione del figlio
- **condivisione delle risorse**
  - le risorse del padre (ad esempio, i file aperti) sono condivise con i figli (es. UNIX), oppure
  - il figlio utilizza le risorse soltanto se esplicitamente richieste da se stesso
- **spazio degli indirizzi**
  - duplicato: lo spazio degli indirizzi del figlio e' una copia di quello del padre (es. fork() in UNIX), oppure
  - differenziato: spazi degli indirizzi di padre e figlio con codice e dati diversi (es. VMS, stesso processo dopo exec() in UNIX)

Terminazione

Ogni processo:

- e' figlio di un altro processo
- puo' essere a sua volta padre di processi

SO deve mantenere le informazioni relative alle relazioni di parentela (nel descrittore: riferimento al padre)

Se un processo termina:

- il padre puo' rilevare il suo stato di terminazione
- i figli adottati da init (o da un processo ancestor, se richiesto esplicitamente tramite dunzione prctl)

Processi leggeri (thread)

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

Task = insieme di thread che riferiscono lo stesso codice e gli stessi dati

Thread = {PC, registri, stack, ...}
Task = {thread1, thread2, ..., threadN, text, dati}

Un processo pesante e' equivalente ad un task con un solo thread

Vantaggi dei thread

- **Condivisione memoria**: a differenza dei processi pesanti, un thread puo' condividere variabili con altri (appartenenti allo stesso task)
- **Miglior costo di context switch**: PCB di thread non contiene alcuna informazione relativa a codice e da cambio di contesto fra thread dello stesso task ha un costo notevolmente inferiore al caso dei processi pesanti
- **Minor protezione**: thread appartenenti allo stesso task possono modificare dati gestiti da altri thread

Realizzazione di thread

Alcuni SO offrono anche l'implementazione del concetto di thread (es. MSWinXP, Linux, Solaris)

Realizzazione

- A livello di kernel (MSWinXP, Linux, Solaris, MacOSX):
  - SO gestisce direttamente i cambi di contesto
    - tra thread dello stesso task (trasferimento di registri)
    - tra task
  - SO fornisce strumenti per la sicnronizzazione nell'accesso di thread a variabili comuni
- A livello utente (es. Andrew - Carnegie Mellon, POSIX pthread, vecchie versioni MSWin, Java thread):
  - il passaggio da un thread al successivo (nello steso task) richiede interruzioni al SOO (maggior rapidita')
  - SO vede processi pesanti: minor efficienza
    - es. sospensione di un thread
    - Cambio di contesto tra thread di task diversi
- Soluzioni miste
  - thread realizzati a entrambi i livelli contemporaneamente

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 e' influenzata da P2, e viceversa
- processi interagenti: P1 e P2 sono interagenti se l'esecuzione di P1 e' influenzata dall'esecuzione di P2, e/o viceversa

Processi interagenti

Tipi di interazione:

- **Cooperazione**: l'interazione consiste nello scambio di informazioni al fine di seguire un'attivita' comune
- **Competizione**: i processi interagiscono per sincronizzarsi nell'accesso a risorse comuni
- Interferenza: interazione non desiderata e potenzialmente deleteria tra i processi

Supporto all'interazione:
L'interazione puo' avvenire mediante

- **memoria condivisa** (modello ambiente globale): SO consente ai processi (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 e interagiscono mediante meccanismi di trasmissione/ricezione di messaggi; SO prevede dei meccanismi di supporto dello scambio di messaggi

Aspetti:

- concorrenza --> velocita'
- suddivisione dei compiti tra processi --> modularita'
- condivisione di informazioni
    - assenza di replicazione: ogni processo accede alle stesse istanze di dati
    - necessita' di sincronizzare i processi nell'access a dati condivisi

Processi cooperanti

Esempio: produttore & consumatore
Due processi accedono a un buffer condiviso di dimensione limitata:

- un processo svolge il ruolo di produttore di informazione che verranno prelevate dall'altro consumatore
- buffer rappresenta un deposito di informazioni condiviso

Produttore & consumatore

Necessita' di sincronizzare i processi:

- quando il buffer e' vuoto --> il cosumatore NON puo' prelevare messaggi
- quando il buffer e' pieno --> il produttore NON puo' depositare messaggi

04 - Processi UNIX

Processi UNIX

UNIX e' un sistema operativo multiprogrammato a divisione di tempo, l'unita' di computazione e' il processo.

Caratteristiche del processo UNIX:

- processo pesante con codice rientrante
  - dati non condivisi
  - codice condivisibile con altri processi
- funzionamento dual mode
  - processi di utente (**modo user**)
  - processi di sistema (**modo kernel**)

diverse potenzialita' e, in particolare, diversa visibilita' della memoria

Modello di processo UNIX

Ogni processo ha un proprio spazio di indirizzamento completamente locale e non condiviso --> Modello ad Ambiente Locale

Eccezioni:

- il codice puo' essere condiviso
- il file system rappresenta un ambiente condiviso

Stati di un processo UNIX

Come nel caso generale

- **Init**: caricamento in memoria del processo e inizializzazione delle strutture dati del SO
- **Ready**: processo pronto
- **Running**: processo usa la CPU
- **Sleeping**: processo e' sospeso in attesa di un evento
- **Terminated**: deallocazione del processo della memoria

In aggiunta

- **Zombie**: processo e' terminato, ma e' in attesa che il padre rilevi lo stato di terminazione
- **Swapped**: processo (o parte di esso) e' temporaneamente trasferito in memoria secondaria

Processi Swapped

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 piu' lunghi)
- da memoria secondaria a centrale: **swap in**
  - si applica preferibilmente ai processi piu' corti

Rappresentazione dei processi UNIX

Il codice dei processi e' rientrante --> piu' processi possono condividere lo stesso codice (text)

- codice e dati sono separati (modello a codice puro)
- SO gestisce una struttura di 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 e' swapped, riferimento alla memoria secondaria)
  - numero dei processi che lo condividono...

Process Control Block (PCB): il descrittore del processo UNIX e' rappresentato da 2 strutture dati

- **Process structure**: informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato del processo)
- **User structure**: informazioni necessarie solo se il processo e' residente in memoria centrale

Process Structure

Contiene, tra le altre, le seguenti informazioni:

- **processo identifier** (**PID**): intero positivo che individua univocamente il processo
- stato del processo
- puntatori alle varie aree dati e stack associati al processo
- riferimento indiretto al codice: la processo structure contiene il riferimento all'elemento della text table associato al codice del processo
- informazioni di scheduling (es: priorita', tempo di CPU, ...)

- riferimento al processo padre (PID del padre)
- info relative alla gestione di segnali (segnali inviati ma non ancora gestiti, maschere)
- puntatori al processo successivo in code di scheduling (ad esempio, ready queue)
- puntatore alla user structure

User structure

Contiene le informazioni necessarie al SO per la gestione del processo, quando e' residente:

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

Immagine di un processo UNIX

Immagine di un processo e' insieme aree di memoria e strutture dati associate al processo

- Non tutta l'immagine e' accessibile in modo user:
  - parte di **kernel**
  - parte di **utente**
- Ogni processo puo' essere soggetto a swapping: non tutta l'immagine puo' essere trasferita in memoria
  - parte **swappable**
  - parte residente o **non swappable**

Componenti

- **process** structure: e' l'elemento della process table associato al processo (kernel, residente)
- **text**: elemento della text table associato al codice del processo (kernel, residente)
- area **dati globali di utente**: contiene le variabili globali del programma eseguito dal processo (user, swappable)
- **stack, heap** di utente: aree dinamiche associate al programma eseguito (user, swappable)
- **stack del kernel**: stack di sistema associato al processo per le chiamate a system call (kernel, swappable)
- **user structure**: struttura dati contenente i dati necessari al kernel per la gestione del processo quando e' residente (kernel, swappable)

PCB = process structure + user structure

- **Process structure (residente)** : mantiene le informazioni necessarie per la gestione del processo, anche se questo e' swapped in memoria secondaria

- **User structure**: il suo contenuto e' necessario solo in caso del processo (stato running); se il processo e' soggetto a swapping, anche la user structure puo' essere trasferita in memoria secondaria

System call per la gestione di processi

Chiamate di sistema per

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

Creazione di processi: fork()

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

fork() non richiede parametri, restituisce un intero che:

- per il processo creato vale 0
- per il processo padre e' un valore positivo che rappresenta il PID del processo figlio
- e' un valore negativo in caso di errore (la creazione non e' andata a buon fine)

Effetti della fork()

- Allocazione di una nuova process structure nella process table associata al processo figlio e alla 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 i dati e stack del padre
- Aggiornamento del riferimento text al codice eseguito (condiviso col padre): incremento del contatore dei processi, ...

Relazione padre-figlio in UNIX

Dopo una fork():

- **concorrenza**
  - padre e figlio procedono in parallelo
- **lo spazio degli indirizzi e' duplicato**
  - ogni variabile del figlio e' inizializzata con il valore assegnatole dal padre prima della fork()
- la **user structure** e' duplicata

    - le risorse allocare al padre (ad esempio, i file aperti) prima della generazione sono condivise coi figli
    - le informazioni per la gestione dei segnali sono le stesse per padre e figlio (associazioni segnali-handler)
    - il figlio nasce con lo stesso program counter del padre: la prima istruzione eseguita dal figlio e' quella che esegue immediatamente fork()

Terminazione di processi - exit()

Un processo puo' terminare:

- involontariamente
    - tentativi di azioni illegali
    - interruzione mediante segnale
    - --> salvataggio dell'immagine nel file core
- volontariamente
    - chiamata alla funzione exit()
    - esecuzione dell'ultima istruzione

La funzione exit() prevede un parametro (status) mediante il quale il processo che termina puo' comunicare al padre informazioni sul suo stato di terminazione (ad esempio esito dell'esecuzione). E' sempre una chiamata senza ritorno.

Effetti della exit():

- chiusura dei file aperti non condivisi
- termianzione 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(), il processo passa nello stato zombie

wait()

Lo stato di terminazione puo' essere rilevato dal processo padre, mediamente la system call wait()

int wait(int *status)

- parametro status e' l'indiruzzo della variabile in cui viene memorizzato lo stato di terminazione del figlio
- risultato del prodotto wait() e' pid del processo terminato, oppure un codice di errore (<0)

Effetti della system call wait(&status)

Il processo che invoca la wait() puo' avere figli in esecuzione:

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

wait(): rilevazione dello stato

In caso di terminazione di un figlio, la variabile status raccoglie stato di terminazione; nell'ipotesi che lo stato sia un intero a 16 bit:

- se il byte meno significativo di status e' zero, il piu' significativo rappresenta lo stato di terminazione (terminazione volontaria, ad esempio con exit)
- in caso contrario, il byte meno significativo di status descrive il segnale che ha terminato il figlio (terminazione involontaria)

wait(): status

E' necessario conoscere la rappresentazione di status

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

System call exec()

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

Effetto principale di system call famiglia exec:

- vengono sostituiti 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

Effetti dell'exec()

Il processo dopo exec()

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

Inizializzazione dei processi UNIX

- init genera un processo per ogni terminale (tty) collegato --> comando **getty**
- getty controlla l'accesso al sistema: exec del comando **login**
- in caso di accesso corretto, login esegue la **shell** (specificata dall'utenet in /etc/psswd)

Interazione con l'utente tramite shell

- Ogni utente puo' interagire con la shell mediante la specifica dei comandi.
- Ogni comando e' presente nel file system come file eseguibile (direttorio /bin).
- Per ogni comando, shell genera un processo figlio dedicato all'esecuzione del comando.

Relazione shell padre-figlio

Per ogni comando, shell genera un figlio; possibilita' 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): `ls -l poippo &`

Gestione degli errori: perror()

Convenzione:

- in caso di fallimento, ogni system call ritorna un valore negativo (tipicamente -1)

- in aggiunta, UNIX prevede la variabike globale di sistema *errno*, alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita. Per interpretarne il valore e' possibile usare la funzione *perror()*:
  - perror("stringa") stampa "stringa" seguita dalla descrizione del codice di errore contenuto in errno
  - la corrispondenza tra codici e descrizioni e' contenuta in <sys/errno.h>

05 - Processi Segnali Pipe

Processi interagenti

Classificazione:

- processi indipendenti
  - due processi sono indipendenti se l'esecuzione di ognuno non e' in alcun modo influenzata dall'altro
- processi integranti
  - **cooperanti**: i processi interagiscono volontariamente per raggiungere obbiettivi comuni (fanno parte della stessa applicazione)
  - **in competizione**: i processi, in generale, non danno parte della stessa applicazione, ma interagiscono indirettamente per l'acquisizione di risorse comuni

L'interazione puo' avvenire mediante due meccanismi:

- **Comunicazione**: scambio di informazioni tra i processi interagenti
- **Sincronizzazione**: imposizione di vincoli temporali, assoluti o relativi, sull'esecuzione dei processi

Ad esempio, l'istruzione k del processo P1 puo' essere eseguita soltanto dopo l'istruzione j del processo P2

Realizzazione dell'interazione: dipende dal modello di esecuzione per i processi

- **modello ad ambiente locale**: non c'e' divisione di variabili (processo pesante)
  - comunicazione avviene attraverso scambio di messaggi
  - sincronizzazione avviene mediante scambio di eventi (Segnali)
- **modello ad ambiente globale**: piu' processi possono condividere lo stesso spazio di indirizzamento --> possibilita' di condividere variabili (come nei thread)
  - variabili condivise e relativi strumenti di sincronizzazione (ad esempio, lock e semafori)

Processi interagenti mediante scambio di messaggi

Facciamo riferimento al modello ad ambiente locale:

- non vi e' memoria condivisa
- i processi possono interagire mediante scambio di messaggi: comunicazione
  Spesso SO offre meccanismi a supporto della comunicazione tra processi (Inter Process Communication - IPC)

Operazioni necessarie:

- **send**: spedizione di messaggi da un processo ad altri
- **receive**: ricezione di messaggi

Scambio di messaggi

Lo scambio di messaggi avviene mediante un canale di comunicazione tra i due processi.

Caratteristiche del canale:

- monodirezionale, bidirezionale
- uno-a-uno, uno-a-molti, molti-a-uno-, molti-a-molti
- capacita'
- modalita' di creazione: automatica, non automatica

Naming

In che modo viene specificata la destinazione di un messaggio?

- **Comunicazione diretta** - al messaggio viene associato l'identificatore del processo destinatario (naming esplicito)
  send(Proc, msg)
- **Comunicazione indiretta** - il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario prelevera' il messaggio
  send(Mailbox, msg)

Comunicazione diretta

Il canale e' creato automaticamente tra i due processi che devono conoscersi reciprocamente:

- canale punto-a-punto
- canale bidirezionale
  - p0: send(query, P1); p1: send(answ, P0)
- per ogni coppia di processi esiste un solo canale (<P0, P1>)

Comunicazione indiretta

I processi cooperanti non sono tenuti a conoscersi reciprocamente e si scambiano messaggi depositandoli/prelevandoli da una mailbox condivisa.

- mailbox (o porta) come risorsa astratta condivisibile da piu' processi, che funge da contenitore dei messaggi

Proprieta'

- il canale di comunicazione e' rappresentato dalla mailbox (non viene creato automaticamente)
- il canale puo' essere associato a piu' di due processi:
  - mailbox di sistema: molti-a-molti (come individuare il processo destinatario di un messaggio)
  - mailbox del processo destinatario: molti-a-uno
- canale bidirezionale:
  - p0: send(query, mbx)
  - p1: send(answ, mbx)
- per ogni coppia di processi possono esistere piu' canali (uno per ogni mailbox condivisa)

Buffering del canale

Ogni canale di comunicazione e' caratterizzato da una capacita': numero dei messaggi che e' in grado di gestire contemporaneamente.

Gestione usuale secondo politica FIFO:

- i messaggi vengono posti in una coda in attesa di essere ricevuti
- la lunghezza massima della coda rappresenta la capacita' del canale

Caso semplificato con capacita' nulla: non ci e' accomodamento perche' il canale non e' in grado di gestire messaggi in attesa

- processo mittente e destinatario devono sincronizzarsi all'atto di spedire (send)/ricevere (receive) il messaggio: comunicazione asincrona o rendez vous
- send e receive possono essere (solitamente sono) sospensive

Capacita' limitata: esiste un limite N alla dimensione della coda

- se la coda non e' piena, un nuovo messaggio viene posto in fondo
- se la coda e' piena: send e' sospensiva
- se la coda e' vuota: receive puo' essere sospensiva

Capacita' illimitata: lunghezza della coda teoricamente infinita. L'invio sul canale non e' sospensivo.

Sincronizzazione tra processi

Si e' visto che due processi possono interagire per

- **cooperare**: i due processi interagiscono allo scopo di perseguire un obiettivo comune
- **competere**: i processi possono essere logicamente indipendenti ma necessitano della stessa risorsa (dispositivo, file, variabile, ...) per la quale sono stati imposti dei vincoli di accesso. Ad esempio:
  - gli accessi di due processi a una risorsa devono escludersi mutuamente nel tempo
- In entrambi i casi e' necessario disporre di **strumenti di sincronizzazione**

Sincronizzazione permette di imporre vincoli temporali sulle operazioni dei processi interagenti

Ad esempio

- **nella cooperazione**
  - per imporre un particolare ordine cronologico alle azioni eseguite dai processi interagenti
  - per garantire che le operazioni di comunicazione avvengano secondo un ordine prefissato
- **nella competizione**
  - per garantire la mutua esclusione dei processi nell'accesso alla risorsa condivisa

Sincronizzazione tra processi nel modello ad ambiente locale

Mancando la possibilita' di condividere memoria:

- Gli accessi alle risorse "condivise" vengono controllati e coordinati da SO
- La sincronizzazione avviene mediante meccanismi offerti da SO che consentono la notifica di "eventi" asincroni (di solito privi di contenuto informativo o con contenuto minimale) tra un processo ed altri
  - segnali UNIX

Sincronizzazione tra processi nel modello ad ambiente globale

Facciamo riferimento ai processi che possono condividere variabili (modello ad ambiente globale, o a memoria condivisa) per descrivere alcuni strumenti di sincronizzazione tra processi

- **cooperazione**: lo scambio di messaggi avviene attraverso strutture dati condivise (Ad es. mailbox)
- **competizione**: le risorse sono rappresente da variabili condivise (ad esempio, puntatori a file)

In entrambi i casi e' necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa: **problema della mutua esclusione**.

Il problema della mutua esclusione

In caso di condivisione di risorse (variabili) puo' essere necessario impedire accessi concorrenti alla stessa risorsa.

- **Sezione critica**: sequenza di istruzioni mediante la quale un processo accede e puo' aggiornare variabili condivise
- **Mutua esclusione**: ogni processo esegue le proprie sezioni critiche in modo esclusivo rispetto agli altri processi

In generale per garantire la mutua esclusione nell'accesso a variabili condivise, ogni sezione critica e':

- preceduta da un prologo (entry section), mediante il quale il processo ottiene l'autorizzazione all'accesso in modo esclusivo
- seguita da un epilogo (exit section), mediante il quale il processo rilascia la risorsa

Esempio: produttore & consumatore

- Necessita' di garantire la mutua esclusione nell'esecuzione delle sezioni critiche (accesso e aggiornamento del buffer)
- Necessita' di sincronizzare i processi:
  - quando il **buffer e' vuoto**, il consumatore non puo' prelevare messaggi
  - quando il **buffer e' pieno**, il produttore non puo' depositare messaggi

Problema: finche' non si creano le condizioni per effettuare l'operazione di inserimento/prelievo, ogni processo rimane in esecuzione all'interno di un ciclo `while(cont==N);` `while(cont==0);`

Per migliorare l'efficienza del sistema, in alcuni SO e' possibile utilizzare system call del tipo:

- **dormo****()** per sospendere il processo che la chiama (stato di waiting e spreco di CPU evitato)
- **sveglia(P)** per riattivare un processo P sospeso (se P non e' sospeso, non ha effetto e il segnale di risveglio viene perso)

Possibile soluzione: semafori (Dijkstra, 1965)

Definizione di semaforo

- Tipo di dato astratto condiviso fra piu' processi al quale sono applicabili due operazioni (system call a esecuzione non interrompibile): wait(s) singal(s)
- A una variabile s di tipo semaforo sono associate:
  - una variabile intera s. value non negativa con valore iniziale >= 0
  - una coda di processi s.queue

Semaforo puo' essere condiviso da 2 o piu' processi per risolvere problemi di sincronizzazione (es. mutua esclusione)

wait()/signal()

- **wait()**
  - in caso di s.value=0, implica la **sospensione** del processo che la segue (stato running --> waiting) nella coda s.queue associata al semaforo
- **signal()**
  - non comporta concettualmente nessuna modifica nello stato del processo che l'ha eseguita, ma puo' causare il **risveglio** di un processo waiting nella coda s.queue
  - la scelta del processo da risvegliare avviene secondo una politica FIFO (il primo processo della coda)

wait() e signal() agiscono su variabili condivise e pertanto sono a loro volta **sezioni critiche**!

Atomicita' di wait() e signal()

Affinche' sia rispettato il vincolo della mutua esclusione dei processi nell'accesso al semaforo (mediante wait/signal), wait() e signal() devono essere operazioni indivisibili (azioni atomiche):

- durante un'operazione sul semaforo (wait() o signal()) nessun altro processo puo' accedere al semaforo fino a che l'operazione non e' completa o bloccata (Sospensione nella corda)

SO che mette a disposizione le primitive di Dijkstra deve realizzare wait() e signal() come operazioni non interrompibili (system call)

Sincronizzazione di processi cooperanti

Mediante semafori possiamo anche imporre vincoli temporali sull'esecuzione di processi cooperanti.

Obiettivo: vogliamo imporre che l'esecuzione della fase A (in P1) preceda sempre l'esecuzione della fase B (in P2)
Soluzione: si introduce un semaforo sync, inizializzato a 0

- se P2 esegue la wait() prima della terminazione della fase A, P2 viene sospeso
- quando P1 termina la fase A, puo' sbloccare P1, oppure portare il valore del semaforo a 1 (se P2 non e' ancora arrivato alla wait)

Produttore & consumatore con semafori

- Problema di mutua esclusione
  - produttore e consumatore non possono accedere contemporaneamente al buffer
    - semaforo binario mutex, con valore iniziale a 1
- Problema di sinconizzazione
  - produttore non puo' scrivere nel buffer se pieno
    - semaforo vuoti, con valore iniziale a N; valore dell'interno associato a vuoto rappresenta il numero di elementi liberi nel buffer

    - consumatore non puo' leggere dal buffer se vuoto
        - semaforo pieno, con valore iniziale a 0; valore dell'interno associato a pieno rappresenta il numero di elementi occupati nel buffer

Strumenti di sincronizzazione

Semafori:

- Consentono una efficiente realizzazione di politiche di sincronizzazione, anche complesse tra processi
- correttezza della realizzazione completamente a carico del programmatore

Alternative: esistono strumenti di piu' alto livello (costrutti di linguaggi di programmazione)che eliminano a priori il problema della mutua esclusione sulle variabili condivise

Problema dei "dining-philosophers"

- Ci sono 5 filosofi seduti a una tavola rotonda
- Ognuno ha davanti a se un piatto e tra ogni piatto c'e' una bacchetta
- Per mangiare un filosofo usa due bacchette, quella alla sua sinistra e quella alla sua destra che pero' sono condivise con i filosofi vicini
- Di conseguenza due filosofi vicini non possono mangiare contemporaneamente
- I filosofi oltre a mangiare, naturalmente, pensano ma questa attivita' avviene in modo indipendente: l'unico momento in cui si devono sincronizzare e' quando mangiano

Risorse condivise:

- Ciotola di riso (data da set)
- Semafori **bastoncini[5]** inizializzati a 1

Meccanismi alternativi di sincronizzazione: monitor

Coda di accesso regolata e disciplinata verso i dati condivisi, magari con priorita' differenziate

---

Sincronizzazione tra processi UNIX: i segnali

Sincronizzazione tra processi

Processi interagenti possono avere bisogno di meccanismi di sincronizzazione.
Ad esempio, abbiamo visto e rivedremo diffusamente il caso di processi pesanti UNIX che vogliano accedere allo stesso file in lettura/scrittura (sincronizzazione di produttore e consumatore)

UNIX: non c'e' condivisione alcuna di spazio di indirizzamento tra processi. Serve un meccanismo di sincronizzazione per modello ad ambiente locale --> segnali.

Segnali

Sono interruzioni software a un processo, che notifica un evento asincrono. Ad esempio segnali:

- generati da terminale (Es. CTRL+C)
- generati da altri processi
- generati dal kernel SO in seguito ad eccezioni HW (violazione dei limiti di memoria, divisione per 0, ...)
- generati da kernel SO in seguito a condizioni SW (time-out, scrittura su pipe chiusa come vedremo in seguito, ...)

Segnali UNIX

Un segnale puo' essere inviato

- dal kernel del SO a un processo
- da un processo utente ad altri processi utente (es. comando kill)

Quando un processo riceve un segnale, puo' comportarsi in tre modi diversi

1. gestire il segnale con una funzione handler definital dal programmatore
2. eseguire un'azione predefinita dal SO (azione di default)
3. ignorare il segnale (nessuna reazione)

Nei primi due casi, processo reagisce in modo asincrono al segnale

1. interruzione dell'esecuzione
2. esecuzione dell'azione associata (handler o default)
3. ritorno alla prossima istruzione del codice del processo interrotto

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

Gestione dei segnali UNIX

Quando un processo riceve un segnale, puo' gestirlo in 3 modi diversi:

- gestire il segnale con una funzione handler definita dal programmatore
- eseguire un'azione predefinita dal SO (azione di default)
- ignorare il segnale

NB: non tutti i segnalo possono essere gestiti in modalita' scelta esplicitamente dai processi: SIGKILL e SIGSTOP non sono ne intercettabili, ne ignorabili.
--> qualunque processo, alla ricezione di SIGKILL o SIGSTOP esegue sempre l'azione di default.

System call `signal`

Ogni processo puo' gestire esplicitamente un segnale utilizzando la system call signal():

```
typedef void (*handler_t)(int);
handler_t signal(int sig, handler_t handler);
```

- sig e' l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro handler e' un puntatore a una funzione che indica l'azione da associare al segnale. handler() puo':
  - puntare alla routine di gestione dell'interruzione (handler)
  - valere SIG_IGN (nel caso di segnale ignorato)
  - valere SIG_DFL (nel caso di azioni di default)
- ritorna un puntatore a funzione:
  - al precedente gestore del segnale
  - SIG_ERR(-1), nel caso di errore

Routine di gestione del segnale (handler)

Caratteristiche:

- handler prevede sempre un parametro formale di tipo int che rappresenta il numero del segnale eddettivamente ricevuto
- handler no restituisce alcun risultato

Esempio: gestore del SIGCHLD

SIGCHLD e' il segnale che il kernel del SO invia a un processo padre quando uno dei suoi figli termina.

Tramite l'uso di segnali e' possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione handler per la gestione di SIGCHLD:

- la funzione handler verra' attivata in modo asincrono alla ricezione del segnale
- handler chiamera' wait() con cui il padre porta' raccoglier ed eventualmente gestire lo stato di terminazione del figlio

Segnali & fork()

Le associazioni segnali-azioni vengono registrate in User Structure del processo

Siccome:

- `fork()` copia User Structure del padre in quella del figlio
- padre e figlio condividono lo stesso codice, quindi
- il figlio eredita dal padre le informazioni relative alla gestione dei segnai:
  - ignora gli stessi segnali ignorati dal padre
  - gestisce con le stesse funzioni gli stessi segnali gestiti dal padre
  - segnali a default del figlio sono gli stessi del padre

--> ovviamente signal() del figlio successive alla fork() non hanno effetto sulla gestione dei segnali del padre

Segnali & exec()

Sappiamo che

- `exec()` sostituisce codice e dati del processo invocante
- User Structure viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo exec() non sono piu' visibili)

quindi

- dopo exec(), un processo:
  - ignora gli stessi segnali ignorati prima di exec()
  - i segnali a default rimangono a default ma
  - i segnali che prima erano festiti, vengono riportati a default

System call `kill()`

I processi possono inviare segnali ad altri processi invocando la system call `kill()`

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

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

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

`alarm()`

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

- imposta un timer che dopo N secondi inviera' allo stesso processo il segnale SIGALRM
- ritorna:
  - 0, se non vi erano time-out impostati in precedenza
  - il numero di secondi mancante allo scadere del time-out precedente

NB: comportamento di default associato a ricezione di SIGALRM e' la terminazione

`pause()`

```
int pause(void)
```

- sospende il processo fino alla ricezione di un qualunque segnale
- ritorna -1 (errno = EINTR)

Gestione durevole di segnali con handler

Non sempre l'associazione segnale/handler e' durevole:

- alcune implementazioni di UNIX (BSD, SystemV r3 e seguenti), prevedono che l'azione rimanga installata anche dopo la ricezione del segnale
- in alcune realizzazioni (SystemV prime release), dopo l'attivazione, handler ripristina automatiamente l'azione di default. In questi casi, occorre riagganciare il segnale handler.

Modello affidabile dei segnali

Aspetti:

1. se il gestore non rimane installato, e' comunque possibile reinstallare il gestore all'interno dell'handler

2. che cosa succede se arriva il segnale durante l'esecuzione dell'handler?
    - innestamento della routine di gestione?
    - perdita del segnale?
    - accomodamento dei segnali (segnali reliable, BSD 4.2)

`sigaction()`

- La primitiva signal() non e' portabile perche' ha una semantica diversa in diverse versioni di Unix. Per ovviare a questo problema POSIX.1 introduce la sigaction()
- La sigaction() permette di esaminare e/o modificare l'azione associata con un particolare segnale. Si noti che POSIX.1 richiede che un segnale rimanga installato (fino a una modifica esplicita del comportamento)
- Con la sigaction() e' anche possibile specificare il restart automatico delle system call interrotte da un segnale.

signal() vs. sigaction() in sintesi

- signal() con semantica variabile reliable/unreliable
    - unreliable in alcune versioni di UNIX/Linux
    - segnali da reinstallare ogni volta, corsa critica tra inizio handler e reinstallazione handler come prima istruzione dell'handler
    - possibile esecuzione innestata dell'handler se ricezione dello stesso segnale quando siamo ancora nell'handler
- sigaction() invece e' sempre reliable
    -
        - semantica ben definita, identica in ogni versione di UNIX/Linux
          - non c'e' bisogno di reinstallare l'handler
          - non perdiamo segnali: il segnale che ha causato l'attivazione dell'handler e' automaticamente bloccato fino alla fine dell'esecuzione dell'handler stesso

Note sul codice dei gestori segnali

1. Il codice dei gestiori dei segnali non dovrebbe mai contenere chiamate a primitive di I/O, che sono lente e potenzialmente bloccanti.
2. Il modello di comunicazione tra il codice di gestione dei segnali e il codice principale dell'applicazione attraverso l'uso di una variabile statica rappresenta invece una "best practice". Tuttavia, in questi casi e' importante che la variabile utilizzata sia di tipo sig_atomic_t e che essa venga dichiarata con la keyword volatile

Dettagli du sig_atomic_t e volatile

- Infatti, lo standard ISO C definisce sig_atomi_t come un tipo di dato che puo' essere acceduto senza interruzioni. Ovverosia, nessuna lettura da o scrittura su una variabile di tipo sig_atomic_t sara' interrrotta, per esempio dall'occorrenza di un nuovo segnale.
- La keyword volatile invece da istruzione al compilatore di non ottimizzare l'accesso alla variabile corrispondente, che in questo caso produrrebbe un comportamento non corretto del nostro programma
- Se non usassimo sig_atomic_t e volatile nella definizione della variabile, sarebbe molto probabile che alcuni cambiamenti allo stato di una variabile nel codice del gestore del segnale non venissero notati dal codice dell'applicazione a causa della ricezione di ulteriori segnali o di ottimizzazioni del compilatore.

Comunicazione tra processi UNIX

Comunicazione tra processi UNIX

Processi UNIX non possono condividere memoria (modello ad ambiente locale).

Interazione tra processi puo' avvenire

- mediante la condivisione di file
  - complessita': realizzazione della sincronizzazione tra i processi
- attraverso specifici strumenti di Inter Process Communication (IPC):
  - tra processi sulla stessa macchina
    - pipe (tra processi della stessa gerarchia)
    - fifo (qualunque insieme di processi)
  - tra ptocessi in noodi diversi della stessa rete:
    - socket

pipe

La pipe e' un canale di comunicazione tra processi

- unidirezionale: accessibile mediante due estremi distinti, uno di lettura e uno di scrittura
- (teoricamente) molti-a-molti:
  - piu' processi possono spedire messaggi attraverso la stessa pipe
  - piu' processi possono ricevere messaggi attraverso la stessa pipe
- capacita' limitata
  - in grado di gestire l'accomodamento di un numero limitato di messaggi, gestiti in modo FIFO. Limite stabilito dalla dimensione della pipe (Es 4096B)

Comunicazione attraverso pipe

Mediante la pipe, la comunicazione tra processi e' indiretta (senza naming esplicito): modello mailbox

Pipe: unidirezionalita'/bidirezionalita'

Uno stesso processo puo':

- sia depositare messaggi nella pipe (send), mediante il lato di scrittura
- sia prelevare messaggi dalla pipe (receive), mediante il lato di lettura

la pipe puo' anche consentire una comunicazione "bidirezionale" tra Pe Q (ma il programmatore deve rigidamente disciplinarne l'uso per l'utilizzo corretto)

System call pipe

Per creare una pipe: `int pipe(int fd[2]);`

fd e' 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] e' il lato di scrittura della pipe

la system call pipe restituisce:

- un valore negativo, in caso di fallimento
- 0, se ha successo

Creazione di una pipe

Se pipe(fd) ha successo:

- vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a fd[0] e fd[1]
  - fd[0] : lato di lettura (receive) della pipe
  - fd[1] : lato di scrittura (send) della pipe

Omogeneita' con i file

Ogni lato di accesso alla pipe e' visto dal processo in modo omogeneo a qualunque altro file (file descriptor). Si puo' accedere alla pipe mediante la system call di lettura/scrittura su file read(), write()

Sincronizzazione automatica della pipe

Il canale (pipe) ha capacita' limitata. Come nel caso di produttore/consumatore e' necessario sincronizzare i processi. Sincronizzazione automatica in UNIX:

- se la pipe e' vuota: un processo che legge si blocca

- se la pipe e' piena: un processo che scrive si blocca

Sincronizzazione automatica: read() e write() sono implementate in modo sospensivo dal SO UNIX

Quali processi possono comunicare mediante pipe?

Per mittente e destinatario il riferimento al canale di comunicazione e' un array di file descriptor:

- soltanto processi appartenenti a una stessa gerarchia (cioe', che hanno un antenato in comune) possono scambiarsi messaggi mediante pipe. Ad esemio, possibilita' di comunicazione:
  - tra **processi fratelli** (che ereditano la pipe dal processo padre)
  - tra un processo **padre** e un processo **figlio**
  - tra **nonno** e **nipote**

Chiusura pipe

Ogni processo puo' chiudere un estremo della pipe con la system call `close()`. La comunicazione non e' piu' possibile su di un estremo della pipe quando tutti i processi che avevano visibilita' di quell'estremo hanno compiuto una close().

Se un processo P tenta:

- **lettura** da una pipe vuota il cui lato di scrittura e' effettivamente chiuso: read ritorna a 0
- **scrittura** da una pipe il cui lato di lettura e' effettivamente chiuso: write ritorna -1, e il segnale **SIGPIPE** viene inviato a P (broken pipe)

System call dup

Per duplicare un elemento della tabella dei file aperti di processo:

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

- fd e' il file descriptor del file da duplicare
- L'effetto di dup() e' copiare l'elemento fd nella tabella dei file aperti nella prima posizione libera (quella con l'indice minimo tra quelle disponibili)
- Restituisce il nuovo file descriptor (del file aperto copiato), oppure -1 (in caso di errore)

Stdin, stdout, stderr

Per convenzione, per ogni processo vengono aperti automaticamente 3 descrittori di file associati ai primi tre elementi della tabella

- stdin (fd 0) --> tastiera
- stdout (fd 1) --> video
- stderr (fd 2) --> video

dup() & piping

Tramite dup() si puo' realizzare il pipin dei comandi.

Ad esempio: `ls -lR | grep Jun | more`

Vengono creati 3 processi (uno per ogni comando) in modo che:

- stdout di ls sia diretto nello stdin di grep
- stdout di grep sia ridiretto nello stdin di more

Pipe: possibili svantaggi

Il meccanismo della pipe ha due svantaggi:

- consente la comunicazione solo tra processi in relazione di parentela
- non e' persistente: pipe viene distrutta quando terminano tutti i processi che hanno accesso ai suoi estremi

Per realizzare la comunicazione persistente tra una coppia di processi non appartenenti alla stessa gerarchia? --> FIFO

fifo

E' una pipe con nome nel fils system:

- Esattamente come le pipe normali, canale unidirezionale del tipo first-in-first-out
- e' rappresentabile da un file nel file system: persistenza, visibilita' potenzialmente globale
- ha un proprietario, un insieme di diritti e una lunghezza
- e' creata dalla system call `mkfifo()`
- e' aperta e acceduta con le stesse system call dei file

Per creare una fifo (pipe con nome):

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

- pathname e' il none della fifo
- mode esprime i permessi

restituisce 0, in caso di successo, un valore negativo, in caso contrario

Apertura/chiusura di fifo

Una volta creata, fifo puo' essere aperta (come tutti i file) mediante `open()`. Ad esempio, un processo destinatario di messaggi:

```
int fd;
fd=open("myfifo", O_RDONLY);
```

Per chiudere la fifo, si usa `close(fd)`.

Per eliminare la fifo, si usa `unlink("myfifo")`

Accesso a fifo

Una volta aperta, fifo puo' essere acceduta (come tutti i file) mediante `read()/write()`. Ad esempio, un processo destinatario di messaggi:

```
int fd
char msg[100]
fd=open("myfifo", O_RDONLY)
read(fd, msg, 10)
```

Tabella file/pipe/socket descriptor

- Tabella associata ad ogni processo utente e costituita da un elemento (riga) per ogni dile dal processo
- Indice della tabella = descrittore del file (fd)
- Per convenzione, per ogni processo vengono aperti automaticamente 3 descrittori di file associati ai primi tre elementi della tabella stdin (0), stdout (1), stderr (2) associati rispettivamente alla tastiera (0) e al video (1 e 2)
- Ogni entry (riga) della tabella contiene un puntatore o indice della riga della tabella globale dei file aperti relativa ai file

06 - Thread Java POSIX

Thread e Multithreading

- Per risolvere i problemi di efficienza del modello a processi pesanti (modello ad ambiente locale) e' possibile far ricorso al modello a ambiente globale, a processi leggeri (o thread)
- **Thread**: singolo flusso sequenziale di esecuzione all'interno di un processo
- **Multithreading**: esecuzione concorrente (in parallelo o interleaved) di diversi thread nel contesto di un piccolo processo

Thread

Un thread e' un singolo flusso sequenziale di controllo all'interno di un processo. Un thread (o processo leggero) e' un'unita' 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 sono appartenenti allo stesso processo condividono loro lo stesso spazio di indirizzamento
- ha stack e program counter privati

Modello ad ambiente globale

Caratteristiche del modello computazionale multithreaded (modello ad ambiente globale):

- I thread non hanno uno spazio di indirizzamento riservato: tutti i thread di un processo condividono lo stesso spazio di indirizzamento --> possibilita' di definire dati thread-local, sia in Java che in POSIX
- I thread hanno eecution stack e program counter privati
- La comunicazione fra thread puo' avvenire direttamente, tramite la condivisione di aree di memoria --> necessita' di meccanismi di sincronizzazione

Nel modello ad ambiente globale il termine "processo" non identifica piu' un singolo flusso di esecuzione di un programma ma invece il contesto di esecuzione di piu' thread, con tutti i dati che possono essere condivisi da questi ultimi.

Vantaggi e svantaggi del Multithreading

- Context switch e creazione piu' leggeri
- Minore occupazione di risorse
- Comunicazione tra thread piu' efficiente
- I thread permettono di sfruttare al massimo le architetture hardware con CPU multiple (prestazioni)
- Problemi di sincronizzazione piu' rilevati e frequenti
- Memoria virtuale condivisa limitata alla memoria riservata dal SO per un singolo processo
- Maggiore difficolta' nello sviluppo e nel debugging!!!

In presenza di applicazioni che richiedono la condivisione di informazioni si puo' ottenere un notevole incremento di efficienza e conseguente miglioramento delle prestazioni.

Multithreading: esempi di utilizzo

Applicazioni web

- Client-side: i browser web usano i thread per creare piu' connessioni simultanee verso server diversi, per scaricare diverse risorse in parallelo da uno stesso server, e per gestire il processo di visualizzazione di una pagina web. Tale concorrenza permette di ridurre il tempo di risposta alle richieste dell'utente e quindi aumentare la qualita' dell'esperienza percepita

- Server-side: i server web usano i thread per gestire le richieste dei client. I thread possono condividere in modo efficiente sia la cache in cui vengono memorizzate le risorse piu' utilizzate recentemente che le informazioni contestuali .

Altri esempi:

- Implementazione di algoritmi parallelizzabili, non-blocking I/O, timer multipli, task indipendenti, responsive User Interface (UI), ecc...

Multithreading in Java

Il linguaggio Java supporta nativamente il multithreading. Ogni esecuzione della JVM da origine a un unico processo, e tutto quello che viene mandato in esecuzione dalla macchina virtuale da origina a un thread. Un thread e' un oggetto particolare al quale si richiede un servizio (chiamato `start()`) corrispondente al lancio di un'attivita', di un thread, ma che non si aspetta che il servizio termini: esso procede in concorrenza a che lo ha lanciato.

A livello di linguaggio, i thread sono rappresentati da istanze della classe **Thread**.

Per creare un nuovo thread ci sono due metodi:

1. **Istanziare Thread** passando come parametro un oggetto ottenuto implementando **l'interfaccia Runnable**
2. Estendere direttamente la classe Thread

In entrambi i casi il programmatore deve implementare una classe che definisca il metodo `run()`, contenente il codice del thread da mandare in esecuzione.

La classe **Thread** e' una classe (non astratta) attraverso la quale si accede a tutte le principali funzionalita' per la gestione dei thread.

L'interfaccia **Runnable** definisce il solo metodo run(), identico a quello della classe Thread (che infatti implementa l'interfaccia Runnable). L'implementazione della interfaccia Runnable consente alle istanza di una classe non derivata da Thread di essere eseguite come un thread (purche' venga agganciata a un oggetto di tipo Thread)

Un programma Java termina quando termina l'ultimo dei thread in esecuzione (contrariamente a quanto accade in POSIX threads e nei thread Windows)

Multithreading in Java - Metodo 1 - Implementazione interfaccia Runnable

Procedimento;

1. Definire una classe che implementi l'interfaccia Runnable, quindi definendone il metodo run()
2. Creare un'istanza di tale classe

3. Creare un'istanza della classe Thread, passando al costruttore un reference all'oggetto Runnable creato precedentemente
4. Invocare il metodo start() sull'oggetto Thread appena creato. Cio' produrra' l'esecuzione, in un thread separato, del metodo run() dell'istanza di Runnable passata come argomento al costruttore di Thread.

Multithreading in Java - Metodo 2 - Sottoclasse di Thread

Procedimento:

1. definire una sottoclasse della classe Thread, facendo un opportuno override del metodo run()
2. creare un'istanza di tale sottoclasse
3. invocare il metodo start() su tale istanza, la quale, a sua volta, richiamera' il metodo run() in un thread separato

Multithreading in Java - confronto tra i due metodi

Metodo 1:

- (+) Maggiore flessibilita' derivante dal poter essere sottoclasse di qualsiasi altra classe (utoile per ovviare all'impossibilita' di avere ereditarieta' multipla in java).
- (-) Modalita' leggermente piu' macchinosa

Metodo 2:

- (+) Modalita' piu' immediata e semplice
- (-) Scarsa flessibilita' derivante dalla necessita' di ereditare dalla classe Thread, che impedisce l'ereditarieta' da altre classi

Java - ciclo di vita di un thread

Durante il suo ciclo di vita, un thread puo' essere in uno dei seguenti stati:

- **New Thread (creato)**: (subito dopo l'istruzione new) ole variabili sono state allocate e inizializzate. Il thread e' in attesa di passare allo stato Runnable, transizione che verra' effettuata in seguito a una chiamata al metodo start.
- **Runnable**: il thread e' in esecuzione, oppure pronto per l'esecuzione e in coda d'attesa per ottenere l'utilizzo della CPU
- **Not Runnable (Bloccato)**: il thread non puo' essere messo in esecuzione dallo scheduler della JVM. I thread entrano in questo stato quando sono in attesa di un'operazione di I/O o sospesi da una primitiva di sincronizzazione come sleep() o wait().
- **Dead**: il thread ha terminato la sua esecuzione in seguito alla terminazione del flusso di istruzioni del metodo run() o alla chiamata del metodo stop() da parte di un altro thread

Java - Metodi per il controllo di un thread

- `start()`: fa partire l'esecuzione di un thread. La macchina virtuale Java invoca il metodo run() del thread appena creato
- `stop()`: forza la terminazione dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente liberate (lock inclusi), come effetto della propagazione dell'eccezione ThreadDeath.
- `suspend()`: blocca l'esecuzione di un thread in attesa di una successiva operazione di resume. Non libera le risorse (neanche lock) impegnate dal thread (possibilita' di deadlock)
- `resume()`: riprende l'esecuzione di un thread precedentemente sospeso. Se il thread riattivato ha una priorita' maggiore di quello corrente in esecuzione, avra' subito accesso alla CPU, altrimenti andra' in coda l'attesa.
- `sleep(long t)`: blocca per un tempo specifico (t) l'esecuzione di un thread. Nessun lock in possesso del thread viene rilasciato
- `join()`: blocca il thread chiamante in attesa della terminazione del thread di cui si invoca il metodo. Anche con timeout
- `yield()`: sospende l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in coda d'attesa

Java - il problema di `stop()` e `suspend()`

stop() e suspend() rappresentano azioni "brutali" sul ciclo di vita di un thread --> rischio di determinare situazioni di deadlock o di inconsistenze:

- se il thread sospeso aveva acquisito una risorsa in maniera esclusiva, tale risorsa rimane bloccata e non e' utilizzabile da altri perche' il thread sospeso non ha avuto modo di rilasciare il lock su di essa.
- se il thread interrotto stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera atomica, l'interruzione puo' condurre a uno stato inconsistente del Sistema

Java - scheduling dei thread

La macchina virtuale Java ha un algoritmo di scheduling basato su livelli di priorita' statici fissati a priori (Fixed Priority Scheduling):

- il thread da mandare in esecuzione viene scelto fra quelli nello stato runnable, tipicamente quello con priorita' piu' alta
- il livello di priorita' non viene mai cambiato dalla JVM (statico)
- le specifiche ufficiali della JVM non stabiliscono come venga scelto il thread da mandare in esecuzione tra quelli disponibili (es. caso di piu' thread con lo stesso livello di priorita'). La politica dipende dalla specifica implementazione della JVM (FIFO, Round-Robin, etc...)

Di conseguenza, non c'e' nessuna garanzia che si implementata una gestione in time-slicing (Round-Robin).

Il thread messo in esecuzione dallo scheduler viene interrotto soltanto se si verifica uno dei seguenti eventi:

- Il thread termina la sua esecuzione oppure chiama un metodo che lo fa uscire dallo stato runnable (ad esempio, Yield())
- Un thread con priorita' piu' alta diventa disponibile per l'esecuzione (ovverosia entra nello stato runnable)
- Il thread termina il proprio quanto di tempo (solo nel caso di scheduling Round-Robin)
- Il processore riceve un Interrupt Hardware

Java - priorita' di Scheduling

- Non si devono progettare applicazioni Java assumendo che il thread a priorita' piu' elevata sara' sempre quello in esecuzione, se nello stato runnable
- Il meccanismo di assegnazione di priorita' diverse ai bari thread e' stato pensato solo per consentire agli sviluppatori di dare suggerimenti alla JVM per migliorare l'efficienza di un programma
- Alla fine, e' sempre la JVM ad avere l'ultima parola nello scheduling. Diverse JVM possono avere comportamenti diversi
- L'assegnazione di diverse priorita' ai thread non e' un sostituto delle primitive di sincronizzazione

Sincronizzazione di thread

- Differenti thread condividono lo stesso spazio di memoria (heap)
  - e' possibile che piu' thread accedano contemporaneamente a uno stesso oggetto, invocando un metodo che modifica lo stato dell'oggetto
  - stato finale dell'oggetto sara' funzione dell'ordine con cui i thread accedono ai dati
- Servono meccanismi di sincronizzazione
- In Java, la sincronizzazione e' implementata a livello di linguaggio (modificatore synchronized) e attraverso primitive di sincronizzazione di basso (`wait()`, `notify()`, `notifyAll()` e di alto livello (concurrency utilities)
- Anche lo standard POSIX definisce funzioni e costrutti per la costruzione e la manipolazione di mutex e altri meccanismi di sincronizzazione

Accesso esclusivo

Per evitare che thread diversi interferiscano durante l'accesso ad oggetti condivisi si possono imporre accessi esclusivi in modo molto facile in Java. JVM supporta la definizione di lock sui singoli oggetti tramite la keywoord synchronized.

Synchronized puo' essere definita:

- **su metodo**
- su singolo blocco di codice

Synchronized

In pratica:

- a ogni oggetto Java e' automaticamente associato un unico lock
- quando un thread vuole accedere ad u metodo/blocco synchronized, si deve acquisire il lock dell'oggetto (impedendo cosi' l'accesso ad ogni altro thread)
- lock viene automaticamente rilasciato quando il thread esce dal metodo/blocco synchronized (o se viene interrotto da un'eccezione)
- thread che non riesce ad acquisire un lock rimane sospeso sulla richiesta della risorsa fino a che il lock non e' disponibile
- Ad ogni oggetto viene assegnato **un solo lock a livello di oggetto** (non di classe ne di metodo in Java)
    - due thread non possono accedere contemporaneamente a due metodi/blocchi synchronized diversi di uno stesso oggetto
- Tuttavia altri thread sono liberi di accedere a metodi/blocchi **non synchronized** associati allo stesso oggetto.

Esistono due situazioni in cui synchronized non e' sufficiente per impedire accessi concorrenti. Supponiamo che un metodo synchronized sia l'unico modo per variare lo stato di un oggetto. Che cosa accade se il thread che ha acquisito il lock si blocca all'interno del metodo stesso in attesa di un cambiamento di stato?

Soluzione tramite uso di wait()

Thread che invoca wait()

- si blocca in attesa che un altro thread invochi notify() o notifyAll() per quell'oggetto
- deve essere in possesso del lock sull'oggetto
- al momento della invocazione di wait() il thread rilascia il lock

notifyAll()

- notify() - il thread che la invoca risveglia uno dei thread in attesa, scelto arbitrariamente
- notifyAll() - il thread che la invoca
    - **risveglia tutti i thread in attesa**: essi competeranno per l'accesso all'oggetto

notifyAll() e' preferibile (puo' essere necessaria) se piu' thread possono essere in attesa

Alcune regole empiriche

1. Se due o piu' thread possono modificare lo stato di un oggetto, e' necessario dichiarare synchronized i metodi di accesso a tale stato

2. Se deve attendere la variazione dello stato di un oggetto, thread deve invocare wait()
3. Ogni volta che un metodo attua una variazione dello stato di un oggetto, esse deve invocare notifyAll()
4. E' necessario verificare che ad ogni chiamata a wait() corrisponda una chiamata a notifyAll()

Thread safety

- Qualsiasi programma in cui piu' thread accedono a una stessa risorsa (con almeno un'operazione di scrittura), senza un'adeguata sincronizzazione, e' **bacato** . Questo tipo di bug, secondo il quale l'output di una porzione di codice dipende dall'ordine in cui i thread accedono le risorse condivise, e' tipicamente molto difficile da individuare e correggere, in quanto potrebbe presentarsi solo in particolari condizioni.
- L'unico modo di risolvere il problema della thread safety e' quello di utilizzare opportunamente le primitive di sincronizzazione
- Una porzione di codice e' thread-safe se si comporta correttamente quando viene usata da piu' thread, indipendentemente da loro ordine di esecuzione, senza alcuna sincronizzazione o cordinazione da parte del codice chiamante.
- E' molto piu' facile scrivere codice thread safe, pianificando un adeguato uso delle primitive di sincronizzazione al momento del progetto, che modificarlo successivamente per renderlo tale.

Java - Daemon Thread

- I thread di Java possono essere di due tipi: **user thread** e **demo thread**
- L'unica differenza tra le due tipologie di thread sta nel fatto che la virtual machine di Java termina l'esecuzione di un daemon thread quando termina l'ultimo user thread.
- I daemon thread svolgono servizi per gli user thread, e spesso restano in esecuzione per tutta la durata di una sessione della virtual machine (ad esempio, il garbage collector).
- Di default, un thread assume lo stato del thread che lo crea. E' possibile verificare e modificare lo stato di un thread con metodi `isDaemon()` e `setDaemon()`, ma solo prima di mandarlo in esecuzione

Multithreading in POSIX threads

Per realizzare programmi multithreaded in UNIX, si deve fare riferimento alla API standard POSIX threads (o Pthreads):

- Va incluso il file header specifico:
  - ```
    #include <pthread.h>
    ```
- Primitiva di sistema per la creazione (spawning) di un nuovo thread:
  - ```
    int pthread_create (pthread_t * t,
                        const pthread_attr_t * attr,
    ```

```
                        void * (*Start_func)(void*),
                        void *arg);
```

- Primitiva di sistema per effettuare il join con un thread:
  - ```
    int pthread_join (pthread_t t, void **retval);
    ```
- Primitiva di sistema per lo spawning di un nuovo thread:
  - ```
    int pthread_create (pthread_t * t,
                        const pthread_attr_t * attr,
                        void * (*Start_func)(void*),
                        void *arg);
    ```
- Restituisce il valore 0 in caso di successo, o un valore positivo in caso di errore
- il thread t viene creato con gli attributi specifici in attr ed esegue la funzione start_func, con argomanto arg
- Attributi specificano diversi comportamenti: detached mode, stack size, scheduling priority, ecc.
- Primitiva di sistema per effettuare il join con un thread:
  - ```
    int pthread_join (pthread_t t, void **retval);
    ```
- Restituisce valore 0 in caso di successo, o un valore positivo in caso di errore
- il thread che lancia la chiamata di join si mette in attesa per la terminazione del thread
- Al ritorno della chiamata, retval conterra' il valore di ritorno della funzione eseguita dal thread t
  - retval puo' essere NULL
- Se un thread viene creato in modalita' noin-detached (default in POSIX), allora la join e' necessaria per evitare di lasciare zombie, e quindi un memory leak

Thread detached in POSIX

Anche Pthreads supporta la creazione di thread di tipo "demone", che pero' prendono il nome di thread detached. La differenza fra le due tipologie di thread sta nella gestione della memoria e nalla possibilita' di effettuare il join con un altro thread.
Un thread non-detached permette ad altri thred di sincronizzarsi con la sua terminazione, tramite la primitiva pthread_join(). E' necessario effettuare la join per liberare la memoria associata ai thread non-detached che abbiano finito la loro esecuzione (evitare di lasciare zombie).
Un thread detached non e' joinable, ma il sistema effettua automaticamente il release della memoria ad esso associata quando questo termina la sua esecuzione (non c'e' il problema di thread zombie)
    ```

07 - File System

Gestione del file system

Parte di SO che fornisce i meccanismi di accesso e memorizzazione delle informazioni (programmi e dati) allocate in mmorie di massa.

Realizza i concetti astratti

- di **file**: unita' 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)

--> Le caratteristiche di file, direttorio e partizione sono del tutto indipendenti da natura e tipo d dispositivo utilizzato

File

E' un insieme di informazioni:

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

rappresentati come insieme di record logici

Ogni file e' individuato da (almeno) un **nome** simbolico mediante il quale puo' essere riferito (ad esempio, nell'invocazione di comandi o system call). Ogni file e' caratterizzato da un insieme di **atttributi**

Attributi del file

A seconda del SO, i file possono avere attributi diversi.

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 modifica)

In SO multiutente anche

- **utente proprietario**

- **protezione**: diritti di accesso al file per gli utenti del sistema

Descrittori del file: e' la struttura dati che contiene gli attributi di un file.

Ogni descrittore di file deve essere memorizzato in modo persistente: SO mantiene l'insieme dei descrittori di tutti i file presenti nel file system in apposite strutture in memoria secondaria (ad es. UNIX: i-list)

Operazioni sui file

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

Tipiche operazioni

- **Creazione**: allocazione di un file in memoria secondaria e inizializzazione dei suoi attributi
- **Lettura** di record logici dal file
- **Scrittura**: inserimento di nuovi record logici all'interno del file
- **Cancellazione**: eliminazione del file dal file system

Ogni operazione richiederebbe la localizzazione di informazioni su disco, come:

- indirizzi dei record logici a cui accedere
- altri attributi del file
- record logici

Per migliorare l'efficienza:

- SO mantiene in memoria una struttura che registra i file attualmente in uso (file aperti) - tabella dei file aperti per ogni file aperto {puntatore al file, posizione su disco, ...}
- Spesso viene fatto il **memory mapping** dei file aperti:
  - i file aperti (o porzioni di essi) vengono temporaneamente copiati in memoria centrale --> accessi piu' veloci

Operazioni necessarie

- **Apertura**: introduzione di un nuovo elemento nella tabella dei file aperti e eventuale memory mapping del file
- **Chiusura**: salvataggio del file in memoria secondaria ed eliminazione dell'elemento corrispondente dalla tabella dei file aperti

Struttura interna dei file

Ogni dispositivo di memorizzazione secondaria viene partizionato in blocchi (o record fisici)

- **Blocco**: unita' di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Sempre di dimensione fissa.

L'utente vede il file come un insieme di record logici

- **Record Logico**: unita' di trasferimento logico nelle operazioni di accesso al file (es. lettura, scrittura di blocchi). Di dimensione variabile.

Blocchi & record logici

Uno dei compiti di SO (parte di gestione del file system) e' stabilire una corrispondenza tra record logici e blocchi

Usualmente:

- Dimensione (blocco) >> Dimensione (record logico)
  impaccamento di record logici all'interno di blocchi

Metodi di accesso

L'accesso a file puo' avvenire secondo varie modalita':

- accesso sequenziale
- accesso diretto
- accesso a indice

Il metodo di accesso e' **indipendente**:

- dal tipo di dispositivo utilizzato
- dalla tecnica di allocazione dei blocchi in memoria secondaria

Accesso sequenziale

Il file e' una sequenza [R_1, R_2, ..., R_N] di record logici:

- per accedere ad un particolare record logico R_i, e' necessario accedere prima agli (i-1) record che lo precedono.
- le operazioni di accesso sono del tipo:
  - **readnext**: lettura del prossimo record logico nella sequenza
  - **writenext**: scrittura del prossimo record logico
- ogni operazione di accesso (lettura/scrittura) posiziona il **puntatore al file** sull'elemento successivo a quello appena letto/scritto

UNIX prevede questo tipo di accesso

Accesso diretto

Il file e' un insieme $\{R_1, R_2, ..., R_N\}$ di record logici numerati con associata la nozione di posizione:

- si puo' 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
- utile quando si vuole accedere a grossi file per estrarre/aggiornare poche informazioni (ad esempio nell'accesso a database)

Accesso a indice

Ad ogni file viene associata una **struttura dati** contenente un indice delle informazioni contenute per accedere a un record logico, si esegue una ricerca nell'indice (utilizzando una chiave)

Directory (o direttorio)

Strumento per organizzare il file all'interno del file system:

- una directory puo' contenere piu' file
- e' realizzata mediante una struttura dati che associa al nome di ogni file come e/o dove e' allocato in memoria di massa

Operazioni sui direttori:

- **Creazione/cancellazione** di directory
- **Aggiunta/cancellazione** di file
- **Listing**: elenco di tutti i file contenuti nella directory
- **Attraversamento** della directory
- **Ricerca** di file in directory

Tipi di directory

La struttura logica delle directory puo' variare a seconda del SO

Schemi piu' comuni:

- **a un livello**
- **a due livelli**
- **ad albero**
- **a grafo aciclico**

Struttura a un livello

Una sola directory per ogni file system

Problemi:

- unicita' nei nomi
- multiutenza: come separare i file dei diversi utenti?

Struttura a due livelli

- primo livello (directory principale): contiene una directory per ogni utente del sistema
- secondo livello: directory utenti (a un livello)

Struttura ad albero

Organizzazione gerarchica a N livelli. Ogni direttorio puo' contenere file e altri direttori

Struttura a grafo aciclico (es. UNIX)

Estende la struttura ad albero con la possibilita' di inserire link differenti allo stesso file

Directory e partizioni

Una singola unita' disco puo' contenere piu' partizioni.
Una singola partizione puo' utilizzare piu' di una unita' disco

File System Mounting

Molti SO richiedono il mounting esplicito all'interno del file system prima di poter usare una (nuova) unita' disco

File system e protezione

Il proprietario/creatore di un file dovrebbe avere la possibilita' di controllare:

- quali azioni sono consentite sul file
- da parte di chi

Possibili tipologie di accesso:

- read
- execute
- delete
- write
- append
- list

Liste di accesso e gruppi (es. UNIX)

Modalita' di accesso: read, write, execute

- 3 classi di utenti
  1. owner access 7 --> RWX
  2. group access 6 --> RW
  3. public access 1 --> X
- Amministratore puo' creare gruppi (con nomi unici) e inserire/eliminare utenti in/da quel gruppo
- Dato un file o una directory, si devono definire le regole di accesso desiderate

Realizzazione del file system

SO si occupa anche della realizzazione del file system sui dispositivi di memorizzazione di massa:

- **realizzazione dei descrittori** e loro organizzazione
- **allocazione dei blocchi fisici**
- **gestione dello spazio libero**

Metodi di allocazione

Ogni blocco contiene un insieme di record logici contigui

Quali sono le tecniche piu' comuni per l'allocazione dei blocchi su disco?

- allocazione **contigua**
- allocazione **a lista**
- allocazione **a indice**

Allocazione contigua

Ogni file e' mappato su un insieme di blocchi fisicamente contigui

Vantaggi

- **costo** della ricerca di un blocco
- possibilita' 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 contigue sempre piu' piccole, a volte inutilizzabili
  - necessita' di azioni di compattazione

- **aumento dinamico delle dimensioni** di file

Allocazione a lista (concatenata)

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

Vantaggi

- non c'e' **frammentazione esterna**
- minor costo di allocazione

Svantaggi

- possibilita' di errore se link danneggiato
- **maggior occupazione** (spazio occupato dai puntatori)
- difficolta' di realizzazione dell'accesso diretto
- **costo della ricerca** di un bloocco

Allocazione a indice

Allocazione a lista: i puntatori ai blocchi sono distribuiti sul disco

- elevato tempo medio di accesso a un blocco
- complessita' della realizzazione del metodo di accesso diretto

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 e' associato un blocco (indice) in cui sono contenuti tutti gli indirizzi dei blocchi su cui e' allocato il file

Vantaggi

- stessi dell'allocazione a lista, piu'
  - possibilita' di accesso diretto
  - maggiore velocita' di accesso (rispetto a liste)

Svantaggi

- possibile scarso utilizzo dei blocchi indice

Tabella di allocazione dei file (FAT)

Alcuni SO (ad es. DOS e OS/2) realizzano l'allocazione a lista in modo piu' efficiente e robusto:

- per ogni partizione, viene mantenuta una tabella (FAT, File Allocation Table) in cui ogni elemento rappresenta un blocco fisico
- concatenamento di blocchi sui quali e' allocato un file e' rappresentato nella FAT

Metodi di allocazione

Riassumendo, gli aspetti caratterizzanti sono:

- grado di utilizzo della memoria
- tempo di accesso medio al blocco
- realizzazione dei metodi di accesso

Esistono SO che adottano piu' di un metodo di allocazione; spesso:

- file **piccoli** --> allocazione congiunta
- file **grandi** --> allocazione a indice

08 - File System Unix

UNIX file system: organizzazione logica

- Omogeneita': tutto e' file
- Tre categorie di file:
  - file **ordinari**
  - **direttori**
  - **dispositivi fisici**: file speciali (nel direttorio /dev)

Nome, i-number, i-node

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

UNIX file system: organizzazione fisica

- Metodo di allocazione utilizzato in UNIX e' a **indice** (a piu' livelli di indirizzamento)
- Formattazione del disco in blocchi fisici (dimensione del blocco: 512B-5096B)

- Superficie del disco partizionata in 4 regioni:
  - boot block
  - super block
  - i-list
  - data blocks
- **Boot block** contiene le procedure di inizializzazione del sistema (da eseguire al bootstrap)
- **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
- **Data Blocks** - area del disco effettivamente disponibile per la memorizzazione dei file. Contiene
  - i blocchi allocati
  - i blocchi liberi (organizzati in una lista collegata)

i-List contiene una lista di tutti i descrittori (**i-node**) dei file normali, direttori e dispositivi presenti nel file system (accesso con l'indice **i-number**)

i-node

i-node e' il **descrittore** del file

Tra gli attributi contenenti 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 link
- 13-15 indirizzi di blocchi (a seconda della realizzazione)

Indirizzamento

L'allocazione del file NON e' su blocchi fisicamente contigui. Nell'i-node sono contenuti **puntatori a blocchi** (ad esempio 13), del quali:

- i primi 10 indirizzi riferiscono blocchi di dati (indirizzamento diretto)
- 11esimo indirizzo: indirizzo di un blocco contenente a sua volta indirizzi di blocchi dati (primo livello di indirettezza)
- 12esimo indirizzo: secondo livello di indirettezza
- 13esimo indirizzo: terzo livello di indirettezza

Se: dimensione del blocco 512B=0,5KB
indirizzi di 32 bit (4 byte)

--> 1 blocco contiene 128 indirizzi

- 10 blocchi di dati sono accessibili direttamente
  file di dimensioni minori di 10*512B = 5KB sono accessibili direttamente
- 128 blocchi di dati sono accessibili con indirettezza singola (mediante il puntatore 11): 128*512B = 64KB
- 128*128 blocchi di dati sono accessibili con indirettezza doppia (mediante il puntatore 12): 128*128*512 = 8MB
- 128*128*128 blocchi di dati sono accessibili con indirettezza tripla (mediante il puntatore 13): 128*128*128*512 = 1GB

La dimensione massima del file realizzabile e' dell'ordine del GB
Dimensione massima = 1GB + 8MB + 64KB + 5KB

--> vantaggio aggiuntivo: l'accesso a file di piccole dimensioni e' piu' veloce rispetto al caso di file grandi

Breve parentesi su Linux file system

All'utente finale il file system Linux appare come una struttura ad albero gerarchico e obbedisce alla semantica UNIX. Internamente, il kernel di Linux e' capace di gestire differenti file system multipli fornendo un livello di astrazione uniforme: Virtual File System (VFS)

Linux VFS sfrutta:

- un insieme di descrittori che definiscono come un file debba apparire (inode object, file object, dile system object)
- un insieme di funzionalita' software per gestire questi oggetti

Linux Ext2fs

Ext2fs usa un meccanismo molto simile a quello standard UNIX per identificare i data block appartenenti a un file specifico

Le differenze principali riguardano le strategie di allocazione dei blocchi:

- Ext2fs usa default block size di 1KB, ma e; anche in grado di supportare blocchi da 2KB e 4Kb

- Ext2fs tenta di collocare i blocchi logicamente adiacenti di un file su blocchi fisicamente adiacenti su disco. In questo modo, e' teoricamente possibile realizzare una singola operazione di I/O che operi su diversi blocchi logicamente adiacenti

Linux: proc file system

- proc file system non serve a memorizzare dati, ma per funzioni tipicamente di monitoraggio
- E' uno pseudo-file system che fornisce accesso alla strutture dati di kernel mantenute da SO: i suoi contenuti sono tipicamente calcolati on demand
- proc realizza una struttura a directory e si occupa di mantenere i contenuti invocando le funzioni di monitoring correlate

Esempi: version, dma, stat/cpu, stat/swap, stat/process

Parentesi su MS file system NTFS

La struttura fondamentale per il file system NTFS (New Technology File System) di MS e' il volume

- basato su una partizione logica di disco
- puo' occupare una porzione di disco, un disco intero, o differenti porzioni su differenti dischi

Tutti i descrittori (metadati riguardanti il volume) sono memorizzati in un file normale

NTFS usa cluster come unita' di allocazione dello spazio su disco

- un cluster e' costituito da un numero di settori disco che e' potenza di 2
- frammentazione interna ridotta rispetto a 16-bit FAT perche' dimensione cluster minore
- Un file NTFS non e' una semplice sequenza di byte, come in MS-DOS o UNIX, ma un oggetto strutturato con attributi
- Ogni file NTFS e' descritto da una o piu' entry di un array memorizzato in un file speciale chiamato Master File Table (MFT)
- Ogni file NTFS su una unita' disco ha ID unico (64 bit)
- Lo spazio dei nomi di NTFS e' organizzato in una gerarchia di directory. Per motivi di efficienza, esiste un indice per ottimizzare l'accesso (B+ tree)

Directory

Anche le directory sono rappresentate nel file system da file

- Ogni file-directory contiene un insieme di record logici con struttura
- Ogni record rappresenta un file appartenente alla directory
    - per ogni file (o directory) appartenenza alla directory considerata, viene memorizzato il suo nome relativo, a cui viene associato il relativo **i-number** (che lo identifica univocamente)

Gestione file system in UNIX

Quali sono i meccanismi di accesso al file system?
Come vengono realizzati?

Vanno considerati

- strutture dati di sistema per il supporto all'accesso e alla gestione di file
- principali system call per l'accesso e la gestione di file

Gestione file system: concetti generali

- Acceso sequenziale
- Assenza di strutturazione: **file = sequenza di byte (stream)**
- Posizione orrente: **I/O Pointer**
- Varie **modalita'** di accesso (lettura, scrittura, lettura/scrittura, ...)
- Accesso subordinato all'operazione di **apertura**

File descriptor

- A ogni processo e' associata una **tabella dei file aperti** di dimensione limitata (attorno al centinaio di elementi)
- Ogni elemento della tabella rappresenta un file aperto dal processo ed e' individuato da un indice intero: **file descriptor**
- I file descriptor **0, 1, 2** individuano rispettivamente standard input, output, error (aperti automaticamente)
- La tabella dei file aperti del processo e' allocata nella sua **user structure**

Struttura dati kernel

Per realizzare l'accesso ai file, SO utilizza **due strutture dati** globali, allocate nell'area dati del kernel

- la **tabella dei file attivi**: per ogni aperto, contiene una copia del suo i-node --> piu' efficienti le operazioni su file evitando accessi al disco per ottenere attributi dei file acceduti
- la **tabella dei file aperti di sistema**: ha un elemento per ogni operazione di apertura relativa a file aperti (e non ancora choiusi). Ogni elemento contiene:
  - I/O pointer, posizione corrente all'interno del file
  - un puntatore all'i-node del file nella tabella dei file attivi

--> se due processo aprono separatamente lo stesso file F, la tabella conterra' due elementi distinti associati a F

Riassumendo:

- **tabella dei file aperti di processo**: nella user area del processo, contiene un elemento per ogni file aperto dal processo

- **tabella dei file aperti di sistema**: in area di SO, contiene un elemento per ogni sessione di accesso a file nel sistema
- **tabella dei file attivi**: in area di SO, contiene un elemento per ogni file aperto nel sistema

Tabella dei file aperti di sistema

Un elemento per ogni "apertura" di file: a processi che accedono allo stesso file corrispondono entry distinte.
Ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)

--> piu' processi possono accedere contemporaneamente allo stesso file, ma hanno **I/O pointer distinti**

Tabella dei file attivi

L'operazione di **apertura** provoca la copia dell'i-node in memoria centrale (se il file non e' gia' in uso).
La tabella dei file attivi contiene gli i-node di tutti i file aperti
Il numero degli elementi e' pari al numero dei file aperti (anche da piu' di un processo)

Gestione file system UNIX: 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()`
- cancellazione: `unlink()`
- linking: `link()`
- accesso diretto: `lseek()`

System Call: apertura di file

L'apertura di un file provoca:

- **inserimento di un elemento** (individuato da un file descriptor) nella prima posizione libera della tabella dei file aperti del processo
- **inserimento di un nuovo record nella tabella dei file aperti di sistema**
- la **copia dell'i-node nella tabella dei file attivi** (solo se il file non e' gia' in uso)

Apertura di un file: open()

Per aprire un file:

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

-
    - nomefile e' il nome del file (relativo o assoluto)
      - flag esprime il modo di accesso; ad esempio O_RONDLY, per accesso in lettura, O_WRONLY, per accesso in scrittura
      - mode e' un parametro richiesto soltanto se l'apertura determina la creazione del file (flag O_CREAT): in tal caso, mode specifica i bit di protezione
- Valore restituito dalla open() del file descriptor associato al file, -1 in caso di errore
    - se open() ha successo, il file viene aperto nel modo richiesto e I/O pointer posizionato sul primo elemento (tranne nel caso di O_APPEND)

Modi di apertura (definiti in <fcntl.h>)

- O_RDONLY (=0), accesso in lettura
- O_WRONLY (=1), accesso in scrittura
- O_APPEND (=2), accesso in scrittura in modalita' append (in fondo al file), sempre da associare a O_WRONLY

Inoltre e' possibile abbinare ai tre metodi precedenti, altri modi (mediante il connettore |):

- O_CREAT, per accesso in scrittura. Se il file non esiste, viene creato, e' necessario fornire il parametro mode, per esprimere i bit di protezione
- O_TRUNC, per accesso in scrittura: la lunghezza del file viene troncata a 0

Apertura del file: creat()

Per creare un file:

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

-
    - nomefile e' il nome del file (relativo o assoluto) da creare
      - mode e' necessario e specifica i 12 bit di protezione per il nuovo file
- Valore restituito da creat() e' il file descriptor associato al file, -1 in caso di errore
- se creat() ha successo, il file viene aperto in scrittura e I/O pointer posizionato sul primo elemento (se file esiste, viene cancellato e riscritto da capo)

Chiusura di file: close()

Per chiudere un file aperto:

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

fd e' il file descriptor del file da chiudere

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

Se close() ha successo:

- file memorizzato sul disco
- eliminato l'elemento di indice fd dalla tabella dei file aperti di processo
- eventualmente eliminati gli elementi dalla tabella dei file di sistema e dei file attivi

System call: lettura e scrittura

Caratteristiche:

- accesso mediante file descriptor
- ogni operazione di lettura (o scrittura) agisce sequenzialmente sul file, a partire dalla posizione corrente di I/O pointer
- possibilita' di alternare operazioni di lettura e scrittura
- atomicita' della singola operazione
- operazioni sincrone, cioe' con attesa del completamento logico dell'operazione

Lettura di file: read()

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

-
    - fd - file descriptor del file
      - buf - area in cui trasferire i byte letti
      - n - numero di caratteri da leggere
- In caso di successo, restituisce un intero (<=n) che rappresenta il nuemero di caratteri effettivamente letti

Per indicare da tastiera (in fase di input) la volonta' di terminare il file, <CRTL-D>

Se read() ha successo:

- a partire dal valore corrente di I/O pointer, vengono letti al piu' n bytes dal file fd e memorizzati all'indirizzo buff
- I/O pointer viene spostato avanti di n bytes

Scrittura file: write()

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

-
    - fd - file descriptor del file
      - buf - area da cui trasferire i byte scritti

- n - numero di caratteri da scrivere

In caso di successo, restituisce un intero positivo (==n) che rappresenta il numero di caratteri effettivamente scritti

Accesso diretto: lseek()

Per spostare I/O pointer:

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

-
  - fd - file descriptor del file
    - offset - spostamento (in byte) rispetto all'origine
    - origine - puo' 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

Cancellazione di file: unlink()

Per cancellare un file, o decrementare il numero dei suoi link

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

-
  - name - nome del file
    - ritorna 0 se OK, altrimenti -1

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

Aggiungere nomi a file esistenti: link()

Per aggiungere un link a un file esistente:

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

-
  - oldname - nome del file esistente
    - newname - nome associato al nuovo link()

link()

- incrementa il numero dei link associato al file nell'i-node
- aggiorna il direttorio (aggiunta di un nuovo elemento)

- ritorna 0 in caso di successo, -1 se fallisce

ad esempio, fallisce se:

- oldname non esiste
- newname esiste gia' (non viene sovrascritto
- oldname e newname appartengono a unita' disco diverse (in questo caso, usare link simbolici mediante symlink)

Diritti di accesso a file: chmod()

Per modificare i bit di protezione di un file:

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

- pathname - nome del file
- newmode - contiene i nuovi diritti

Ad esempio: `chmod("file.txt", "0777")`
`chmod("file.txt", "o-w")`
`chmod("file.txt", "g+x")`

Diritti di accesso a file: chown()

Per cambiare il proprietario e il gruppo di un file:

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

- pathname - nome del file
- owner - uid del nuovo proprietario
- group - grid del gruppo

Cambia proprietario/gruppo del file

Gestione directory

Alcune system call a disposizione in C/UNIX per la gestione delle directory

- chdir(): per cambiare directory (come comando shell cd)
- opendir(), closedir(): apertura e chiusura di directory
- readdir(): lettura di directory

La system call sopra fanno uso di tipi astratti per la descrizione della struttura dati directory e sono indipendenti da come il direttorio viene realizzato (BSD, System V, Linux)

Per effettuare un cambio di directory

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

-
    - nomedir - nome della directory in cui entrare
      - restituisce 0 in caso di successo (cioe' cambio di directory avvenuto), -1 in caso di fallimento
- Analogamente ai file normali, lettura/scittura di una directory possono avvenire solo dopo l'operazione di apertura opendir()
- Apertura restituisce un puntatore a DIR:
    - DIR e' un tipo di dato astratto predefinito (<dirent.h>) che consente di riferire (mediante puntatore) una directory aperta

Apertura e chiusura di directory: opendir(), closedir()

Per aprire un direttorio

```
#include <dirent.h>
DIR *opendir (char *nomedir);
```

nomedir - nome del direttorio da aprire
restituisce un valore di tipo puntatore a DIR:

- diverso da NULL se l'apertura ha successo: per gli accessi successivi, si impieghera' questo valore per riferire il direttorio
- altrimenti restituisce NULL

Chiusura del direttorio riferito al puntatore dir:

```
#include <dirent.h>
int closedir (DIR *dir);
```

Restituisce 0 in caso di successo, -1 altrimenti

Lettura di una directory

Una directory aperta puo' essere letta con readdir():

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *descr;
descr = readdir(DIR *dir);
```

dir - puntatore alla directory da leggere (valore restituito da opendir)

Restituisce:

-

    - un puntatore diverso da NULL se la lettura ha avuto successo
      - altrimenti restituisce NULL (in caso di insuccesso)

- In caso di successo, le readdir() legge un elemento dalla directory data e lo memorizza all'indirizzo puntato da descr
- descr punta ad una struttura di tipo dirent (dichiarata in dirent.h)

L'elemento nella directory: dirent

Il generico elemento (file o directory) in una directory e' rappresentato da un record di tipo dirent:

```
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[1]; /* nome del file */
}
```

La stringa che parte da d_name rappresenta il nome del file (o directory) nella directory aperta; d_nalelen la lunghezza del nome.

--> possibilita' di nomi con lunghezza variabile

Gestione di directory: creazione

Creazione di una directory

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

- pathname - nome del direttorio da creare
- mode - esprime i bit di protezione

Restituisce il valore 0 in caso di successo, altrimenti -1.

In caso di successo, crea e inizializza una directory con il nome e i diritti specificati. Vengono sempre creati i file:

- . (link alla directory corrente)
- .. (link alla directory padre)

09 - Scheduling della CPU

Scheduling della CPU

Obiettivo della multiprogrammazione: massimizzazione dell'utilizzo CPU

- **Scheduling della CPU**: commuta l'uso della CPU tra i vari processi
- **Scheduler della CPU (a breve termine)**: e' quella parte del SO che seleziona dalla coda dei processi pronti il prissimo processo al quale assegnare l'uso della CPU

Coda dei processi pronti (ready queue):

Contiene i descrittori (process control block, PCB) dei processi pronti.

Strategia di gestione della ready queue e' realizzata mediante **politiche** (algoritmi) di scheduling

Terminologia: CPU burst & I/O burst

Ogni processo alterna:

- **CPU burst**: fasi in cui viene impiegata soltanto la CPU senza interruzioni dovute a operazioni di I/O
- **I/O burst**: fasi in cui il processo effettua I/O da/verso una risorsa (dispositivo) del sistema

--> quando un processo e' in I/O burst, la CPU non viene utilizzata: in un sistema multiprogrammato, short-term scheduler assegna la CPU a un nuovo processo

Terminologia: processi I/O bound & CPU bound

A seconda delle caratteristiche dei programmi eseguiti dai processi, e' possibile classificare i processi in

- **I/O bound**: prevalenza di attivita' I/O
  - molti CPU burst di breve durata, intervallati da I/O burst di lunga durata
- **CPU bound**: prevalenza di attivita' di computazione
  - CPU burst di lunga durata, intervallati da pochi I/O burst di breve durata

Terminologia: pre-emption

Gli algoritmi di scheduling si possono classificare in due categorie:

- senza prelazione (**non pre-emptive**): CPU rimane allocata al processo running finche' esso non si sospende volontariamente o non termina
- con prelazione (**pre-emptive**): processo running puo' essere prelazionato, cioe' SO puo' sottrargli CPU per assegnarla ad un nuovo processo

--> i sistemi a divisione di tempo hanno sempre uno scheduling **pre-emptive**

Politiche e meccanismi

- Scheduler: decide quale processo assegnare la CPU
- A seguito della decisione, viene attuato il cambio di contesto (context-switch)
- Dispatcher: e' la parte di SO che realizza il cambio di contesto

Scheduler = POLITICHE
Dispatcher = MECCANISMI

Criteri di scheduling

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni indicatori di performance:

- **Utilizzo della** CPU: percentuale media di utilizzo CPU nell'unita' di tempo
- **Throughput** (del sistema): numero di processi completati nell'unita' di tempo
- **Tempo di attesa** (di un processo): tempo totale trascorso nella ready queue
- **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento
- **Tempo di risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround, non dipende dalla velocita' dei dispositivi di I/O)

In generale:

- devono essere **massimizzati**
  - utilizzo della CPU
  - Throughput
- Invece, devono essere **minimizzati**
- Turnaround (sistema batch)
- Tempo di attesa
- Tempo di risposta (sistemi interattivi)

Non e' possibile ottimizzare tutti i criteri contemporaneamente

A seconda del tipo di SO, gli algoritmi di scheduling possono avere diversi obiettivi

- nei sistemi batch:
  - massimizzare throughput
  - minimizzare turnaround
- nei sistemi interattivi
  - minimizzare il tempo medio di risposta dei processi
  - minimizzare il tempo di attesa

Algoritmo di scheduling FCFS

Firts-Come-First-Served: la coda dei processi pronti viene gestita in modo FIFO

- i processi sono schedulati secondo **l'ordine di arrivo** nella coda
- algoritmo **non pre-emptive** (nella versione "pura" iniziale)

Problemi dell'algoritmo FCFS

Non e' possibile influire sull'ordine dei processi:

- nel caso di processi in attesa dietro ai processi lunghi con CPU burst (processi CPU bound), il tempo di attesa e' alto
- Possibilita' di effetto convoglio, se molti processi I/O bound seguono un processo CPU bound: scarso grado di utilizzo della CPU

Algoritmo di scheduling SJF (Shortest Job First)

Per risolvere i problemi dell'algoritmo FCFS:

- per ogni processo nella ready queue, viene stimata la lunghezza del prossimo CPU-burst
- viene schedulato il processo con il CPU burst piu' corto (Shortest Job First)

SJF puo' essere:

- non pre-emptive
- pre-emptive: (Shortest Remaining Time First, SRTF) se nella coda arriva un processo (Q) con CPU burst minore della CPU burst rimasto al processo running (P) --> pre-emption

Problema: e' difficile stimare la lunghezza del prossimo CPU-burst di un processo (di solito, uso del passato per predirre il futuro

Stimare la lunghezza di CPU burst

Unica cosa ragionevole: stimare probabilisticamente la lunghezza in dipendenza dai precedenti CPU burst di quel processo.

Possibilita' molto usata: exponential averaging

Algoritmo di scheduling Round Robin

E' tipicamente usato in sistemi time sharing:

- Ready queue gestita come una coda FIFO circolare (FCFS)
- ad ogni processo viene allocata la CPU per un intervallo di tempo costante Delta t (time slice o quanto di tempo)

- il processo usa la CPU per Dt (oppure si blocca prima)
- allo scadere del quanto di tempo: prelazione della CPU e re-inserimento in coda

Round Robin (RR)

Obiettivo principale e' la minimizzazione del tempo di risorsa (adeguato per sistemi interattivi). Tutti i processi sono trattati allo stesso modo (assenza di starvation)

Problemi:

- dimensione del quanto di tempo
  - Dt piccolo (ma non troppo piccolo: Dt >> $T_{context\ switch}$) tempi di risposta ridoti, ma alta frequenza di context switch
  - Dt grande, overhead di context switch ridotto, ma tempi di risposta piu' alti
- trattamento equo dei processi
  - possibilita' di degrado delle prestazioni del SO, che deve avere maggiore importanza dei processi utente

Scheduling con priorita'

Ad ogni processo viene assegnata una priorita':

- lo scheduler seleziona il processo pronto con priorita' massima
- processi con uguale priorita' vengono trattati in modo FCFS

Priorita' possono essere definite

- **internamente**: SO attribuisce ad ogni processo una priorita' in base a politiche interne
- **esternamente**: criteri esterni al SO (es: nice in UNIX)

--> Le priorita' possono essere costanti o variare dinamicamente

In ogni istante e' in esecuzione il processo pronto **a priorita' massima** (algoritmi preemptive e non preemptive)

Problema: starvation dei processi
Starvation: si verifica quando uno o piu' processi di priorita' bassa vengono lasciati indefinitamente nella coda dei processi pronti, perche' vi e' sempre almeno un processo di priorita' piu' alta

Soluzione: invecchiamento (aging) dei processi, ad esempio

- la priorita' cresce dinamicamente con il tempo di attesa del processo
- la priorita' decresce al crescere del tempo di CPU gia' utilizzato

Approcci misti

Nei SO reali, spesso si combinano diversi algoritmi di scheduling

Esempio: **Multiple Level Feedback Queue**

- piu' code, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, ...)
- ogni coda ha una diversa priorita': scheduling delle code con priorita'
- ogni coda viene gestita con scheduling FCFS o Round Robin
- i processi possono muoversi da una coda all'altra, i base alla loro storia:
  - passaggio da priorita' bassa ad alta: processi in attesa da molto tempo (feedback positivo)
  - passaggio da priorita' alta a bassa: processi che hanno gia' utilizzato molto tempo di CPU (feedback negativo)

Esempio di Multi Level Feedback Queue

3 code

- Q_0 - RR con time quantum=8ms
- Q_1 - RR con time quantum=16ms
- Q_2 - FCFS

Scheduling

- Un processo nuovo entra in Q_0; quando acquisisce la CPU ha 8ms per utilizzarla; se non termina nel quanto di tempo viene spostato in Q_1
- In Q_1 il processo e' servito ancora RR e riceve 16ms di CPU; se non termina nel quanto di tempo, viene spostato in Q_2

--> Priorita' elevata a processi con breve uso di CPU

Scheduling in UNIX (BSD 4.3)

Obiettivo: **privilegiare i processi interattivi**

Scheduling MLFQ: Multilevel Feedback Queue Scheduling

- piu' livelli di priorita' (circa 160): piu' grande e' il valore, piu' bassa e' la priorita'
- Viene definito un valore di riferimento pzero
  - Priorita' >= pzero: processi di utente ordinari
  - Priorita' < pzero: processi di sistema (ad es. esecuzione di system call), non possono essere interrotti da segnali (kill)

- Ad ogni livello e' associata una coda, gestita Round Robin (quanto di tempo: 100ms)
- Aggiornamento dinamico delle priorita': ad ogni secondo viene ricalcolata la priorita' di ogni processo
- La priorita' di un processo decresce al crescere del tempo di CPU gia' utilizzato
    - feedback negativo
    - di solito, processi interattivi usano poco la CPU: in questo modo vengono favoriti
- L'utente puo' influire sulla priorita': comando **nice** (ovviamente soltanto per decrescere la priorita')
- Quando un processo esegue una system call, puo' venire bloccato (es.: richiesta di uso del disco)
- A verificarsi dell'evento atteso, il processo diventa pronto e viene inserito nel relativo livello di priorità
  (priorita' negativa = priorita' alta)
- Favorisce i **processi interattivi** (attesa per un terminale) e i processi che eseguono I/O

Linux scheduling (da v2.5)

Due algoritmi: **time sharing** e **real-time**

- **Time-sharing**
    - Con priorita' dinamiche, basato su crediti - processi con piu' crediti schedulati prima
    - Crediti vengono decrementati in base a timer
    - Quando crediti = 0, il processo viene deschedulato
    - Si rialza il credito di tutti quando tutti i processi arrivano a credito=0
- **Real-time**
    - Soft real-time con priorita' statiche
    - Conforme a POSIX.1b compliant - due classi
        - FCFS e RR all'interno della stessa priorita'
        - proesso a priorita' maggiore esegue sempre per primo

Scheduling dei thread Java

Java Virtual Machine (JVM) usa **scheduling con prelazione e basato su priorita'**

- FCFS tra thread con stessa priorita'

JVM mette in stato di running un thread quando:

1. il thread che sta usando la CPU esce dallo stato Runnable
2. un thread a priorita' piu' alta entra nello stato Runnable

*NB: le Java Specifications non indicano se i thread hanno un quanto di tempo oppure no. JVM puo' adottare una propria politica di scheduling oppure delegare lo scheduling dei thread al sottostante

sistema operativo

Time-Slicing

Siccome JVM non garantisce time-slicing, andrebbe usato il metodo yield() per trasferire il controllo ad altro thread di uguale priorita':

```
while (true) {
    // perform CPU-intensive task
    ...
    thread.yield();
}
```

Si possono assegnare valori di priorita' tramite metodo `setPriority()`.

10 - Gestione della memoria

Multiprogrammazione e gestione memoria

Obiettivo primario della multiprogrammazione e' l'uso efficiente delle risorse computazionali:

- efficienza nell'uso della CPU
- velocita' di risposta dei processi
- ...

Necessita di mantenere piu' processi in memoria centrale: SO deve gestire la memoria in modo da consentire la presenza contemporanea di piu' processi

Caratteristiche importanti :

- Velocita'
- Grado di multiprogrammazione
- Utilizzo della memoria
- Protezione

Gestione della memoria centrale

A livello hw: ogni sistema e' equipaggiato con un unico spazio di memoria accessibile direttamente da CPU e dispositivi

Compiti di SO

- allocare memoria ai processi
- deallocare memoria
- separare gli spazi di indirizzi associati ai processi (protezione)
- realizzare i collegamenti (binding) tra gli indirizzi logici specificati dai processi e le corrispondenti locazioni nella memoria fisica
- memoria virtuale: gestire gli spazi di indirizzi logici di dimensioni superiori allo spazio fisico

Accesso alla memoria

Memoria centrale:

- vettore di celle, ognuna univocamente indivuduata da un indirizzo
- operazioni fondamentali sulla memoria: load/store dati e istruzioni
- Indirizzi
  - **simbolici**: riferimenti a celle di memoria nei programmi in forma sorgente mediante nomi simbolici
  - **logici**: riferimenti a celle nello spazio logico di indirizzamento del processo
  - **fisici:** riferimenti assoluti delle celle in memoria a livello HW

Indirizzi simbolici, logici e fisici

Ogni processo dispone di un proprio spazio di indirizzamento logico [0, max] che viene allocato nella memoria fisica.

Binding degli indirizzi

Ad ogni indirizzo logico/simbolico viene fatto corrispondere un indirizzo fisico: l'associzione tra indirizzi relativi e indirizzi assoluti viene detta **binding**

Binding puo' essere effettuato:

- **staticamente**
  - a tempo di compilazione: il compilatore degli indirizzi assoluti (esempio: file .com DOS)
  - a tempo di caricamento: il compilatore genera degli indirizzi relativi che vengono convertiti in indirizzi assoluti dal loader (codice rilocabile)
- **dinamicamente**

    - a tempo di esecuzione: durante l'esecuzione un processo puo' essere spostato da un'area all'altra

Caricamento/collegamento dinamico

Obiettivo: ottimizzazione della memoria

Caricamento dinamico

- in alcuni casi e' possibile caricare in memoria una funzione/procedura a runtime solo quando avviene la chiamata
- loader di collegamento rilocabile: carica e collega dinamicamente la funzione al programma che la usa
- la funzione puo' essere usata da piu' processi simultaneamente. Problema di visibilita' --> compito SO e' concedere/controllare:
    - l'accesso di un processo allo spazio di un altro processo
    - l'accesso di piu' processi agli stessi indirizzi

Overlay

In generale, la memoria disponibile non puo' essere sufficiente ad acogliere codice e dati di un processo.

Soluzione a **overlay** mantiene in memoria istruzioni e dati:

- che vengono utilizzati piu' frequentemente
- che sono necessari nella fase corrente

Codice e dati di un processo vengono suddivisi in **overlay** che vengono caricati e scaricati dinamicamente (dal gestore di overlay, di solito esterno al SO)

Overlay: esempio

Assembler a 2 passi: 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)

4 componenti distinte nel codice assembler:

- Tabella dei simboli (ad es. dim 20KB)
- Sottoprogrammi comuni ai due passi (ad es. 30KB)
- Codice passo 1 (ad es. 70KB)
- Codice passo 2 (ad es. 80KB)

--> spazio richiesto per l'allocazione integrale dell’assembler e' quindi 200KB

Tecniche di allocazione memoria centrale

Come vengono allocati codice e dati dei processi in memoria centrale?

Varie tecniche

- 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 e' partizionata

- un solo processo alla volta puo' essere allocato in memoria: non c'e' multiprogrammazione

Di solito:

- SO richiede nella memoria bassa [0, max]
- necessita' di proteggere codice e dati di SO da accessi di processi utente:
  - uso del registro rilocazione (RL=max+1) per garantire la correttezza degli accessi

Allocazione contigua: partizioni multiple

Multiprogrammazione --> necessita' di proteggere codice e dati di ogni processo

Partizioni multiple: ad ogni processo caricato viene associata un'area di memoria distinta (partizione)

- partizioni fisse
- partizioni variabili

Partizioni fisse (MFT, multiprogramming with Fixed number of Tasks): dim di ogni partizione fissata a priori

- ogni partizione ha dimensione prefissata, eventualmente diversa da partizione a partizione
- quando un processo viene schedulato, SO cerca una partizione libera di dim sufficiente

Problemi:

- **frammentazione interna**: sottoutilizzo della partizione
- **grado di multiprogrammazione limitato** al numero di partizioni

- **dim massima** dello spazio di indirizzamento di un processo limitata da dim della partizione piu' estesa

Partizioni variabili

Partizioni variabili (MVT, Multiprogramming with Variable number of Tasks): ogni partizione allocata dinamicamente e dimensionata in base a dim processo da allocare

- quando un processo viene schedulato, SO cerca un'area sufficientemente grande per allocarvi dinamicamente la partizione associata

Vantaggi (rispetto a MFT):

- elimina frammentazione interna (ogni partizione e' della esatta dimansione del processo)
- grado di multiprogrammazione variabile
- dimansione massima dello spazio di indirizzamento di ogni processo limitata da dim spazio fisico

Problemi:

- scelta dell'area in cui allocare: best fit, worst fit, first fit, ...
- frammentazione esterna - man mano che si allocano nuove partizioni, la memoria libera e' sempre piu' frammentata
    - --> necessita' di compattazione periodica

Partizioni & protezione

Protezione realizzata a livello HW mediante:

- registro di rilocazione RR
- registro limite RL

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

Paginazione

Allocazione contigua a partizioni multiple: il problema principale e' la frammentazione esterna.

Allocazione non contigua --> paginazione

- eliminazione frammentazione esterna
- riduzione forte di frammentazione interna

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

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

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

Vantaggi

- Pagine logiche contigue possono essere allocate su pagine fisiche non contigue: non c'e' frammentazione esterna
- Le pagine sono di dim limitata: frammentazione interna per ogni processo limitata dalla dimensione del frame
- E' possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo

Supporto HW a Paginazione

Struttura dell'indirizzo logico (m bit):**

p	d

- p - numero di pagina logica (m-n bit)
- d - offset della cella rispetto all'inizio della pagina (n bit)

Hp: indirizzi logici *m* bit (*n* bit per offsett, e *m-n* per la pagina)

- dim massima dello spaio logico di indirizzamento --> 2^m
- dim della pagina --> 2^n
- numero di pagine --> 2^{m-n}

Struttura dell'indirizzo fisico:

f	d

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

Binding tra indirizzi logici e fisici puo' essere realizzato mediante **tabella delle pagine** (associata al processo):

- a ogni pagina logica associa la pagina fisica verso la quale e' realizzato il mapping

Realizzazione della tabella delle pagine

Problemi da affrontare

- tabella puo' essere molto grande
- traduzione (ind. logico --> ind. fisico) deve essere il piu' veloce possibile

Varie soluzioni

1. Su registri di CPU
    - accesso veloce
    - cambio di contesto pesante
    - dimensioni limitate della tabella
2. In memoria centrale: registro PageTableBaseRegister (PTBR) memorizza collocazione tabella in memoria
    - 2 accessi in memoria per ogni operazione (load, store)
3. Uso di cache: (Translation Look-aside Buffers, TLB) per velocizzare l'accesso

Translation Look-aside Buffers (TLB)

- Tabella delle pagine e' allocata in memoria centrale
- Una parte della tabella delle pagine (di solito, le pagine accedute piu' d frequente o piu' di recente) e' copiata in cache: TLB

Se la coppia (p, f) e' gia' presente in cache l'accesso e' veloce; altrimenti SO deve trasferire la coppia richiesta dalla tabella delle pagine (in memoria centrale) in TLB

Gestione TLB

- TLB inizialmente vuoto
- mentre l'esecuzione procede, viene gradualmente riempito con indirizzi pagine gia' accedute

HIT-RATIO: percentuale di volte che una pagina viene trovata in TLB

- dipende dalla dimensione del TLB (Intel486: 98%)

Paginazione & protezione

La tabella delle pagine

- ha dimensione fissa
- non sempre viene utilizzata completamente

Come distinguere gli elementi significativi da quelli non utilizzati?

- **Bit di validita'**: ogni elemento contiene un bit
    - se e' a 1, entry valida (pagina appartiene allo spazio logico del processo)
    - se e' 0, entry non valida

- **Page Table Length Register**: registro che contiene il numero degli elementi validi nella tabella delle pagine

In aggiunta, per ogni entry della tabella delle pagine, possono esserci uno o piu' nit di protezione che esprimono le modalita' di accesso alla pagina (Es. read-only)

Paginazione a piu' livelli

Lo spazio logico di indirizzamento di un processo puo' essere molto esteso:

- elevato numero di pagine
- tabella delle pagine di grandi dimensioni

Ad esempio:

- per ipotesi, indirizzi di 32 bit --> spazio logico di 4GB dimansioone pagina 4KB (2^{12})
- la tabella delle pagine dovrebbe contenere $2^{32}/2^{12}$ elementi --> 2^{20} elementi circa (1M)

Paginazione a piu' livelli: allocazione non contigua anche delle pagine --> si applica ancora la tecnica di paginazione alla tabella delle pagine!

Vantaggi:

- possibilita' di indirizzare spazi logici di dimensione elevate riducendo i problemi di allocazione delle tabelle
- possibilita' di mantenere in memoria soltanto le pagine della tabella che servono

Svantaggio:

- tempo di accesso piu' elevato: per tradurre un indirizzo logico sono necessari piu' accessi in memoria (ad esempio, 2 livelli di paginazione --> 2 accessi)

Tabella delle pagine invertita

Per limitare l'occupazione di memoria, in alcuni SO si usa un'unica struttura dati globale che ha un elemento per ogni frame: **tabella delle pagine invertita**

Ogni elemento della tabella delle pagine invertita rappresenta un frame (indirizzo pari alla posizione della tabella) e, in caso di frame allocato, contiene:

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

Per tradurre l'indirizzo logico <pid, p, d>:

- ricerca nella tabella dell'elemento che contiene la coppia (pid, p) --> l'indice dell'elemento trovato rappresenta il numero del frame allocato alla pagina logica p

Problemi:

- tempo di ricerca nella tabella invertita
- difficolta' di realizzazione della condivisione di codice tra processi (**rientranza**): come associare un frame a piu' pagine logiche di processi diversi?

Organizzazione di memoria in segmenti

La segmentazione si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti (segmenti), ognuna caratterizzata da nome e lunghezza

- Divisione semantica per funzione: ad esempio
  - codice
  - stack
  - dati
  - heap
- Non e' stabilito un ordine tra i segmenti
- Ogni segmento allocato in memoria in modo contiguo
- Ad ogni segmento SO associa un intero attraverso il quale lo si puo' riflettere

Segmentazione

Struttura degli indirizzi logici: ogni indirizzo e' costituito dalla coppia <segmento, offset>

- segmento: numero che individua il segmento nel sistema
- offset: posizione cella all'interno del segmento

Supporto HW alla segmentazione

Tabella dei segmenti: ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica mediante la coppia <base, limite>

- base: indirizzo prima cella del segmento nello spazio fisico
- limite: indica la dimensione del segmento

Realizzazione della tabella dei segmenti

Tabella globale: possibilita' di dimensioni elevate

Realizzazione

- Su registri di CPU

- In memoria, mediante registri base (Segment Table Base Register, STBR) e limite (Segment Table Length Register, STLR)
- Su cache (solo l'insieme dei segmenti usati piu' recentemente)

Segmentazione

Estensione della tecnica di allocazione a partizioni variabili

- partizioni variabili: 1 segmento/processo
- segmentazione: piu' segmenti/processo

Problema principale:

- come nel caso delle partizioni variabili, **frammentazione esterna**

Soluzione: allocazione dei segmenti con tecniche

- **best fit**
- **worst fit**
- **...**

Segmentazione paginata

Segmentazione e paginazione possono essere combinate (ad esempio Intel x86):

- spazio logico segmentato (Specialmente per motivi di protezione)
- ogni segmento suddiviso in pagine

Vantaggi:

- eliminazione della frammentazione esterna (ma introduzione di frammentazione interna ...)
- non necessario mantenere in memoria l'intero segmento, ma e' possibile caricare soltanto le pagine necessarie

Strutture dati:

- tabella dei segmenti
- una tabella delle pagine per ogni segmento

Ad esempio, segmentazione in Linux

Linux adotta una gestione della memoria basata su segmentazione paginata

Vari tipi di segmento:

- code (kernel, user)
- data (kernel, user)

- task state segments (registri dei processi per il cambio di contesto)
- ...

I segmenti sono paginati con paginazione a tre livelli

Memoria virtuale

La dimensione della memoria puo' rappresentare un vincolo importante, riguardo a

- dimensione dei processi
- grado di multiprogrammazione

Puo' essere desiderabile un sistema di gestione della memoria che:

- consenta la presenza di **piu' processi** in memoria (ad es. partizioni multiple, paginazione e segmentazione), indipendentemente dalla dimensione dello spazio disponibile
- svincoli il grado di multiprogrammazione dalla dimensione effettiva della memoria

Con le tecniche viste finora:

- l'intero spazio logico di ogni processo e' allocato in memoria
- overlay, caricamento dinamico: si possono allocare/deallocare parti dello spazio di indirizzi --> a carico del programmatore

Memoria Virtuale: e' un metodo di gestione della memoria che consente l'esecuzione di processi non completamente allocati in memoria principale

Vantaggi:

- **dimensione dello spazio logico degli indirizzi non vincolata** all'estensione della memoria
- **grado di multiprogrammazione indipendente** dalla dimensione della memoria fisica
- **efficienza**: caricamento di un processo e swapping hanno un costo piu' limitato (meno I/O)
- **astrazione**: il programmatore non deve preoccuparsi dei vincoli relativi alla dimensione della memoria

Memory Management Unit

- Nei calcolatori senza memoria virtuale, una "move reg, 1000" provoca una lettura o scrittura di un byte all'indirizzo fisico 1000
- Con la memoria virtuale l'indirizzo viene inviato dalla CPU alla MMU che trasforma l'indirizzo logico in uno fisico
- Gli indirizzi (generati usando registri base, registri indici, etc..) si chiamano indirizzi virtuali (logici) e costituiscono lo spazio degli indirizzi virtuali

Paginazione su richiesta

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

- **tutte le pagine di ogni processo possiedono una memoria di massa**; durante l'esecuzione alcune di esse vengono trasferite, all'occorrenza, in memoria centrale

Pager: modulo del SO che realizza i trasferimenti delle pagine da/verso memoria secondaria/centrale ("swapper" di pagine)

- paginazione su richiesta (o "su domanda"): pager lazy ("pigro") trasferisce in memoria centrale una pagina soltanto se ritenuta necessaria

Esecuzione di un processo puo' richiedere swap-in del processo

- **swapper**: gestisce i trasferimenti di interi processi (mem. centrale <--> mem. secondaria)
- **pager**: gestisce i trasferimenti di singole pagine

Prima di eseguire swap-in di un processo:

- pager puo' prevedere le pagine di cui (probabilmente) il processo avra' bisogno inizialmente --> caricamento

HW necessario:

- tabella delle pagine (con PTBR, PTLR, e/o TLB, ...)
- **memoria secondaria** e strutture necessarie per la sua gestione (uso di dischi veloci)

Quindi in generale, una pagina dello spazio logico di un processo:

- puo' essere **allocata in memoria centrale**
- puo' essere **in memoria secondaria**

Come distinguere i due casi?

La tabella delle pagine contiene **bit di validita'**:

- se la pagina e' presente in memoria centrale
- se e' in memoria secondaria oppure e' invalida --> interruzione al SO (**page fault**)

Trattamento page fault

Quando kernel SO riceve l'interruzione dovuta al page fault

1. Salvataggio del contesto di esecuzione del processo (registri, stato, tabella delle pagine)
2. Verifica del motivo del page fault (tramite una tabella interna al kernel)
    - riferimento **illegale (violazione delle politiche di protezione**) --> terminazione del processo
    - riferimento **legale**: la pagina in memoria e' secondaria
3. Copia della pagina in un frame libero

4. Aggiornamento della tabella delle pagine
5. Ripristino del processo: esecuzione dell'istruzione interrotta dal page fault

Paginazione su richiesta: sovrallocazione

In seguito a un page fault:

- se e' necessario caricare una pagina in memoria centrale, puo' darsi che non ci siano frame liberi

Sovrallocazione

Soluzione

--> **sostituzione** di una pagina P_{vitt} (vittima) allocata in memoria con la pagina P_{new} da caricare:

1. Individuazione della vittima P_{vitt}
2. Salvataggio di P_{vitt} su disco
3. Caricamento di P_{new} nel frame liberato
4. Aggiornamento tabelle
5. Ripresa del processo

Sostituzione di pagine

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

- per scaricare la vittima
- per caricare la pagina nuova

Pero' e' possibile che la vittima non sia stata modificata rispetto alla copia residente sul disco, ad esempio:

- pagine del codice (read-only)
- pagine contenenti dati che non sono stati modificati durante la permanenza in memoria

In questo caso la copia della vittima sul disco puo' essere evitata:
--> introduzione del **bit di modifica (dirty bit)**

Dirty bit

Per rendere piu' efficiente il trattamento del page fault in caso di sovrallocazione

- si introduce in ogni elemento della tabella delle pagine un bit di modifica (**dirty bit**)
    - se settato, la pagina ha subito almeno un aggiornamento in memoria da quando e' caricata la memoria
    - se a 0, la pagina non e' stata modificata

- algoritmo di sostituzione esamina il bit di modifica della vittima:
    - esegue swap-out della vittime solo se il dirty but e' settat

Algoritmi di sostituzione della pagina vittima

La finalita' di ogni algoritmo di sostituzione e' sostituire quella pagina la cui probabilita' di essere accedute a breve termine e' bassa

Algoritmi

- **LFU** (Leatest Frequently Used): sostituita la pagina che e' stata usata meno frequentemente (in un intervallo di tempo prefissato)
    - e' necessario associare un contatore degli accessi ad ogni pagina
    - la vittima e' quella con minimo valore del contatore
- **FIFO**: sostituita la pagina che e' da piu' tempo caricata in memoria (indipendentemente dal suo uso)
    - e' necessario memorizzare la cronologia dei caricamenti in memoria
- **LRU** (Least Recently Used): di solito preferibile per principio di localita', viene sostituita la pagina che e' stata usata meno recentemente
    - e' necessario registrare la sequenza degli accessi alle pagine in memoria
    - overhead, dovuto all'aggiornamento della sequenza degli accessi per ogni accesso in memoria

Implementazione LRU: necessario registrare la sequenza temporale degli accesssi

Soluzioni

- **Time** sharing: l'elemento della tabella delle pagine contiene un campo che rappresenta l'istante dell'ultimo accesso alla pagina
    - Costo della ricerca della pagina vittima
- **Stack**: struttura dati di tipo stack in cui ogni elemento rappresenta una pagina; l'accesso a una pagina provoca lo spostamento dell'elemento corrispondente al top dello stack --> **bottom contiene la pagina LRU**
    - gestione puo' essere costosa, ma non c'e' overhead di ricerca

Algoritmi di sostituzione: LRU approssimato

Spesso si utilizzano versioni semplificate di LRU introducendo, al posto della sequenza degli accessi, un bit di uso associato alla pagina:

- al momento del caricamento e' inizializzato a 0
- quando la pagina viene acceduta, viene settato
- periodicamente, i bit di uso vengono resettati

--> viene sostituita una pagina con bit di uso==0; il criterio di scelta, ad esempio, potrebbe inoltre considerare il dirty bit:

- tra tutte le pagine non usate di recente (bit di uso0), ne viene scelta una non aggiornata (dirty bit0)

Localita' dei programmi

Si e' osservato che un processo, in una certa fase di esecuzione:

-
  - usa solo un sottoinsieme relativamente piccolo delle sue pagine logiche
    - sottoinsieme delle pagine effettivamente utilizzate varia lentamente nel tempo
- **Localita' spaziale**
  - alta probabilita' di accedere a locazioni vicine (nello spazio logico/virtuale) a locazioni appena accedute (ad esempio, elementi di un vettore, codice sequenziale, ...)
- **Localita' temporale**
  - alta probabilita' di accesso a locazioni accedute di recente (ad esempio cicli)

Working set

In alternativa alla paginazione su domanda, tecniche di gestione della memoria che si basano su pre-paginazione:

- si prevede il seti di pagine di cui il processo da caricare ha bisogno per la proissima fase di esecuzione
  working set

Working set puo' essere individuato in base a criteri di **localita' temporale**

Dato un intero **Δ**, il working set di un processo P (nell'istante t) e' l'insieme di pagine **Δ**(t) indirizzate da P nei piu' recenti **Δ** riferimenti.
Δ definisce la "finestra" del working set

Prepaginazione con working set

- Caricamento di un processo consiste nel caricamento su working set iniziale
- SO mantiene working set di ogni processo aggiornandolo dinamicamente, in base al principio di localita' temporale:
  - all'istante t vengono mantenute le pagine usate dal processo nell'ultima finestra **Δ** (t)
  - le altre pagine (esterne a **Δ** (t)) possono essere sostituite

**
Vantaggio**

- riduzione del numero di page fault

Un esempio: gestione della memoria in UNIX (prime versioni)

In UNIX spazio logico segmentato:

- nelle prime versioni (prima di BSDv3), allocazione contigua dei segmenti
  - segmentazione pura
  - non c'era memoria virtuale
- in caso di difficolta' di allocazione dei processi --> **swapping dell'intero spazio degli indirizzi**
- **condivisione codice**: possibilita' di evitare trasferimenti di codice da memoria secondaria a memoria centrale --> minor overhead di swapping

Tecnica di **allocazione contigua** dei segmenti:

- **first fit** sia per l'allocazione in memoria centrale che in memoria secondaria (swap-out)

Problemi

- frammentazione esterna
- stretta influenza dim spazio fisico sulla gestione dei processi in multiprogrammazione
- crescita dinamica dello spazio --> possibilita' di riallocazione di processi gia' caricati in memoria

UNIX: swapping

In assenza di memoria virtuale, **swapper** ricopre un ruolo chiave per la **gestione delle contese di memoria** da parte dei diversi processi:

- periodicamente (ad esempio nelle prime versioni ogni 4s) lo swapper viene attivato per provvedere (eventualmente) a **swap-in** e **swap-out** di processi
  - **swap-out**:
    - processi inattivi (sleeping)
    - processi "ingombranti"
    - processi da piu' tempo in memoria
  - **swap-in**:
    - processi piccoli
    - processi da piu' tempo **swapped**

La gestione della memoria in UNIX (versioni attuali)

Da BSDv3 in poi:

- **segmentazione paginata**
- memoria virtuale tramite paginazione su richiesta

L'allocazione di ogni segmento non e' contigua:

- si risolve il problema della frammentazione esterna
- frammentazione interna trascurabile (pagine di dimensioni piccole)

La gestione della memoria in UNIX (versioni attuali)

Paginazione su richiesta

- **pre-paginazione**: uso dei frame liberi per per-caricare pagine non strettamente necessarie. Quando viene un page fault, se la pagina e' gia' in un frame libero, basta soltanto modificare:
  - tabella delle pagine
  - lista dei frame liberi
- **core map**: struttura dati interna al kernel che descrive lo **stato di allocazione dei frame** e che viene consultata in caso di page fault

UNIX: algoritmo di sostituzione

LRU modificato o algoritmo di seconda chance (BSDv4.3 Tahoe)
ad ogni pagina viene associato un bit di uso:

- al momento del caricamento e' inizializzato a 0
- quando la pagina viene 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

Sostituzione della vittima:

- pagina viene **resa invalida**
- frame selezionato viene inserito **nella lista dei frame liberi**
  - se c'e' **dirty bit**:
    - **solo se dirty bit=1** --> pagina va copiata in memoria secondaria
    - se non c'e' **dirty bit** --> pagina va sempre copiata in memoria secondaria

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

UNIX: sostituzione delle pagine

Scaricamento di pagine (sostituzione) attivato quando numero totale di frame liberi e' ritenuto insufficiente (minore del valore **lotsfree**)

Parametri

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

lotsfree > desfree > minfree

UNIX: scheduling, paginazione e swapping

Scheduler attiva l'algoritmo di sostituzione se

- il numero di frame liberi < lotsfree

Se il sistema di **paginazione e' sovraccarico**, ovvero:

- numero di frame liberi < minfree
- numero medio di frame liberi nell'unita' di tempo < desfree

--> scheduler attiva swapper (al massimo ogni secondo)

SO evita che pagedaemon usi piu' del 10% del tempo totale di CPU: attivazione (al massimo) ogni 250ms

Gestione della memoria in Linux

- Allocazione su **segmentazione paginata**
- Paginazione **a piu' (2 o 3) livelli**
- Allocazione contigua dei moduli di codice caricati dinamicamente
- **Memoria virtuale, senza working set**

Linux: organizzazione della memoria fisica

Alcune aree riservate a scopi specifici:

- **Area codice kernel**: pagine di quest'area sono locked (non subiscono paginazione)
- **Kernel cache**: heap del kernel (locked)
- **Area moduli gestiti dinamicamente**: allocazione mediante algoritmo buddy list (allocazione contigua dei songoli moduli)
- **Buffer cache**: gestione I/O su dispositivi a blocchi
- **Inode cache**: copia degli inode utilizzati recentemente
- **Page cache**: pagine non piu' utilizzate in attesa di sostituzione

Il resto della memoria e' utilizzato per i processi utente

Linux: spazio di indirizzamento

Ad ogni processo Linux possono essere allocati 4GB di memoria centrale (in caso di sistema a 32 bit):

- 3GB al massimo possono essere utilizzati per lo **spazio di indirizzamento virtuale**
- 1GB **riservato al kernel**, accessibile quando il processo esegue in kernel mode
- con architettura a 64 bi, spazio di indirizzamento >= 1TB, paginazione a 4 livelli

Spazio di indirizzamento di ogni processo puo' essere suddiviso in un insieme di **regioni omogenee e contigue**

- ogni regione e' costituita da una sequenza di pagine accomunate dalle stesse caratteristiche di protezione e di paginazione
- ogni pagina ha una dimensione costante (4KB su architettura Intel)

Linux: page-fetching e sostituzione

- NON viene utilizzata la tecnica del **working set**
- viene mantenuto un **insieme di pagine libere** che possano essere utilizzate dai processi (**page cache**)
- analogamente a UNIX, una volta al secondo:
  - viene controllato che ci siano **sufficienti pagine libere**
  - altrimenti, viene **liberata una pagina occupata**

11 - Input Output

Gestione dell'Input/Output

Classificazione dei dispositivi I/O

- Classificazione in base alla **sorgente/destinazione**:
  - **input**, e.g., tastiera, dischi, ...
  - **output**, e.g., video, dischi, ...
  - **rete**, e.g., IEE 802.11, BLE, Ethernet, ...
- Classificazione in base alle **modalita' di trasferimento dati**: blocchi, caratteri, speciali

    - dispositivi **a blocchi**: dati trasferiti a blocchi di dimensione fissa, e.g., dischi
    - dispositivi **a caratteri**: dati trasferiti un carattere alla volta senza alcuna struttura interna, e.g., tastiera, mouse, stampante, ...
    - dispositivi **speciali**: e.g., timer, genera interruzioni ad istanti programmati

Velocita' dei dispositivi

- Tastiera: 10 B/s
- Mouse: 100 B/s
- Modem PSTN: 7 KB/s
- Linea ISDN: 16 KB/s
- Stampante laser: 100 KB/s
- Scanner 400 KB/s
- Porta USB 1.5-1200 MB/s
- Disco IDE 5 MB/s
- CD-ROM 6 MB/s
- Fast Ethernet 12.5 MB/s
- Monitor XGA 60 MB/s
- Ethernet gigabit: 125 MB/s
- Fibra: 125 MB/s

Architettura hardware del sottosistema I/O

- La **CPU** legge e scrive i registri del controller mediante apposite istruzioni
- Il **deposito** invia e riceve le informazioni e i dati tramite i registri e il buffer del controller

Funzioni del livello dipendente dai dispositivi

Per ogni dispositivo esiste un programma **device driver** che implementa il protocollo operativo associato al dispositivo

- il device driver da parte del livello del SO devide-dependent
- le funzioni di gestione delle interruzioni generati dai dispositivi di periferici fanno parte del sottosistema di I/O devce-dependent

Organizzazione logica per la gestione dei dispositivi

- **Driver**: i driver solo la parte del sistema operativo che gestiscono i comandi
- Compito del driver e' di **inviare i comandi** appropriati ai dispositivi (al controller) e **gestire le interruzioni**
- E' la sola parte del sistema operativo che conosce i comandi del controller, il numero dei registri, etc.

Funzioni del livello indipendente dai simboli

Naming: ogni dispositivo e' identificato univocamente. In UNIX ogni dispositivo ha un nome simbolico all'internod ello spazio dei nomi del file system (si veda la directory/dev)

Buffering: aree buffer che ospitano i dati nel trasferimento tra i dispositivi e le aree di memoria dei processi applicativi.
Servono per:

1. mediare tra diverse velocita' di produzione/consumo tra processi e dispositivi
2. trasferire efficacemente dei blocchi dati
3. parallelizzare le operazioni di accesso I/O

Gestione eccezioni: nelle operazioni di I/O si possono verificare molti eventi anomali, che possono essere:

- mascherati e nascostio agli utenti (il sistema prova a completare le operazioni fallite)
- comunicati e propagati a processi e utenti

Spooling: tecnica di gestione per le risorse condivise (un processo gestore per ogni risorsa)

Input/Output a controllo di programma vs. guidato dalla interruzioni

- **A controllo di** programma: approccio sincrono, ogni processo che inizia un'operazione di I/O viene bloccato in attesa che il sistema operativo porti a termina l'operazione di I/O richiesta
- **Guidato dalle interruzioni**: approccio asincrono, il processo non si blocca ma al termine dell'operazione di I/O (per esempio lettura di un blocco di file da un disco) il controller del dispositivo lancia una **interruzione hardware** al sistema operativo che puo' quindi informatre il processo richiedente
- La gestione a interruzione evita l'inefficienza delle attese attive presente del SO nella dell'I/O eseguital al controllo di programma (polling)

Gestione degli Hard Disk

Gli Hard Disk sono dispositivi particolarmente importanti perche' offrono uno spazio di memoria di massa, utilizzato per il file system ma **anche per la memoria virtuale**

Organizzazione fisica dei dischi

Il **settore di una traccia** e' l'unita' minima di allocazione e di trasferimento (ordine di grandezza KB), identificato da:

- N. della faccia del disco
- N. della traccia (o cilindro)
- N. del settore dentro la traccia

Prestazioni Hard Disk

Le **prestazioni** di un Hard Disk sono valutate in termini di **tempo medio di trasferimento**:

TF = TA + TT
TF: Tempo medio di trasferimento
TA: Tempo medio di accesso (per posizionare testina)
TT: Tempo medio di trasferimento dati (per trasferire dati)

TA = ST + RL
ST: Seek Time, tempo per spostare longitudinalmente la testina del disco sulla traccia richiesta
RL: Rotational Time, tempo necessario per ruotare il disco in modo da leggere il settore richiesto.
Prestazioni dischi espresse in giri al minuto, tra 5.400 e 15.000

TT ordine microsecondi, ST e RL ordine millisecondi

Per ridurre tempi di accesso ai dati, progettare strategie, **politiche**, per:

- allocazione dei file (in settori se possibile contigui)
- schedulare le richieste di accesso ai dischi (per minimizzare tempi di spostamento testina)

Politiche scheduling di accesso Hard Disk

In un sistema concorrente, molti processi al file system, che si trova quindi a gestire molte richieste, che devono essere schedulate (adottano specifiche **politiche**) opportunamente per ridurre i tempi di attesa dei processi.

Dischi RAID

Per migliorare ulteriormente le **prestazioni**, si possono utilizzare in parallelo piu' dischi fissi. Questo puo' permettere anche di migliorare l'**affidabilita'** e la **tolleranza ai guasti** (tramite ridondanza dei dati)

Sistemi **RAID** ((Redundant Array of Independent Disks)

RAID livello 0 (striping)

Si crea **un solo volume** logico su tutti i dischi.

I dati sono allocati su dischi diversi, per **parallelizzare** operazioni di I/O

RAID livello 1 (mirroring)

Tutti i dati sono **replicati su due dischi.** Il sistema scrive un dato sempre su due dischi.

- Lettura puo' essere parallelizzata sui due dischi
- Possibile mirroring anche arree del sistema
- Tolleranza al guasto di un disco
- Elevato **costo** (utilizzo dischi del 50%)

RAID livello 5 (striping con parita')

- Ogni sezione di parita' contiene **l'XOR (or-esclusivo)** delle 4 sezioni dati corrispondenti
- Nel caso di perdita di UNA delle sezioni dati, il sistema ricostruisce la perdita utilizzando la sezione di parita'
- Minore costo rispetto a mirroring (in questo esempio, costo del 20%)
- Ogni scrittura richiede modifica sezione di parita'

RAID livello 6 (striping con doppia parita')

- Molto simile al RAID livello 5 ma con **un blocco di parita' aggiuntivo**: stripng dei dati su tutti i dischi con due blocchi di parita'
- Le operazioni di scrittura sono piu' costose a causa dei calcoli della parita' ma le letture non hanno svantaggi prestazionali
- Maggiore affidabilita' rispetto al RAID livello 5

Serial Advanced Technology Attachment (SATA)

L'interfaccia SATA (composta da 6 unita') permette lo scambio di dati tra un host e un device SATA attraverso un **link seriale**

- vengono raccolti dati da un host e formato un frame di informazioni attraverso la Data Processing Unit
- i dati nel frame vengono poi codificati e serializzati prima di essere trasmessi al device
- i dati ricevuti dall'interfaccia vengono de-serializzati e de-codificati attraverso un processo inverso per poi essere processati e restituiti all'host

Tutte le operazioni dell'interfaccia vengono monitorate dal Controller SATA

Unita' a stato solido (Solid State Drive - SSD)

Dispositivi **molto veloci** con **prestazioni asimmetriche** di lettura e scrittura, non contengono parti mobili.

Ache se alcuni sono conformi allo standard SATA concepito per dischi meccanici, sempre piu' SSD si interfacciano al sistema attraverso **Non-Volatile Memory express (NVMe)**

Non-Volatile Memory express (NVMe)

- **Standard di accesso** ultraveloce alle memorie non volatili che sfrutta eglio la velocita' di connessione e il parallelismo disponibile negli SSD
- Supportando l'**uso di code multiple** rende possibile **elaborare richieste in parallelo** attraverso le sue molteplici pagine e chip (in aggiunta ai tanti core a disposizione nei moderni elaboratori)
- La macchina ha bisogno di meno dispositivi per supportare lo stesso numero di operazioni I/O e inoltre riducono molto i requisiti di energia e raffreddamento
- Permette un accesso diretto al bus PCIe e all'SSD --> in NVMe sono coinvolti meno strati software rispetto alle operazioni SATA
- Offre una **coda di comandi** (submission queue) e una **coda di risposte** (completion queue) per ciascun core
- Per eseguire richieste memorizzazione un core scrive i comandi di I/O nella sua coda richieste e NVMe scrivera' in un registro chiamato **doorbell** quando i comandi sono pronti

SSD e RAID

Rispetto ai dischi magnetici, gli SSD offrono prestazioni molto migliori e una maggiore affidabilita'. C'e' ancora bisogno del RAID?

Tipicamente si, un RAID di piu' SSD puo' offrire prestazioni e affidabilita' migliori di uno singolo:

- un RAID livello 0 fornisce prestazioni di lettura e scrittura sequenziali migliori rispetto al singolo SSD
- anche RAID livello 5 e 6 sono utilizzati con SSD: migliorano prestazioni e affidabilità ma al costo di operazioni di scrittura molto intense e costose, che nel lungo periodo aumentano l'usura degli SSD

12 - Protezione

Protezione

- **Protezione**: garantire che le **risorse** di un sistema di elaborazione siano accedute solo dai **soggetti** autorizzati.
- **Risorse**: fisiche e logiche (fisiche: CPU, memoria, stampanti; logiche: file, semafori, ...)
- **Soggetti**: utenti, processi, procedure.
- Servono metodologie, modelli, strumenti per la **specifica dei controlli** e la loro **realizzazione** (enforcement)
- **Obiettivo della protezione**: assicurare che ciascun componente di programma/processo/utente attivo in un sistema usi le risorse del sistema solo **in modi consistenti** con le **politiche** stabilite per il loro uso.

- **Separazione tra politiche e meccanismi**: un sistema di protezione deve essere iin grado di realizzare una varieta' di politiche

Protezione o Sicurezza?

Protezione e Sicurezza sono due termini vicini ma diversi.

- La **protezione** serve per prevenire errori o usi scorretti da parte di procesi/utenti che operano nel sistema
- La **sicurezza** serve per difendere un sistema dagli attacchi esterni

Sicurezza

Autenticazione

Verifica l'identita' dell'utente attraverso:

- Possesso di un oggetto (es., smart card)
- Conoscenza di un segreto (password)
- Caratteristica personale fisiologica (impronta digitale, venature retina)

Problema della mutua autenticazione

Si noti che **l'autorizzazioen** (protezione) serve per specificare le azioni concesse a ogni utente.

Autenticazione ≠ Autorizzazione

- **Riservatezza**: previene la lettura non autorizzata delle informazioni (es. messaggi cifrati. Se intercettati, non rivelano comunque il contenuto)
- **Integrità:** previene la modifica non autorizzata delle informazioni (Es. un messaggio spedito dal mittente e' ricevuto tale e quale dal destinatario).
- **Disponibilità**: garantire in qualunque momento la possibilita' di usare le risorse
- **Paternità**: chi esegue un'azione non puo' negarne la paternita' (per esempio un assegno firmato)

Protezione (e least privilege)

- In qualunque momento un processo (o un utente) puo' accedere solo agli oggetti per cui e' autorizzato
- Nell'informatica moderna e' importante rispettare il principio di "**least privilege"**, cioe' in ogni istante un processo deve accedere **solo a quelle risorse strettamente necessario** per compoiere la sua funzione (in questo odo di limita il danno che un processo con errori puo' creare nel sistema).

Esempi di **least privilege**:

1. Processo **P** chiama una procedura **A. A** deve poter accedere a sue variabili e parametri formali passate, e non a tutte le variabili del processo **P**.

2. Un amministratore di sistema che deve fare il backup di tutto un file system, deve avere i diritti di lettura su tutto, non quelli di scrittura

Protezione e controllo degli accessi

- Nei sistemi operativi ci sono dei componenti incaricati di verificare che i processi possano accedere alle sole risorse per cui sono autorizzati.
- Si parla di **Reference Monitor**, come il componente del sistema che media tra le richieste di accesso dei processi e le risorse
- TUTTE le richieste di accesso passano dal Reference Monitor
- E' importante che tutte le decisioni di accesso alle risorse siano concentrate in un unico componente

Dominio di protezione

- Quali sono le soluzioni per gestire il controllo degli accessi e garantire quindi uan corretta protezione delle risorse?
- Consideriamo il **dominio di protezione**, che definisce un insieme di risorse (oggetti) e i relativi tipi di operazione (diritti di accesso) sugli oggetti stessi che sono permesse a processi (soggetti) appartenenti a tale dominio
- Per esempio, un processo opera all'interno di un **dominio di protezione** che specifica le risorse che il processo puo' usare

Il dominio e' un concetto astratto che puo' essere realizzato in una varieta' di modi:

- un dominio per ogni **utente**. L'insieme degli oggetti che l'utente puo' accedere dipende dall'**identita' dell'utente**. Il cambio di dominio e' **legato dall'identita' dell'utente** (avviene quando cambia l'utente, es. Unix)
- un dominio per ogni **processo**. Ogni riga descrive oggetti e i diritti di accesso per un processo. Il cambio di dominio corrisponde **all'invio di un messaggio** a un altro processo
- un dominio per ogni **procedura**. Il cambio del dominio corrisponde **alla chiamata di procedura**

Matrice degli accessi (modello di protezione)

oggetto / dominio	F_1	F_2	F_3	disco	stamp.
D_1	read		read		
D_2				read	print
D_3		read	execute		
D_4	read write		read write		

diritto di accesso

- **access(i,j)** definisce l'**insieme dei diritti di accesso** che un processo che opera nel dominio **i** puo' esercitare sull'oggetto **j**

- Si puo' realizzare come un **insieme ordinato** di triple <dominio, oggetto, insieme dei diritti> (tavola globale)
- Quando un'operazione **M** deve essere eseguita nel dominio $\mathbf{D_i}$ su $\mathbf{O_j}$, si cerca la tripla $\mathbf{<D_i, O_j, R_k>}$ con $\mathbf{M \in R_k}$**.** Se esiste, l'operazione puo' essere eseguita; diversamente, si ha situazione di **errore **
- Domini **statici o dinamici**
  - caso statico: l'associazione processo/dominio non puo' cambiare durante la vita del processo
  - caso dinamico: c'e' uno "switch" per permettere a un processo di cambiare dominio di protezione
- Problemi della matrice degli accessi: dimensioni matrice troppo grandi (e sparse)
- Soluzioni meno generali ma piu' efficienti e diffuse: **access control list, capability**

Access Control List (ACL)

- **Per ogni oggetto** viene indicata la coppia ordinata <dominio, insieme dei diritti>
  limitatamente ai domini con un insieme di diritti non vuoto
- Quando **deve essere eseguita** un'operazione **M** su un oggetto $\mathbf{O_j}$ nel dominio $\mathbf{D_i}$, si cerca nella lista degli accessi
  $\mathbf{<D_i, R_k>}$ con **M** e $\mathbf{R_k}$
- Se non esiste, si cerca in **una lista di "default"**; se non esiste, si ha condizione di **errore**
- Per motivi di **efficienza**, si puo' cercare prima nella lista di default e successivamente nella lista degli accessi
- Esempio: **file system**, lista degli accessi **associata al file** contiene: nome utente (il dominio) e diritti di accesso

Capability Lis

- Per ogni dominio viene indicato **l'insieme degli oggetti e dei relativi diritti di accesso** (capability list)
  D1: $<O_1$, diritti> , $<O_2$, diritti>, etc.
  D2: $<O_2$,diritti> , $<O_5$, diritti>, etc.
  etc.
- Spesso un oggetto e' identificato **dal suo nome fisico** o dal **suo indirizzo** (capability). Il possesso della capability corrisponde all'autorizzazione a eseguire una certa operazione
- Quando un processo opera in un dominio, **chiede di esercitare un diritto di accesso** su un oggetto. Se cio' e' consentito, il processo **entra in possesso di una capability** per l'oggetto e puo' eseguire l'operazione
- La lista delle capability non e' direttamente **accessibile** a un processo in esecuzione in quel dominio. E' **protetta** e **gestita** dal SO. Non puo' migrare in qualsiasi spazio direttamente accessibile a un processo utente (non puo' essere **manipolata** dai processi).