SETI (parte di Lagorio)

Introduzione ai Sistemi Operativi

26 settembre 2024

I programmi sono eseguiti dall'hardware, sono eseguiti dalla CPU.
La CPU esegue: fetch, decode ed execute

- **Fetch**: estrae dalla ram la prossima istruzione da eseguire
- **Decode**: decodifica l'istruzione letta dalla ram
- **Execute**: eseguo l'istruzione

Da che indirizzo faccio il fetch? In generale, tutte le CPU hanno un **registro**, qualcuno lo chiama Program Counter, altri Instruction Pointer (IP), che indica l'indirizzo della prossima istruzione da eseguire. Quando la CPU fa il fetch, lo farà all'indirizzo indicato dal Program Counter o dall'Instruction Pointer.

Come si fa a sapere se è successo qualcosa nel mondo esterno? **Interrupt**: ci sono uno o più **segnali di interruzione** sulla CPU, in questo modo le periferiche possono dire alla CPU che è successo qualcosa. Quindi quando viene attivata una linea di interruzione, la CPU smette di eseguire fetch, decode e execute, e darà una risposta all'Interrupt, risposta che di solito consiste nel:

- Passare in modalità privilegiata
- Eseguire **l'Interrupt Hander** (gestore delle interruzioni), che non è altro che un pezzo di codice tipicamente all'interno del sistema operativo che decide cosa fare in risposta a queste interruzioni.

Quanti programmi per volta? Se abbiamo una CPU a singolo core (esempio semplificativo), può eseguire solo un programma per volta. Un computer con una CPU da 8, 16 core, esegue molti più programmi; questo perché non sono veramente tutti contemporaneamente in esecuzione, ma danno l'impressione che lo siano. Il sistema operativo manda in esecuzione tutti i programmi per delle fettine di tempo molto piccole e quindi a noi dà l'illusione che stiano procedendo tutti in parallelo. Quindi ogni **core** esprime quanti programmi una CPU può eseguire contemporaneamente.

Il software che gestisce tutto è il **sistema operativo**: gestisce l'hardware, la CPU e le periferiche, e assegna a ciascun programma che vogliamo mandare in esecuzione una porzione di CPU per eseguire un ciclo per dare l'illusione di avere più programmi in esecuzione in parallelo.
Un **programma** è un concetto statico, un programma è un **insieme di istruzioni**; quando lo mandiamo in esecuzione lo chiamiamo **processo**.

La CPU fa il fetch dalla ram, quindi vuol dire che i programmi per essere eseguiti devono stare in ram, finchè un programma rimane in disco non può essere eseguito.
Chi porta i programmi in ram? Il sistema operativo.
Il sistema operativo rende **isolati** i processi, quindi quando scrivo un programma non mi preoccupo probabilmente degli ambienti circostanti (se ci sono altri processi in esecuzione, qual è l'hardware, qual è la scheda grafica, ecc.), perché il programma si appoggia a librerie, che si appoggiano al sistema operativo ed è il sistema operativo che si preoccupa di parlare con le periferiche, ecc.

Se un nostro programma ha un puntatore che gira per la memoria dove non dovrebbe, gli altri programmi in esecuzione non sono "in pericolo" perché ogni processo ha l'illusione che tutta la memoria sia per lui; questa astrazione si chiama **spazio di indirizzamento** ed è un'astrazione della ram.
Il termine sistema operativo è un termine che può indicare cose diverse a seconda del contesto: in genere quando si parla di sistema operativo si intende un **nucleo**, un **kernel**, e poi anche tutte le utility che ci stanno intorno, l'ambiente grafico, ecc.
Il kernel del sistema operativo è lo **strato più basso**, quello che si occupa di gestire l'hardware (ad esempio in uno smartphone android, sotto ha un kernel linux).
Il sistema operativo si occupa di gestire e virtualizzare le risorse.
I programmi non parlano direttamente con l'hardware, ci pensa direttamente il sistema operativo.

Ogni hardware ha delle regole diverse, per cui chi produce l'hardware produce anche i **driver**, che sono **pezzetti di software che vanno ad inserirsi nel sistema operativo** e gli insegnano a dialogare con quel particolare hardware. Se inserisco un nuovo hardware, se il sistema operativo non ha il drive per il nuovo hardware, non lo vede. Al giorno d'oggi, quando collego qualcosa di nuovo, il sistema è collegato ad internet e scarica dalla rete i driver.
Sono necessari ma sono anche una **minaccia**, perché girano con **gli stessi privilegi del sistema operativo** (per questioni di efficienza, non si può fare altrimenti); quindi, se c'è una vulnerabilità in un driver, questo inficia completamente la sicurezza di tutto il sistema operativo.

Per usare il manuale si usa: **man *cosa stiamo cercando*** (esempio: man atoi)

pid: ogni processo viene identificato all'interno del sistema da un pid (process identifier), ed è un intero non negativo che identifica un particolare programma in esecuzione. In ogni istante, i pid sono tutti diversi, ma in momenti diversi lo stesso numero può essere usato da processi diversi (esempio: lancio un processo con pid 10, il processo finisce; quando lancio un nuovo processo, il pid può essere 10).

Che istruzioni macchina può eseguire un processo? Può eseguire quasi tutte le istruzioni macchina della CPU a parte quelle che sono disponibili solo nella modalità privilegiata.
Perché serve avere una modalità utente e una modalità privilegiata? Serve avere queste due modalità perché **altrimenti non ci sarebbe modo di isolare e proteggere i processi**. Se ogni processo potesse eseguire qualsiasi istruzione, potrebbe per esempio cambiarsi la tabella delle pagine e andare a scrivere ovunque nella memoria fisica. Se un processo potesse parlare direttamente con l'unità disco non potrei creare un file leggibile solo da me, privato (in un sistema multiutente).
Per poter implementare questa protezione in maniera efficiente, i processori, le CPU, devono avere almeno due modalità di esecuzione diverse:

- Una **modalità privilegiata,** spesso chiamata **kernel**, perché è la modalità usata dal kernel del sistema operativo; in quella modalità si possono fare tutte le istruzioni della CPU, parlare con l'hardware;
- Una **modalità non privilegiata**, o **utente**, che viene utilizzata per far girare i processi utente.

Un processo che gira in modalità non privilegiata deve avere un modo per chiedere al sistema operativo di usare l'hardware, un'istruzione macchina che fa passare la modalità di esecuzione da livello utente a livello kernel: si usano le **chiamate di sistema** (system call).
Quando il sistema operativo ha finito di fare l'azione che gli è stata chiesta, riprenderà l'esecuzione del programma che ha fatto la system call dal punto in cui l'ha chiamata ma tornando in modalità utente.

Le system call sono **macroistruzioni** che poi corrispondono ad un sacco di istruzioni macchina che vengono eseguite dentro al kernel dal sistema operativo. Tutti i sistemi operativi hanno centinaia di system call che permettono ad un processo di lavorare, sempre tramite il sistema operativo.
Il sistema operativo, in particolare il **kernel**, è una **parte del software estremamente delicata**; dev'essere particolarmente **efficiente**, ma deve essere anche particolarmente **sicura** perché un bug all'interno del sistema operativo compromette la sicurezza dell'intero sistema.

Come siamo arrivati ai sistemi operativi moderni? I primissimi sistemi operativi erano semplicemente delle librerie, uno **strato di libreria** che permetteva un programma utente di leggere/scrivere. Se non mettiamo una protezione tra codice del sistema operativo e codice utente, possiamo avere problemi. Con l'avvento dei **minicomputer** (grandi quanto una lavatrice), si potevano caricare più programmi in memoria perché ne avevano di più, ed è nata l'idea della **multiprogrammazione**: mentre un programma aspetta un input o un output, noi possiamo usare la CPU per eseguire un altro programma che al momento non ha bisogno di input/output.

Unix è un sistema operativo nato negli anni 60, al tempo stesso è **semplice**, ma molto **potente**. È scritto principalmente in **C** con alcune parti in **Assembler**. Ad oggi esistono molti sistemi operativi **Unix-like**.

Shell e terminali

1 ottobre 2024

Nei sistemi Unix-like, la **shell** è un **interprete di comandi**, che si può utilizzare sia in modalità interattiva che negli script. Per usare il kernel servono dei programmi, quindi l'utente per **interagire indirettamente con il kernel** usa una shell. Kernel è il nucleo è shell il guscio intorno.
La shell non ha nessun privilegio in più di qualsiasi altro programma che lanciamo. Si interagisce con la shell **tramite terminale**.
Che cos'è un terminale? Un terminale è un'evoluzione della telescrivente, quindi il terminale fisico è composto da una tastiera, per mandare i comandi al calcolatore, e un dispositivo di output che ci permette di vedere il risultato delle cose che chiediamo al calcolatore. In Unix, quasi tutto è un file, e file speciali corrispondono ai terminali collegati. Oggi chiamiamo questi terminali TTY perché un tempo era l'acronimo di tele type.
Quando apro un terminale, che è una finestra che gira nell'ambiente grafico, collegato a questa finestra c'è una **bash** e se uso il comando tty, mi ritorna il file che corrisponde a questa finestra.

Su linux ci sono due concetti simili:

- Le virtual console, dei terminali che condividono tastiera e schermo, e si può passare da una all'altro usando una combinazione di tasti. Ormai non sono quasi più utilizzate.
- Emulatori di terminali, ossia dei programmi che emulano l'hardware (la finestrella da cui possiamo interagire). Il collegamento fra l'emulatore di terminale e un'istanza della shell avviene tramite l'uso di pseudo-terminali (pty).

Che relazione c'è tra i processi e i terminali?
Ogni processo ha tre **file descriptor**, un numero intero che rappresenta un file aperto per il processo:

- 0: standard input (cin)
- 1: standard output (cout)
- 2: standard error (cerr)

Comando **echo**: scrive su standard output gli argomenti che passiamo su riga di comando (esempio: echo Ciao, output: Ciao)

Il **file system** è un **albero** composto da **directory**, che può contenere altre directory e così via. Ci sono delle directory standard, e in ogni sistema Unix si trovano queste directory:

- / radice
- /bin e /sbin comandi essenziali e quelli per l'amministrazione
- /boot file per il boot del sistema
- /dev file speciali che corrispondono a dispositivi
- /etc file di configurazione del sistema
- /home e /root home degli utenti (non root) e root (ogni utente ha una cartella)
- /lib* librerie
- /media e /mnt mount-point per i media rimovibili e altri FS (quando in un sistema unix vogliamo usare un altro disco, una usb, ecc., per poterne vedere il contenuto dobbiamo montare il file system della usb in un nodo dell'albero che abbiamo già, ossia l'albero del file system. In windows invece gli alberi sono diversi, con C primo disco per windows)
- /proc e /sys FS virtuali, interfaccia alle strutture dati del kernel
- /tmp file temporanei, spesso un ramdisk nei sistemi moderni
- /usr gerarchia secondaria (/usr/[s]bin, /usr/lib*, …), che può essere condivisa in sola lettura fra più host

Tra i terminali e i processi c'è la **disciplina di linea**, ossia che quando un utente legge da standard input, quello che gli arriva è semplicemente il risultato finale (dopo l'invio).

Cosa fa la shell? In modalità interattiva, stampa un **prompt di comandi** (ossia una sequenza di caratteri che indica che è in attesa di comandi). La bash legge una linea di input, dopodiché:

- separa in **token**, riconosce le parole e gli operatori;
- espande gli **alias**, delle stringhe che vengono espanse in qualcosa di più elaborato;
- fa il **parsing** di comandi semplici e comandi composti;
- fa delle **espansioni** (ad esempio se scrivo a*, verrà poi sostituito da tutti i nomi di file che iniziano con a);
- dopodiché esegue le varie **redirezioni** di input e output;
- dopo aver fatto tutte le espansioni e le redirezioni, va davvero ad **eseguire il comando** che abbiamo chiesto. I **comandi** possono essere **built-in**, ossia implementati direttamente dalla bash, e tutti gli altri, ossia **comandi esterni**.

Comando **help**: cercare aiuto nei comandi built-in
Comando **man**: cercare aiuto nei comandi esterni

Alcuni caratteri sono speciali (per esempio, lo spazio) e devono essere messi fra **virgolette** o preceduti da un **backslash** per essere utilizzati.
Le variabili: Possono essere create o aggiornate scrivendo **name=value**
Possono essere lette scrivendo **$varname** o **${varname}**

3 ottobre 2024

La shell è un interprete di comandi, molto spesso usata in maniera interattiva. Alcuni caratteri che digitiamo nella shell rimangono uguali mentre altri vengono espansi:

- **$var** , **${var}** = la shell espande quel nome nel contenuto della variabile.
- Le graffe possono essere usate anche per inserire delle stringhe all'interno di altre stringhe. Esempi:

  comando: echo a{b, c, d} *output:* ab ac ad
  comando: mv pippo{,.txt} = mv pippo pippo.txt
- Usando la tilde (~) o tilde + nome utente -> ti porta nella home directory (tua o dell'utente di cui si scrive il nome).
- Se io voglio identificare tutti i file che hanno per esempio estensione .c nella directory, faccio "*.c". L'asterisco sta per un numero arbitrario di caratteri qualsiasi, quindi nell'esempio "qualsiasi cosa che finisca in .c" (altro esempio a*, tutti gli argomenti che iniziano con a).
- (espressioni) = $((5 + 2)) = 7 per far fare conti alla shell
- $(cmd) = l'output di quel comando (cmd) diventa l'argomento di qualcos'altro (esempio lol=404, ./arg $(lol)=404)

Redirezione di input e output:
Se io voglio che lo standard input di un comando non sia il terminale ma un file faccio **<nomefile** e quindi quello che il programma scrive verso lo standard output va a finire dentro al file;
>nomefile o **>>nomefile** apre il file in append, quindi, se il file aveva già un contenuto, aggiunge al contenuto che c'era già.
Alcuni file utili sono:

- **/dev/null** = file in cui si può scrivere tutto quello che si vuole e poi viene eliminato (per ignorare un output ad esempio)
- **/dev/zero** = sfila infinita di byte a 0, se ad esempio devo produrre degli input per un comando
- **Yes** = dice y all'infinito

I comandi possono essere:

- **Semplici** = una sequenza di parole, separata da blank, dove non compare | (pipe)
- **Pipeline** = sequenza di comandi messi in pipe, ossia separati da | o |&
- **Liste** = sequenze di pipeline, separati da ;, &, && o ||
  - Possono essere racchiuse fra ( ) o { }, per applicare un'unica redirezione a tutti i comandi nella lista
  -

&& e || usati per comporre pipelines
a && b oppure a || b vale la valutazione cosiddetta cortocircuitata, ossia in alcuni casi non serve valutare entrambi gli operandi per sapere i risultati degli and (&&) o degli or (||): se a è andato a buon fine, allora valuto anche b, se non è andato a buon fine non ha senso valutare b.

Ogni comando quando non viene ucciso da un segnale, quindi esce normalmente, restituisce un exit status, che finisce in $? e un codice numerico:

0 = senza errori non-0 = errori

Nei contesti booleani zero = true e non-zero = false (!!!).
Quando un comando va a buon fine restituisce 0, quindi è l'equivalente di true.

Lo status di uscita di un comando può essere recuperato tramite una variabile **$?** (esempio echo $?).
L'exit status è il valore restituito dalla funzione main oppure il valore che voi passate alla funzione exit.
Esempio:

int main () { exit(42); } *output di echo $?:* 42

Exit status in un programma terminato da un segnale
ctrl + c = termina segnale, invia al comando un segnale sigint (2) gli altri segnali danno una possibilità al comando di "ignorare" il segnale (può essere catturato), il sigkill (9) no. l'exit status di un processo terminato da un segnale è 128 + s (segnale).
Kill è una system call ma anche un comando che serve a mandare segnali; quindi, non uccide necessariamente ma manda un segnale.

la besh è un linguaggio di programmazione, quindi è possibile eseguire cicli, fare degli if, definire funzioni, ecc:

- Gli argomenti sono $1, $2, ecc. Per sapere quanti argomenti sono stati passati si usa $#;
- si può usare **return**;
- **local** definisce variabili locali;
- la sintassi è abbreviata.
-

Fork-bomb: : ( ) { : | : & } ; : Blocca il pc, creando processi su processi e non si riesce a killarlo.

È possibile gestire quelli che la shell chiama lavori (job) con uno stesso terminale. È possibile mandare in background un lavoro e poterlo rimandare in foreground per usarlo. Quando si lancia un comando con **ctrl z** si può sospendere, con **pg** mandarlo in background, e con **fg** riportarlo in foreground. Ci può essere un solo programma in foreground che è quello con cui interagisce il terminale e ci possono essere tutti quelli che si vogliono in background, che possono scrivere nel terminale ma non possono leggere (un processo per volta può leggere, ossia quello in foreground).

File e processi

14 ottobre 2024

Un processo non può interagire direttamente con l'hardware, ma per farlo deve utilizzare delle chiamate di sistema, che delegano al nucleo la comunicazione con l'hardware. Non si può effettuare una chiamata di sistema in standard C, è necessario usare assembly, ma, con la libreria C, si verrà a chiamare una funzione C detta **wrapper**, che sta intorno alla system call vera e propria, e offre l'interfaccia C per effettuare una system call. Vengono implementate come una normale funzione in C, prendono i parametri e poi, a seconda della convenzione del sistema operativo, preparano i registri ed effettuano la vera e propria system call. A seconda del sistema e della piattaforma hardware sottostante, le convenzioni di chiamata possono essere diverse, tendenzialmente però:

- il sistema assegna ad ogni system call un numero che le identifica, poi utilizza dei registri per passare i vari parametri;
- dopodiché, viene eseguita un'istruzione speciale, che tipicamente genera una trap a livello di processore e fa sì che l'esecuzione passi a modalità kernel;
- il kernel controlla la validità della system call (quindi controlla se il numero è esistente), esegue la richiesta controllando che i parametri scritti siano giusti, e scrive il risultato in un registro;
- torna in modalità utente.

Se una system call fallisce? Se una system call fallisce, la ragione viene codificata all'interno di una variabile globale chiamata errno. La funzione wrapper controlla il risultato della funzione: se ha successo ritornerà un valore maggiore o uguale a 0, in caso di insuccesso -1; **solo la ragione del fallimento viene ritornata in errno**. Ogni thread ha una sua variabile errno (quindi non è una variabile globale).
Una system call è quindi molto più costosa di una normale chiamata a funzione.
Quando si chiama una funzione si deve sempre controllare il valore di ritorno, per sapere se la funzione è andata a buon fine o meno. Se qualcosa fallisce e non si guarda il valore di ritorno, non si saprà mai se è andata a buon fine o fallita.
Se si vuole stampare un codice di errore per l'utente che traduca il valore intero, si può usare **perror**, ossia print error, oppure **strenerror_r**, che traduce da int a string l'errore con una frase decisa da noi in precedenza. L' _r indica che la funzione è rientrante, ossia safe per il threading, ossia flussi di esecuzione in parallelo (che non devono accedere alla stessa risorsa).

Le system call per portabilità restituiscono dei tipi che molto spesso sono degli alias a tipi esistenti (in C un alias si crea con typedef).

Tutto l'input/output avviene tramite file descriptors, quindi le system call che scrivono qualcosa da qualche parte, sapranno dove scrivere perché glielo comunicheremo tramite file descriptor. Se io voglio scrivere qualcosa, ci sono tre file speciali per convenzione:

- 0 = standart input;
- 1 = standard output;
- 2 = standard error.

Se voglio scrivere in un file, per prima cosa devo aprire il file (con system call open).
La open può avere almeno due parametri, e in base a cosa c'è scritto nel secondo parametro, potrebbe servire specificare anche il terzo parametro.

int open (const char *pathname, int flags[, mode_t mode]);

Il primissimo parametro pathname è la stringa che rappresenta il percorso del file che vogliamo leggere o scrivere; il secondo parametro flags dice cosa vogliamo fare con file, ossia se vogliamo aprirlo in lettura, scrittura, letturascrittura, ecc. È importante aprire il file chiedendo il minimo che ci basta a fare il nostro lavoro. Questo parametro flag è un intero ed è detto **bitmask**, vuol dire che non è importante il valore completo ma i singoli insiemi di bit possono significare qualcosa. Se il secondo parametro (flags) specifica che vuole creare un file nuovo, con il flag create o createfile, allora bisogna specificare un terzo parametro che rappresenta la bitmask dei **permessi del file**.
Sotto unix ogni file ha i permessi relativi a tre categorie di utenti: il proprietario del file, il gruppo a cui appartiene il file e tutti gli altri utenti del sistema. Per ognuno di questi insiemi di utenti, può specificare 3 bit che sono r (lettura), w (scrittura) e x (esecuzione). Questi permessi sono spesso codificati in ottale.
Per cambiare i permessi esiste il comando **chmod**, mentre per cambiare proprietario **chown** (ma solo se si è in root). Root (amministratore di sistema) può leggere, scrivere ed eseguire (purché ci sia almeno un bit x settato) qualsiasi file, non si fanno controlli sui permessi.

un problema dei programmi in c e c++ è il memory leak: si può usare valgrind o address sanitizer.
ctrl-c = uccide il processo, quindi se si stanno controllando gli errori, non vengono correttamente segnalati; **ctrl-z** = sospende un processo, non lo termina.

Come si legge o scrive? Con le chiamate read e write.

ssize_t read (int fd, void *buf, size_t count);

sssize_t write (int fd, const void *buf, size_t count);

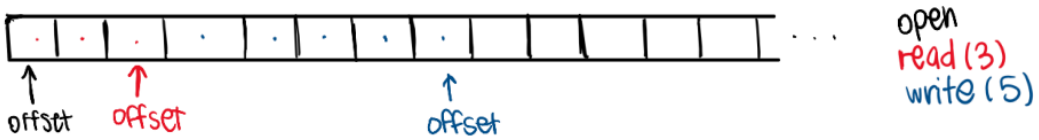
Il primo argomento è fd (file descriptor) che dice su cosa voglio leggere o scrivere, che può essere file, una connessione di rete, ecc. Il parametro buf è l'indirizzo del buffer dove leggere o dove scrivere e count è il numero di byte da leggere o scrivere.
Se falliscono ritornano -1, se non falliscono, ritornano quanti byte hanno letto o scritto (può leggerne meno, non è un errore).

21 ottobre 2024

File offset:
Per il sistema operativo, un file è semplicemente una **sequenza di byte**. Il sistema operativo non cerca in nessun modo di interpretare quei byte. I file regolari (jpg, zip, pdf, ecc.) dal punto di vista del sistema operativo sono tutti uguali; poi, se questa sequenza di byte rispetta un **formato di file** (esempio jpg, per essere un'immagine jpeg significa che dev'esserci un particolare header, l'immagine dev'essere codificata in un certo modo, ecc.), poi posso usare un programma che legge il formato e visualizzarlo. Spesso si usa un'estensione standard per etichettare il formato del file, però l'estensione può essere di default (.jpg), ma l'estensione non significa nulla (posso avere un eseguibile con .jpg, ma posso chiamarlo con system call ed eseguirlo come eseguibile).

Un file regolare è una sequenza di byte, che può essere anche molto lunga, e per questa ragione è molto comodo il fatto che abbiamo **accesso casuale** a quei byte (come gli array, quindi un file è più simile ad un array che ad una lista linkata). Per questa ragione, quando apriamo un file regolare, il sistema operativo associa al file aperto un offset (puntatore), che indica in che posizione leggere o scrivere.

Ad esempio, se abbiamo un file fatto di tantissimi byte, non appena facciamo la open, il suo offset sarà 0, ossia "non ho ancora letto né scritto niente". Se gli dico "voglio leggere 3 byte", la read andrà a leggere i primi 3 byte del file, restituirà il contenuto dei primi 3 byte e sposterà l'offset puntatore di 3 posizioni. Se facciamo un'altra read o una write nello stesso file descriptor, questa azione inizierà ad agire partendo dall'offset, per poi spostarlo in avanti. Se volessi leggere gli ultimi 10 byte del file, potrei spostare l'offset e metterla in fondo al file, poi fare una read di 10.



Il fatto di spostare l'offset ha un costo costante, indipendente dalla lunghezza del file, perché non mi devo davvero muovere.

In Unix tutto è un file; alcuni di questi file **non sono file regolari** e quindi **l'offset non può essere usato**. Ad esempio, se ho un file descriptor che corrisponde a uno stream tcp, non posso andare avanti o indietro, perché significherebbe andare avanti o indietro nel tempo.

In generale, la system call **lseek** si usa per spostare il puntatore offset, nei file che funziona.
La funzione lseek prende un file descriptor, un offset e una costante che gli dice rispetto a cosa indichiamo la nuova posizione.

off_t lseek (int fd, off_t offset, int whence);

(esempio: se scrivo (fd, 46, SEEK_SET), l'azione parte dal byte 46)
L'offset può essere positivo o negativo in base se voglio andare avanti o indietro.
Whence può essere:

- SEEK_SET (inizio del file)
- SEEK_CUR (posizione corrente)
- SEEK_END (fine del file)

questo cursore può essere spostato anche **"oltre" la fine del file**. Se cerco di leggere fallisce, ma se ci scrivo funziona e quindi creo un file "con dei buchi", riempiendo tutto lo spazio che ho saltato con degli zeri, che però non occupano spazio sul disco.

I **metadati** di un file sono delle **informazioni riguardo al file**, ad esempio come si chiama, quanto è lungo, quando è stato modificato per l'ultima volta, quando l'ho creato, ecc. Quasi tutti i metadati in un file system alla Unix vengono memorizzati in una struttura data che si chiama enode.
Ci sono delle system call, stat, lstat e fstat, che permettono di leggere i metadati di un file; in particolare vanno ad inizializzare una struttura chiamata **struct stat** che dà le informazioni sul file.

```
struct stat {
        dev_t     st_dev;         // major (12 bits) + minor (20 bits)
        ino_t     st_ino;         // Inode number
        mode_t    st_mode;        // File type and mode
        nlink_t   st_nlink;       // Number of hard links
        uid_t     st_uid;         // User ID of owner
        gid_t     st_gid;         // Group ID of owner
        dev_t     st_rdev;        // Device ID (if special file)
        off_t     st_size;        // Total size, in bytes
        blksize_t st_blksize;     // Block size for filesystem I/O
        blkcnt_t  st_blocks;      // Number of 512B blocks allocated
        struct timespec st_atim;  // Time of last access
        struct timespec st_mtim;  // Time of last modification
        struct timespec st_ctim;  // Time of last status change
}
```

Che differenza c'è tra le tre system call? Stat e lstat prendono come parametro il percorso sottoforma di stringa, ma la seconda non segue i link simbolici, ossia un tipo di file il cui contenuto punta a qualcos'altro, ad un percorso. Fstat al posto di prendere un percorso come parametro, prende un file descriptor; quindi, prima si deve aprire il file e poi chiedere i dati di quel file.

Esistono più filesystem diversi, e filesystem è il modo in cui vengono memorizzati i dati sul disco.
Ogni filesystem ha dei limiti, come, per esempio, la lunghezza massima di un nome di un file.

Attenzione: esistono delle costanti _POSIX_qualocosa che, nonostante abbiano "MAX" nel nome, corrispondono alla misura minima che deve essere garantita. Per esempio, se _POSIX_NAME_MAX è 14, il nome non può essere minore di 14. La lunghezza massima di un nome di un file si scopre chiedendo al sistema tramite getconf (esempio: getconf NAME_MAX).

Che cosa sono i programmi binari?

Quando abbiamo un sorgente in C o C++, per poterlo eseguire devo compilarlo; questo perché i processori non hanno idea di cosa sia C, C++. Il linguaggio che parla il processore è il codice macchina, e ogni processore ha il suo codice macchina. Quindi il compilatore prende il nostro codice ad alto livello e lo traduce nel codice macchina della piattaforma target.

- Compilatori e assemblatori sono quelli che producono gli **object file**, detti anche file oggetto o file rilocabili: sono essenzialmente del codice macchina con dei buchi e con dei metadati.
- Ld, il **link editor** (linker statico), mette insieme file oggetto e librerie, eseguendo rilocazione e risoluzione dei simboli.

Quando il **linker** mette insieme i pezzi, crea un **file eseguibile** dove tutto il codice necessario per l'esecuzione è dentro a quel file. Quindi il file conterrà il nostro codice e parti della libreria C che servono per l'esecuzione del nostro codice.

Ha alcuni grossi **svantaggi**: uno svantaggio è che maggior parte dei file e dei programmi usa la libreria C, quindi quasi ogni eseguibile ha la sua copia delle funzioni principali in C, e questo occupa spazio disco e di ram; sono più **pesanti** perché oltre al nostro codice hanno appunto anche il codice di libreria.

L'approccio più moderno è quello di avere quindi un **linking dinamico**. C'è sempre una parte di linking statico dove vengono controllate le cose (se le cose sono definite, ecc.), però le librerie non vengono copiate nell'eseguibile; nell'eseguibile vengono solo scritti dei metadati che dicono quali sono le **dipendenze di quell'eseguibile**.

Che vantaggi ho con il linking dinamico? I file, i programmi su disco sono molto più **piccoli**, perché non ho copie della libreria. Anche quando lancio l'eseguibile esisterà un'unica copia della libreria C e varie copie logiche all'interno di vari processi (paginazione), quindi fa risparmiare spazio in ram. Facilita **l'aggiornamento**: se io sostituisco l'implementazione di una libreria dinamica con una nuova versione, tutti i programmi che utilizzavano la vecchia versione mandati in esecuzione da quel momento in poi automaticamente useranno la versione nuova. L'unico **svantaggio** è il fatto che un programma non è il solo contenuto; quindi, se il mio programma dipende dalla libreria C allora posso spostarlo in qualsiasi macchina in quanto qualsiasi sistema ha la libreria C. Se il mio programma dipendeva però da una libreria che non è installata di default, se lo sposto in un'altra macchina dev'essere installata anche lì la libreria o non funzionerà.

24 ottobre 2024

Cosa contiene un eseguibile? Dentro ad un eseguibile ci sono:

- Il **codice macchina** corrispondente al codice sorgente che abbiamo compilato, che finisce in una sezione chiamata .text;
- I **dati**: variabili globali inizializzate, oppure variabili globali non inizializzate e la memoria allocata per loro; finiscono in .data e .rodata (read only data, quindi costanti che non possono essere modificate);
- I **metadati** che dicono al sistema l'architettura per cui l'eseguibile è stato compilato. Il formato degli eseguibili si chiama ELF (executable and linkable format), e gli ELF per i vari tipi di architettura seguono lo stesso formato, ma sono diversi per ogni tipo di architettura e quindi il sistema operativo non è in grado di eseguirlo.

Queste sezioni vengono raggruppate in **segmenti**.

Com'è fatto un file ELF?
Un file ELF ha un'intestazione, un header (che dice l'architettura, l'entrypoint, ecc), conterrà il codice, i dati e altro. Se voglio mandarlo in esecuzione, dovrò **mappare** in memoria il codice e i dati. Mappare in memoria vuol dire che il sistema operativo terrà traccia del fatto che ci saranno una serie di pagine in memoria che corrispondono a parti di questo file, come con i dati. Il codice e i dati hanno però una differenza sostanziale: mentre un programma gira i dati cambiano, mentre il codice no. Quindi la parte di **codice** può essere mappata in **modalità solo lettura** in memoria fisica, mentre la parte dei **data** non può essere mappata in read only. Un altro vantaggio di mappare il codice in sola lettura: se abbiamo più istanze dello stesso programma, il codice dei processi che devono essere isolati è sempre lo stesso; per tanto quelle pagine, quei page frame possono essere condivisi da processi diversi e mappato ad indirizzi diversi per ogni processo.

Cosa ci serve per far girare i programmi? Cosa serve avere in memoria per eseguire un programma? Servono sicuramente il **codice e dati**, e il codice e i dati delle **librerie** da cui dipende. Serve inoltre un'area **heap**, che contiene la memoria dinamica (ossia cose che non sono scritte nell'ELF, ad esempio se faccio malloc). Ci dev'essere un'area di memoria **stack**, che cresce dinamicamente man mano che si chiama la funzione ricorsivamente; serve anche per allocare le variabili locali della funzione. Il **kernel mappato** nello spazio di indirizzamento, potrebbe non essere visto dal programma utente, però la tabella delle pagine deve esserci, per rispondere all'interrupt. Non è accessibile in modalità utente.
Abbiamo alcune regioni di memoria che hanno dimensione fissa, quindi ad esempio il codice, le variabili globali hanno una dimensione fissa, allocata a priori; ci sono alcune aree che crescono con l'esecuzione del programma, ossia l'heap e lo stack. Per ragioni storiche, lo stack viene messo in cima agli indirizzi e lo heap in basso, crescendo in direzione opposta finchè non si incrociano. Lo stack cresce ad indirizzi decrescenti, mentre l'heap cresce ad indirizzi crescenti.

Ci sono delle system call che ci dicono qual è il **PID** (identificatore del processo chiamante) e **PPID** (parent pid).

pid_t getpid(void) **pid_t getppid(void)**

Tutti i processi, tranne uno speciale ossia init, vengono creati con una system call chiamata fork, e si dice che sono figli del processo che li ha creati, quindi è una gerarchia ad albero. Il punto di inizio è **init** e ha il PID=**1**. Quando parte il sistema, quando fa boot il kernel fa partire un processo che è appunto init, e da lì poi lui fa partire tutti gli altri.
ATTENZIONE: I PID identificano i processi in un determinato momento ma poi vengono riutilizzati.
Le informazioni dei processi vengono esposte in user mode tramite lo pseudo-filesystem **/proc**. Pseudo-filesystem perché non rappresenta dei file che stanno davvero su disco ma sono dei file finti in cui quando noi andiamo a leggere da quei file in realtà stiamo chiedendo al kernel le informazioni rispetto a quel processo.
C'è una directory per ogni processo e dentro a questa directory avete tante informazioni come il numero di file descriptor aperti. La maggior parte di questi file sono a sola lettura.
Ogni processo ha due directory:

- Una **directory root**, che è quella che viene utilizzata tutte le volte che si usiamo un percorso assoluto (ossia quelli che iniziano con /). Tendenzialmente la directory root è la root del file system, però non è necessariamente così (ci sono delle system call che possono cambiare cosa significa /);

- Una **directory di lavoro**, una directory corrente, che viene usata per i percorsi relativi (chdir per cambiarla).

Le system call per gestire i processi sono quattro:
- **Fork**, che crea un nuovo processo. È l'unica system call che crea un processo;
- **_exit** (exit funzione di libreria) termina il processo chiamante;
- **Wait** serve ad aspettare la terminazione di un processo figlio;
- **Execve** serve ad eseguire un programma. Sostituisce completamente lo spazio di indirizzamento del processo che lo invoca; quindi, molte volte vedremo eseguita un execve dopo una fork, perché la fork copia il processo, che poi eseguirà l'execve (exec e exec* funzioni di libreria).

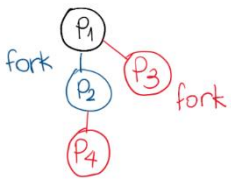
La system call fork **clona** un processo, quindi crea un **processo figlio** che è una copia del processo che ha evocato la fork. Init è l'unico processo speciale che non viene creato tramite fork perché almeno un processo deve essere creato senza la fork per poter inizializzare il resto. Il processo figlio è quasi identico in quanto lo spazio di indirizzamento è identico, **cambiano il PID e il PPID**, quest'ultimo sarà il PID di chi ha fatto la fork. I file descriptor vengono duplicati (system call dup) e l'intero spazio di indirizzamento viene copiato.
Copiare davvero l'intero spazio di indirizzamento sarebbe un'operazione molto costosa, quello che succede nei sistemi moderni è che viene effettuata un'ottimizzazione chiamata copy-on-write, ovvero viene creata una tabella delle pagine per il processo figlio che punta alle stesse pagine fisiche del processo padre. I due processi condividono le pagine, marcate in read-only e se un processo prova a scrivere viene attivata una trap che rende write-only **solo la pagina in cui il processo vuole scrivere**.
La particolarità della system call fork è che è una funzione che è chiamata una volta ma ritorna due volte perché, quando la system call ha successo, clona il processo; quindi, ottengo un processo che è fatto come il padre e in entrambi questi processi la fork ritorna. Quello che cambia tra i due processi è il valore di ritorno della fork: il processo figlio ritorna 0, mentre il processo padre ritorna il PID del processo figlio.

Quando viene creato un nuovo processo con fork, i debugger non capiscono che ci sono due processi, ma di solito vscode ti fa scegliere se vuoi seguire il padre o il figlio con **set follow-fork-mode[child|parent]**. Posso anche decidere di seguire entrambi con **set detach-on-fork off**.

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // per far stare nella slide, non controllo i valori di
        // ritorno; NON fatelo nel codice "vero"
        char msg[] = "pippo\n";
        fork();
        fork();
        write(STDOUT_FILENO, msg, sizeof msg);
}
```

Considerando il codice, quanti "pippo" va a scrivere?
P1 (processo 1) con la prima fork crea P2. Sia il padre che il figlio eseguono la fork, quindi P1 avrà un altro figlio P3, ma anche P2 avrà un figlio ossia P4.
Avremo quindi 4 system call write che andranno a scrivere "pippo".



Come vengono terminati i processi:
Lo standard C prevede che esista una funzione che si chiama exit che ha un unico parametro che è status. La system call è invece _exit e sono entrambi void.

void exit(int status) **void _exit(int status)**

Le differenze tra la prima e la seconda:

- La **funzione di libreria** prima di uscire davvero, ossia prima di chiamare _exit, chiama tutte le funzioni che sono state registrate tramite le funzioni **atexit** e **on_exit**, quando lo chiediamo noi tramite puntatori. Poi **svuota i buffer di input/output**, quindi prima di uscire dal programma se c'è ancora del contenuto nel buffer che non è ancora stato stampato viene stampato per non perderlo. Infine, va ad eliminare i file temporanei che sono stati creati con la funzione **tmpfile**.
- In **entrambi i casi**, vengono **chiuse** e **rilasciate** le **risorse del processo** (file descriptor, ecc.). L'**exit-status**, quello che viene passato alla funzione exit, rimane registrato, e dalla shell viene recuperato con $?. Eventuali figli, ora **orfani**, vengono **adottati da init**. Per convenzione, lo standard C definisce le costanti EXIT_SUCCESS (=0) ed EXIT_FAILURE (=1).

La system call wait:
la system call wait serve ad **attendere un cambio di stato di un figlio**, ossia una terminazione, stop o ripartenza tramite segnali. Wait aspetta un figlio qualsiasi: il primo figlio che termina sarà quello che verrà restitutito da wait. Se si vuole aspettare un figlio in particolare, esiste anche la system call **waitpid** che permette di specificare quale figlio si vuole aspettare. Se la wait va a buon fine, **restituisce il PID** del figlio che è terminato e se **wstatus** è diverso da 0 va a scriverci cosa è successo, quindi se il processo è terminato tramite exit o tramite segnale, e anche qual è stato il numero del segnale che ha ucciso il figlio. Se il puntatore di wstatus è WIFEXITED, allora si può recuperare l'exit-status con WEXITSTATUS; se il puntatore di wstatus è WIFSIGNALED, allora si può recuperare il segnale con WTERMSIG.

28 ottobre 2024

Cosa succede se un processo termina ma il padre non lo aspetta?
Questo processo diventa uno **zombie**. Per essere uno zombie un processo deve essere **terminato**, il sistema non può buttare via quel processo, non può dimenticarlo perché il **padre è ancora vivo** e potrebbe in futuro fare la wait e sapere cos'è successo al processo figlio. **Un processo zombie è quindi un processo di cui il sistema non può liberarsi perché esiste ancora il padre.** Tutti i processi che sono terminati di cui il padre non è terminato e non li ha aspettati sono dei processi zombie, che **consumano** un po' di risorse del sistema (ma non la cpu).
Quando un processo termina, al padre viene inviato un segnale **SIGCHLD**, che di solito è ignorato. Quindi se un processo genera dei processi figli e vuole gestirli, o li aspetta ciclicamente o non ignora il segnale SIGCHLD e fare delle wait per i figli persi.

Cosa vuol dire eseguire un programma?
Eseguire un programma vuol dire utilizzare la system call **execve**. Ci sono poi tutta una serie di funzioni di libreria (execl, execlp, execle, execv, execvp, execvpe) che offrono semplicemente un'interfaccia diversa alle funzionalità delle execve.

int execve (const char *pathname, char *const argv[ ], char *const envp [ ]);

Execve prende tre parametri: il primo parametro, pathname, è il percorso del programma che si vuole eseguire; il secondo argomento (argv[ ]) è l'array dei parametri da riga di comando; l'ultimo parametro (envp[ ]) sono le variabili d'ambiente.

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
// libc:
int execl(const char *pathname, const char *arg, ...  /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...  /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...  /*, (char *) NULL,
                                        char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Le varianti hanno v, l, p ed e. Dove c'è scritto **l**, gli argomenti al posto di essere specificati da un vettore sono messi come lista di argomenti all'execl. Dove c'è una **p**, non prende pathname come primo argomento ma il nome di un file. Quelli che hanno la **e**, sono quelli che ti permettono di specificare l'ambiente.

Il programma che viene mandato in esecuzione viene **avviato all'interno del processo chiamante**; quindi, non viene avviato un nuovo processo.
Eseguire un programma vuol dire che viene creato un nuovo **spazio di indirizzamento** completamente vuoto e viene inizializzato a partire dal programma che abbiamo specificato. Il sistema va innanzitutto a controllare se il programma specificato esiste, se non esiste lo spazio di indirizzamento viene eliminato e si ritorna -1; se il file esiste, si controlla se è **eseguibile**, ossia se l'utente può eseguire. Se si può eseguire, se si tratta di un file ELF, ci saranno i vari metadati, ecc. e queste **porzioni di file** vengono **mappate** nel nuovo spazio di indirizzamento, dove viene allocata una nuova tabella delle pagine, e vengono poi vengono creati la regione heap e la regione stack, copia le variabili d'ambiente, ecc. Se tutto va a buon fine, il vecchio spazio di indirizzamento viene eliminato e lo spazio di indirizzamento del processo diventa quello nuovo. Se fallisce, si rimane nel vecchio spazio di indirizzamento e l'exec ritornerà -1.
Se il file che andiamo ad eseguire ha il bit **set-user-ID/set-group-ID** abilitato, succedono cose che approfondiremo più avanti. Exec non è come fork, quindi **non crea un processo**, lo esegue solo.

Come funziona la redirezione dell'input/output:
Quando apriamo un file, il sistema operativo, il kernel, restituisce un **file descriptor**, che non è altro che un numero intero non negativo che corrisponde al file aperto. Dietro le quinte, questo numero viene usato dalla system call come **indice all'interno della tabella del file descriptor di quel processo.**

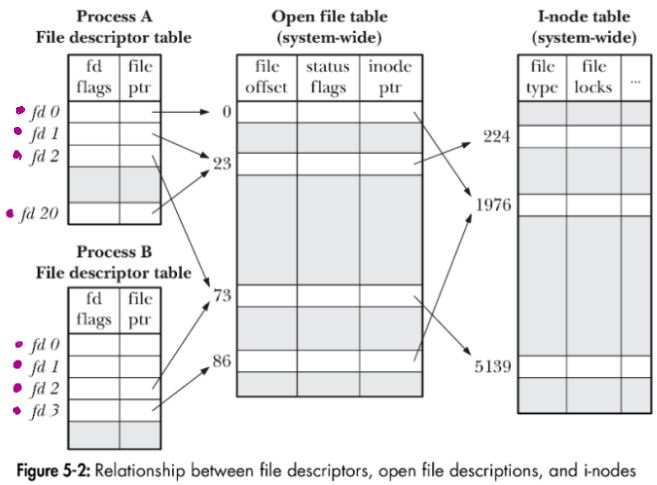


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Per ogni entry in questa tabella, ci sono due informazioni:

- Una colonna che rappresenta i **flag** di quel file descriptor;
- Un **puntatore ad un'altra struttura dati** che corrisponde ai file aperti nel sistema; questa struttura dati ha a sua volta un offset, altri flag e un puntatore ad un **i-node**, ossia una struttura dati che corrisponde ad un **file all'interno del file system**.

Per ogni file che si ha su un file system linux ci sarà un i-node corrispondente, che è una struttura dati che **contiene vari metadati riguardo al file** (dimensione, permessi, ecc.).
Quando si va ad aprire un file ad esempio con una open, alla open si passa la stringa che è il percorso per raggiungere il file che si vuole aprire. Se il file esiste, a quel file corrisponderà un i-node.

Per esempio: supponiamo che il file "quiquoqua" sia l'i-node 224.

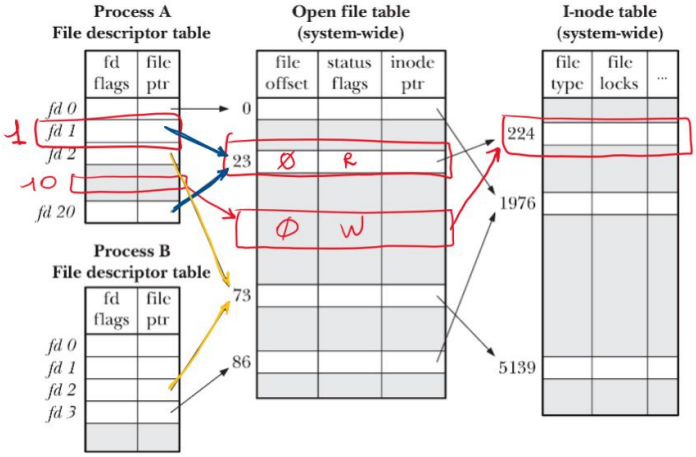


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Se un processo apre il file che corrisponde all'i-node 224, **l'i-node sarà portato in memoria** e verrà creata una **entry** nella tabella **open file table**. **L'offset** all'inizio sarà **0** e i **flag** spiegano **il modo in cui è aperto il file** (nell'esempio solo lettura, read). Dopodiché, viene creata una **entry** nella **tabella del file descriptor** e verrà restituito quel file descriptor al processo. Quindi se tutto va a buon fine, in questo caso la open restituisce 1. Se un altro processo o anche lo stesso processo fa una open dello stesso file, allora l'i-node sarà uguale, ma non è detto che l'offset e i flag siano uguali (ad esempio se lo apro in scrittura, write); quindi, avremo **altre entry in open file table**, e corrisponderanno a **file descriptor diversi**. Se io cerco di leggere con il file descriptor 1 ci riesco, se provo con il file descriptor 10 non ci riesco: il file descriptor è valido, ma è stato aperto in solo scrittura, quindi non posso leggerci. Se **chiudo** il file descriptor 1, perdo l'entry nella file descriptor table, la entry nella open file table, ma non perdo 224 in quanto ancora puntato da 10. Se chiudo anche 10, allora verrà perso anche il file.

Un'altra possibilità è avere dei **file descriptor diversi che fanno riferimento alla stessa entry** nella tabella open file table (nell'esempio fd 1 e fd 20). Per quel processo, utilizzare il file descriptor 1 e il file descriptor 20, non avrà nessuna differenza. Questo succede con la system call **dup** crea un altro file descriptor per un file che abbiamo già aperto.

Un secondo caso è quello dell'esempio delle foto con fd 2 e fd 2 di entrambi i file descriptor, che sono **entrambi puntati alla stessa entry**. Probabilmente il **secondo processo** è stato **creato tramite fork** del primo processo, quindi eseguono le stesse cose, ma dopo la fork ognuno ha continuato diversamente in maniera dipendente.

Un file descriptor può essere **duplicato**. **Dup** restituisce un altro file descriptor che **però punta allo stesso file** del file descriptor duplicato. Dup promette di dare il file descriptor **più piccolo disponibile**. Esiste anche **dup2**, che funziona più o meno allo stesso modo, ma al posto di dare il più piccolo si può chiedere un file descriptor in **particolare**.

Sappiamo che, quando usiamo la **bash**, **l'output** di una funzione chiamata **finisce sul terminale**. Questo succede perché l'output della funzione finisce sul file descriptor **1**, ossia **standard output**, che normalmente è collegato al **terminale**. Se però dopo la funzione faccio **>*nomefile***, l'output della funzione finirà nel file.

{ Ad esempio, quando sulla bash scrivo ls, la bash farà una fork, un exec e poi farà una wait. Se va a buon fine, sparisce ed exec diventa ls. Quando ls termina, wait termina a sua volta.
Se vogliamo mandare output su pippo, dobbiamo fare in modo che nel processo figlio il file descriptor 1 non sia più collegato al terminale ma collegato al file pippo. Come si fa? Intanto si apre pippo, con il file descriptor 7, chiudo 1 e apro pippo. Ma se fallisce mi sono giocato 1. Quindi facciamo prima la open, se va a buon fine posso chiudere 1, posso duplicare 7 e la dup di 7 mi darà 1. Quindi sia fd1 che fd7, corrisponderanno a pippo. Quindi, chiudo 7. Queste operazioni le faccio tutte dopo la fork, che farà una copia della bash che eseguirà queste operazioni.
Il fatto che la execve non tocchi i file descriptor è fondamentale; perché, se la execve resettasse i file descriptor al valore di default, la redirezione dell'input/output non la faremo.
Questo ci permette di isolare la redirezione del input/output sulla bash, che la esegue prima di lanciare qualsiasi comando. L'alternativa sarebbe che tutti i comandi dovrebbero gestirsi la possibilità di fare input/output rediretto ad un file, ma non avrebbe senso.
La redirezione dell'output è uguale ma al posto di fd1 uso fd2.
Con la pipe invece è diverso: ho due processi che girano in parallelo e l'output del processo di sinistra è l'input del processo di destra (?). In questo caso esiste una system call che si chiama pipe che crea un tubo dove avete un file descriptor per scrivere e un file descriptor per leggere da questo tubo. Un tubo è un buffer all'interno del kernel. Alla pipe passate un array di due file descriptor e poi fd1 scrive nella pipe e fd0 legge nella pipe. L'output del comando di sinistra dovrà essere rediretto all'interno della pipe, e l'input del comando di destra dovrà leggere dalla pipe. La bash crea quindi prima la pipe, poi fa due fork perché ci sono due processi figli; in uno legherà lo standard output all'input della pipe, nell'altro legherà lo standard input all'output della pipe. Poi dovrà chiudere tutti i file descriptor che non servono. Quando un processo cerca di leggere da una pipe vuota, se ci sono ancora dei file descriptor aperti che possono scrivere nella pipe, il processo che cerca di leggere viene sospeso, in attesa che qualcuno scriva nella pipe. Se invece il buffer si riempie perché nessuno lo legge, rimarrà pieno e il processo rimarrà sospeso fino a quando un file descriptor non leggerà e quindi il processo ripartirà. }

Scheduling

4 novembre 2024

Lo **scheduling**, nel contesto dei sistemi operativi, riguarda il **come la cpu / le cpu vengano allocate ai vari processi**. Ad esempio: abbiamo una sistema con una sola cpu e un solo core, e abbiamo un'unità di esecuzione e 10 processi che vogliono andare in esecuzione; lo scheduling fa in modo che il nucleo decida quali processi mandare in esecuzione e per quanto.

Lo scheduling in generale riguarda il **come allocare dei lavori su delle unità di esecuzione**. Per quanto riguarda il sistema operativo, ci sono due parti che riguardano la **schedulazione dei processi**:

- Il **meccanismo** che permette al kernel di cambiare processo, chiamato **cambio di contesto** (context switch), ossia salvo i registri che sta utilizzando un processo e vado a caricare quelli dell'altro processo, scritto in **linguaggio macchina**;
- Lo **scheduler** è la parte del kernel che decide il prossimo processo da mandare in esecuzione, seguendo una certa **politica** (policy), scritto in **C**. Si tratta di valutare, fare delle considerazioni sui processi e decidere il prossimo da eseguire.

I sistemi ci danno l'illusione di **far andare in parallelo i programmi** perché fanno eseguire un processo per una fettina di tempo, poi un altro, poi un altro ecc. Se questa fettina di tempo è molto piccola a noi dà l'illusione che tutti progrediscano contemporaneamente.
Il context switch ha un costo, perché tutti i registri di una cpu vanno salvati in un'**area all'interno del kernel**, e per ogni processo nel sistema, il kernel ha una **struttura dati** (**PCB**, process control block), dove vengono salvate tutte le informazioni di un processo. Siccome il **context switch** è un'operazione che viene fatta migliaia di volte al secondo, dev'essere **molto efficiente**.
Nell'esempio con un solo processore da singolo core, se sta girando il processo non sta girando il sistema operativo.

<u>Come fa il sistema operativo a riprendere il controllo ed eventualmente cacciare il processo che si stava eseguendo e farne partire un altro?</u>
Ci sono diversi **approcci**:

- **Cooperativo**: il sistema operativo si "fida" dei processi
    - i processi faranno dei system call, facendo delle system call chiamano il kernel e a quel punto il kernel può decidere cosa fare con il processo
    - se un processo non fa system call il computer rimane per sempre nel processo (quindi questo approccio ha un enorme limite)
- **non cooperativo**: il sistema operativo riprende il controllo a forza, tramite un timer interrupt

Nel **context-switch** si salvano i registri del processo che si stava eseguendo e si vanno a caricare i registri del processo che vogliamo mandare in esecuzione: tutti i registri, in particolare anche il registro che contiene il **puntatore alla tabella delle pagine**, e questo può avere un impatto per esempio sulla **TLB**, la **translation lookaside buffer**, quella **cache** che memorizza le **traduzioni tra indirizzi logici e indirizzi fisici**. Quando cambia la tabella delle pagine, questa cache va invalidata perché altrimenti nella cache si troverebbero delle associazioni che riguardavano il processo di prima e non quello corrente.

Ogni processo ha uno stack utente e uno stack kernel.
Lo **stack pointer** è un registro che punta alla cima dello stack, però un processo può usarlo come vuole, nulla vieta ad esempio che in un certo momento lo stack pointer può avere un valore sbagliato, oppure il processo potrebbe decidere di utilizzare lo stack pointer come registro e usarlo per conti, ecc.
Quando arriva un **interrupt** automaticamente la cpu passa allo **stack kernel**, che è un valore che ha settato il kernel nell'inizializzazione del sistema, in questo modo sappiamo quindi che l'interrupt handler (il pezzo di codice che gestisce l'interrupt) avrà uno stack valido.

Quindi se vogliamo passare da un processo A ad un processo B:

- si stava eseguendo il processo a;
- arriva l'interrupt;
- verranno salvati dei registri sul kernel stack del processo a;
- si va ad eseguire l'interrupt handler;
- il kernel decide di passare al processo b;
- si salverà nel pcd (struttura che contiene le informazioni dei processi, in questo momento quelli del processo a) i valori attuali del registro della cpu e andrà a caricare quelli del processo b;
- ritornerà dall'interrupt.

Questa cosa avviene migliaia e milioni di volte al secondo, per questo si ha bisogno che sia molto efficiente.

Un processo nella sua vita può essere in tanti stati diversi:

- lo stato ready: il processo è pronto ad essere eseguito ma non si sta eseguendo;
- lo stato running: il processo si sta eseguendo;
- lo stato blocked/waiting: il processo sta aspettando qualcosa, il processo non ha ricevuto un input e sta aspettando (quando ricevuto entrerà in ready).

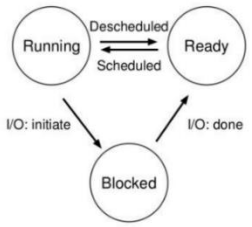


Figure 4.2: **Process: State Transitions**

Se abbiamo un'unica cpu con un solo core, possiamo avere un **unico processo in running**, e possiamo avere un **numero arbitrario di processi ready e processi blocked**.

Le politiche di scheduling sono quegli algoritmi che decidono quale processo mandare in esecuzione. Ci sono varie politiche diverse, e, a seconda del contesto, una potrebbe funzionare meglio dell'altra.
Come facciamo a capire quale dobbiamo usare? Ci sono delle metriche.

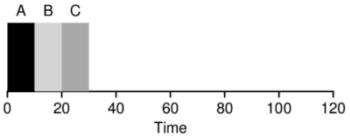
Partiamo con assunzioni irrealistiche: ciascun job duri lo stesso tempo; tutti i job arrivino allo stesso momento; una volta iniziato un job si esegue fino in fondo, senza interruzioni; non c'è input/output; il tempo di ciascun job è noto a priori.

Tempo di turn-around: il tempo di turn-around è definito come

tempo di completamento – tempo di arrivo

e se assumiamo che tutti arrivino allo stesso momento (con le assunzioni irrealistiche), il tempo di arrivo è uguale a 0, e quindi il tempo di turn-around è semplicemente il tempo di completamento.
Questo ci può servire per misurare la **bontà di un algoritmo di scheduling** rispetto ad un altro, però non ci interessa solo la **performance**, ma anche **equità**; in particolare, quando si hanno dei processi, si deve evitare una situazione chiamata **starvation**, in cui un processo non riesce mai ad avere accesso alla cpu.

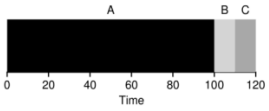
Algoritmo FIFO (first in first out):
il primo job che arriva è il primo che viene eseguito. Se tutti i job arrivano allo stesso momento, l'algoritmo sarà obbligato a trovare un ordine.



Come faccio a decidere un ordine?

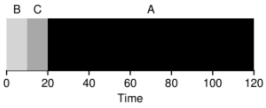
$$T_{\text{turnaround}} = \frac{10 + 20 + 30}{3} = 20$$

- **Effetto convoglio:** arrivano A, B e C allo stesso momento. A dura 100 unità di tempo, mentre B e C durano 10 unità di tempo.



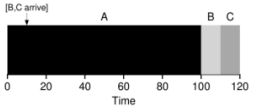
$$T_{\text{turnaround}} = \frac{100 + 110 + 120}{3} = 110$$

- **Shortest-job first:** se andiamo ad eseguire prima i job corti, il tempo totale è lo stesso, però il tempo medio di turn-around è più basso.



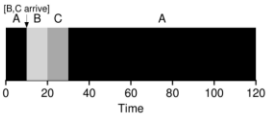
$$T_{\text{turnaround}} = \frac{10 + 20 + 120}{3} = 50$$

- **Shortest Time-To-Completion First:** Se non arrivano tutti allo stesso momento, ad esempio se arriva prima A lungo, e poco dopo arriva B più corto, ormai si sta eseguendo A quindi non possiamo mandare in esecuzione gli altri.



$$T_{\text{turnaround}} = \frac{100 + (110 - 10) + (120 - 10)}{3} \approx 103$$

Quindi cosa si potrebbe fare? Quando arrivano B e C metto in pausa l'esecuzione di A, completo B e C, e poi riprendo A.



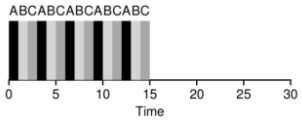
$$T_{\text{turnaround}} = \frac{120 + (20 - 10) + (30 - 10)}{3} = 50$$

Può però soffrire di starvation perché, se continuano ad arrivare job corti, il job a lungo non finirà mai; inoltre, non può essere davvero usato, in quanto in un sistema operativo non sappiamo quanto sarà davvero lungo un processo.

Inoltre, mentre nei **sistemi non interattivi**, detti anche sistemi batch, il tempo di turnaround medio può essere una metrica sensata, nei sistemi interattivi c'è un'altra metrica molto importante, ossia il **tempo di risposta**, che deve essere molto basso. È definito come

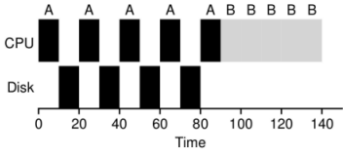
tempo di inizio esecuzione – tempo di arrivo

Un modo per avere un tempo di risposta buono è il **Round-Robin** (girotondo), e l'idea è che mandiamo in esecuzione tutti i job che sono arrivati per una piccola fettina di tempo per alternarli.

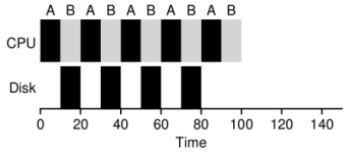


Dal punto di vista del tempo di risposta sono ottimi, ma dal punto di vista del turn-around è pessimo, perché andiamo a diluire l'esecuzione di tutti e quindi terminano tutti "alla fine". Il response-time con il round-robin si può ottimizzare restringendo la fettina di tempo, ma non troppo piccola perché se no la cpu fa solo context-switch e non fa altro di utile (overhead). Evita la starvation ma penalizza i job corti.

Nei sistemi reali un processo fa input/output, pertanto sarebbe stupido lasciare la cpu ad un programma in wait; quindi, lo buttiamo fuori (nella memoria disco) e possiamo usare la cpu per altro.



diventa

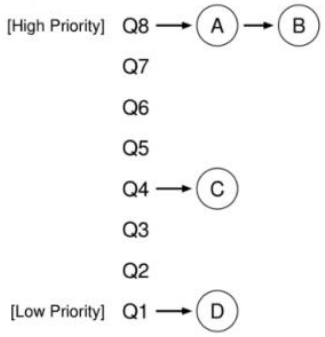


Esempi di algoritmi realistici sul come possiamo schedulare i processi:

- **Multi-level Feedback Queue** (algoritmo che utilizza windows)
- **Propotional share e Linux CFS** (Completely Fair Scheduler)

Il **MLFQ** (Multi-level Feedback Queue) tenta di **ottimizzare** sia il turnaround time che il response time; cerca di eseguire prima i job corti. L'idea è quindi che i processi interattivi, quelli che fanno molto input/output, avranno una priorità tendenzialmente più alta, mentre invece i processi "CPU Bound" avranno invece una priorità più bassa. Però chiaramente a priori non sappiamo se il programma che facciamo partire è **i/o bound** o **CPU bound**, e un programma in momementi diversi può essere i/o bound o CPU bound.
l'idea del MDFQ è di avere più **code** a priorità diversa, e per tutti i processi che stanno sulla stessa coda, vanno in round-robin. Dopodiché, la proprietà di un processo verrà variata in base a come si comporta il processo; quello che fa l'algoritmo è cercare di **prevedere come si comporterà un processo basandosi su quello che ha fatto fin'ora.**
Descriviamo l'algoritmo con delle regole:

1) Se p(A) > p(B), gira A (e non gira B)
2) Se p(A) = p(B), A e B vanno in Round-Robin
3) Un nuovo job entra con la priorità massima
4) Se un job usa un tempo fissato t a una certa priorità x (considerando la somma dei tempi usati nella coda x), allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità)
5) Ogni s secondi, spostiamo tutti i job alla priorità più alta (se no starvation)



Il problema più grosso dell'algoritmo è definire i parametri (t, x, s, ecc.) quindi si fa scegliere principalmente dall'amministratore di sistema.

Linux usa un approccio completamente diverso, il **CFS** (Completely Fair Scheduler), uno scheduler completamente equo che non faccia preferenze (anche se non è sempre vero).
Siccome vuole essere equo, se ho tre processi che vogliono utilizzare la CPU, do 1/3 di CPU a testa. In generale manderò in esecuzione il processo che ha utilizzato la CPU per meno tempo (perché se ne ha usata meno "se ne merita ancora" per essere pari agli altri).
Il cfs misura il tempo utilizzato sulla cpu tramite **virtual runtime**, **vruntime**. Se tutti i processi avessero la stessa identica priorità, il virtual runtime sarebbe semplicemente il runtime (quanto tempo un processo ha speso usando la cpu); il virtual run time ci permette di avere processi con più priorità, in cui facciamo degli sconti, in cui lasciamo usare più cpu perché ha più priorità, gli viene conteggiato meno il tempo in cui sta sulla cpu. In generale, quando c'è da **scegliere quale processo mandare in esecuzione**, il CFS sceglie quello che ha il **virtual runtime più piccolo**.
CFS usa vari parametri, tarati dall'amministratore, uno è **sched_latency**, ossia il **tempo assegnato ad ogni processo**. Se ci sono n processi che sono pronti ad essere eseguiti, quindi nello stato ready, la fettina di tempo che viene data a ogni processo è **sched_latency / n**. Se però è troppo piccolo, vado a prendere il valore di un altro parametro, **min_granularity**, che dice qual è la fettina più piccola che vado ad assegnare (non posso scendere sotto min_granularity).
Vengono mandati in esecuzione questi processi, e poi il vruntime di ogni processo verrà **aggiornato** in base a quanto tempo sono stati effettivamente in esecuzione.

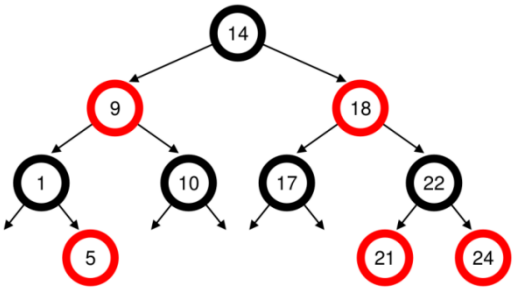
Nei sistemi reali, alcuni processi sono più importanti di altri, quindi c'è un valore **nice** che indica la priorità di un processo: più un processo è **nice**, più **cede la cpu ad altri**.

$$\texttt{slice}_k = \max(\texttt{sched\_latency} \cdot \frac{w_k}{\sum_{i=0}^{n-1} w_i}, \texttt{min\_granularity})$$

$$\texttt{vruntime}_k \mathrel{+}= \frac{\texttt{runtime}_k}{w_k}$$

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Per trovare efficientemente il minimo/aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero:



con costo log(n) per trovarlo

Che vruntime devo dare ad un processo nuovo? Il vruntime minimo che c'è in quel momento nel sistema, quindi andrà subito in esecuzione senza pero utilizzarsi la CPU per tutto il tempo. Quindi non è "completely fair" con i processi che fanno frequentemente i/o, perché un processo che diventa ready si prende un vruntime uguale al minimo degli altri.

Threading

5 novembre 2024

Un **thread** è un **flusso di esecuzione all'interno di un processo**. Creare un thread sostanzialmente vuol dire creare una **CPU virtuale** e allocare uno stack per il nuovo flusso di esecuzione.
Un processo è un programma in esecuzione, e dentro ad un processo abbiamo una virtualizzazione della memoria, che è lo spazio di indirizzamento e pensa di avere la CPU tutta per sé, non sa che viene sospesa l'esecuzione, viene fatto il context-switch, ecc.
Creare un thread vuol dire che aggiungo una virtualizzazione della CPU all'interno di un processo. Da un certo punto di vista c'è una somiglianza tra creare un thread e creare un processo; la grossa **differenza** tra un thread e un processo è il fatto che **più thread vedono lo stesso spazio di indirizzamento**, quindi la stessa virtualizzazione della memoria, mentre un processo è completamente isolato dagli altri processi.
Quando uno scheduler passa da un thread di un processo ad un altro thread dello stesso processo, **non deve cambiare lo spazio di indirizzamento.**
A livello di kernel, un sistema operativo ha una struttura dati per ogni processo, ossia la PCB (Process Control Block), e analogamente ogni thread ha una struttura dati che rappresenta i suoi dati, chiamata **Thread Control Block**.
Dal punto di vista logico, ogni thread ha i propri:

- **Stack**, regione di memoria dove vengono allocate le variabili locali, salvati gli indirizzi di ritorno di una funzione, metadati, ecc. Però tutti questi stack vivono nello stesso spazio di indirizzamento,

quindi, se un puntatore va dove non deve, un thread può leggere o scrivere all'interno dello stack di un altro thread, cosa impossibile tra processi diversi.

- **Variabile errno** (che sta nel proprio TLS, Thread Local Storage, memoria privata che ha ogni thread), importante che ogni thread ne abbia una perché ogni thread può fare system call, e se la system call fallisce, la ragione per cui fallisce viene salvata dentro errno. Sono quindi uniche per ogni thread perché, se fosse in comune, verrebbe sovrascritta.

Il **Thread Local Storage** è uno spazio locale a ogni thread, accessibile tramite interfaccia di tipo **dizionario chiave/valore**.

Perché dovremmo voler usare i thread? Potremmo voler usare i thread per sfruttare il **parallelismo**, suddividere il carico di lavoro fra più core per far andare un programma più veloce; un altro motivo è facilitare l'input/output.

Come si crea un thread? Per creare un processo usiamo fork, per creare un thread c'è una **libreria Posix** chiamata **pthread**, e si usa **pthread_create**.

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);

Il primo argomento è un puntatore a pthread_t, che è sostanzialmente l'equivalente del PID, quindi identifica il thread. Poi il secondo parametro permette di specificare attributi opzionali, molto spesso non usato. Il terzo parametro *start_routine è un puntatore, che punta a funzione che prende un void * e restituisce un void *, e questo sarà il codice eseguito sul thread; il paramentro arg di tipo void * è l'argomento che è passato a start_routine.
In caso di successo la funzione restituisce **0**, altrimenti restituiscono il codice di errore ma non viene scritto su errno.
Per uscire, per restituire un valore dal thread, si può fare return dalla funzione che è stata fatta partire con **pthread_create**, oppure si può chiamare da qualsiasi funzione void **pthread_exit(void *retval);**
Quando si fa exit terminano tutti i thread di quel processo, perché si sta uscendo dal processo, mentre con pthread_exit si esce solo da quel thread.
Si può attendere la terminazione di un thread con **int pthread_join(pthread_t thread, void **retval)**, si deve specificare quale thread aspettare, e in retval è scritto il valore di uscita. Se non si vuole attendere la terminazione si usa **pthread_detach**, altrimenti rimane un thread zombie.

Una **sezione critica** è un pezzo di codice che va ad accedere ad una risorsa condivisa. Si parla di **race condition** quando il risultato finale dipende da come sono schedulati i thread. Per evitare questi problemi serve **sincronizzare i thread** (eventualmente anche di processi diversi).
La **mutua esclusione** è: se un thread sta accedendo alla risorsa, nessun altro thread in quel momento può accedere alla risorsa, ci deve essere un'alternanza.
Possiamo rendere la sezione critica atomica utilizzando i **lock**, o **mutex** (da mutua esclusione). Prima di iniziare la sezione critica, un thread deve accedere al lock, accedere al mutex, quindi prendersi il lock sul mutex. Chi vuole lavorare su una risorsa condivisa deve accedere per poterlo fare.
Si dichiara con **pthread_mutex_t**, può essere acquisito (preso/tenuto), da un thread alla volta, tramite **pthread_mutex_lock**, e va rilasciato, il prima possibile, tramite **pthread_mutex_unlock**.

Quanti lock utilizzare all'interno di un programma? Non c'è una regola fissa, ma chiaramente se uso un unico lock in tutto il programma, vuol dire che, quando dei thread devono sincronizzarsi su qualcosa, utilizzano quell'unico lock. Sarebbe quindi più intelligente utilizzare un lock per ogni struttura dati all'interno di un programma.

Come possiamo andare ad implementare i lock? Quando andiamo ad implementare i lock ci andiamo a preoccupare di:

- Se la nostra implementazione offre mutua esclusione;
- Fairness e starvation (un thread in attesa ha qualche garanzia di ottenere, prima o poi, il lock?);
- Performance.

Senza un supporto hardware non si può implementare. Diversi processori forniscono delle primitive atomiche quali:

- Test-and-set: singola operazione che testa il valore di un'allocazione di memoria e se trova 0 lo mette a 1;
- Scambi atomici: inverte il valore di 2 allocazioni di memoria.

Spin-lock:

```
typedef struct __lock_t {
        int is_locked;
} lock_t;
void init(lock_t *lock) {
        lock->is_locked = 0;
}
void lock(lock_t *lock) {
        while (AtomicExchange(&lock->is_locked, 1) == 1)
; // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
        lock->is_locked = 0;
}
```

- garantisce mutua-esclusione;
- non è fair ed è possibile starvation;
- cosa succede se un thread che ha già acquisito il lock cerca di farne nuovamente il lock? lock non ricorsivo (ne esistono varianti ricorsive);
- performance: dobbiamo distinguere rispetto al numero di core:
    - singolo core: se un thread che ha il lock viene deschedulato, gli altri sprecano tempo e CPU;
    - core multipli: funziona discretamente bene.
- Ha senso aspettare in un loop, consumando CPU? se si aspetta poco sì: i context-switch costano negli anni, varie proposte di approcci ibridi, ovvero "lock in due fasi" (un po' di spin, poi eventualmente sleep).

Scheduling e (spin-)lock possono interagire in modi inaspettati

Inversione delle priorità: Supponiamo di avere uno scheduler a priorità e due thread:

- Con T1 bassa priorità e T2 alta priorità
- Assumiamo che T2 sia in attesa di qualcosa, quindi va in esecuzione T1
- T1 acquisisce un certo lock che chiamiamo l
- T2 torna ready
- Lo scheduler deschedula T1 e manda in esecuzione T2
- T2 va in spin-wait per il lock l
- Game over: T1, che è l'unico che può rilasciare il lock l, non viene messo in esecuzione, perché T2 ha priorità

Funzione rientrante (definizione nata in un'epoca single-threaded): funzione che si comporta correttamente anche se interrotta a metà di un'esecuzione per essere nuovamente chiamata (esempi: interrupt-hander o ricorsione).
Funzione thread-safe: funzione che si comporta correttamente anche se eseguita da più thread contemporaneamente.

Sono due definizioni separate e una non implica l'altra.
Ma in posix useremo la definizione di "rientrante" quando parliamo di funzioni thread-safe. Non tutte le funzioni posix sono rientranti; quelle che non lo sono, hanno tipicamente la variante _r.

18 novembre 2024

Bug tipici dovuti alla concorrenza, ossia codici che in una situazione single-threaded non danno problemi ma che in una situazione multi-threaded potrebbero dare problemi.

Esempio 1:

```
struct foo *p;

void f() {
        ...
        if (p!=NULL) {
                p->bar = 3;
        }
}
```

Se ci sono altri thread all'interno del processo, un altro thread potrebbe andare a mettere a null quel puntatore; p potrebbe essere == 0.
Questo bug si chiama **violazione dell'atomicità.**

Esempio 2:

```
/* some pointer type */ glob_thread = NULL;

void init()
{
        ...
        glob_thread = create_thread(thread_func, ...);
        ...
}

void thread_func(...)
{
        ...
        foo = glob_thread->bar;
        ...
}
```

Ci sono due thread, uno che ha chiamato la funzione create_thread, e l'altro creato. Quando arrivo alla linea di codice di thread_func, il thread chiamato potrebbe non essere ancora stato inizializzato.
Questo bug si chiama **violazione dell'ordine.**

C'è un'altra categoria di bug chiamata **deadlock.** Supponiamo che ci siano due mutex nel programma (M1 e M2); un thread acquisisce il mutex M1, poi acquisisce il mutex M2 e poi fa qualcosa ecc. il secondo thread acquisisce il mutex M2, poi acquisisce il mutex M1 e poi fa qualcosa ecc.

```
Thread-A lock(m1); lock(m2);
Thread-B lock(m2); lock(m1);
```

Cosa può succedere? Si possono incrociare i flussi. Il thread-A non può proseguire perché sta aspettando che qualcuno rilasci M2, e thread-B non può proseguire (e rilasciare M2) perché aspetta M1.

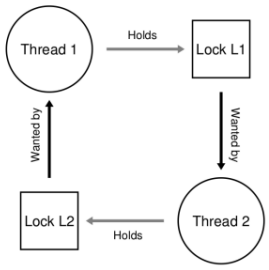


Figure 32.2: **The Deadlock Dependency Graph**

Le **condizioni per arrivare ad un deadlock** sono:

- La mutua esclusione: se non abbiamo la mutua esclusione non abbiamo il deadlock, ma è difficile ignorare e non usare la mutua esclusione;
- Hold-and-wait: i thread mantengono le risorse acquisite mentre ne richiedono/aspettano altre;
- No preemption: le risorse non possono essere tolte;
- Circular wait: ci deve essere un'attesa circolare.

Se rinunciamo ad una di queste preveniamo il deadlock.

I due modi per mantenere la correttezza del programma ed evitare il deadlock:

- Fare in modo che i lock vengano sempre acquisiti tutti insieme (se un thread ha bisogno di M1 e M2 allora acquisisce entrambi), quindi no hold-and-wait;
- Do un ordine ai lock e se li acquisisco sono obbligato ad acquisirli in ordine, quindi no circular wait.

Sicurezza

il sistema operativo è quello che **gestisce l'hardware** e **tiene isolati i processi** (ad esempio in un pc con più utenti i file di pippo sono visualizzabili solo da lui e da nessun altro utente ecc.).

Per sviluppare un sistema operativo sicuro, dobbiamo costruirlo su una base sicura: se il processore non è sicuro, ha delle falle, sicuramente anche il sistema operativo non potrà essere sicuro

Che cosa vuol dire sicuro? In generale ci sono tre grossi obbiettivi che vanno preservati per ottenere un sistema operativo sicuro:

- La confidenzialità, la segretezza, ossia possono accedere ai dati solo i rispettivi proprietari;
- L'integrità, ossia un altro utente non può verificare il contenuto di un mio file, anche accidentalmente;
- La disponibilità, ossia che il sistema risponda: un attacco contro la disponibilità, Denial Of Service (DOS) o Distributed Denial Of Service (DDOS), fa sì che il sistema venga sovraccaricato da richieste fasulle tanto che non riesce a rispondere agli utenti legittimi.

Quando un processo fa una system call, il kernel deve valutare la richiesta che viene fatta:

- Controlla se è sensata, ossia se il numero passato corrisponde ad una system call e se i parametri sono validi;
- Rispetta la politica di sicurezza che dice chi può fare cosa:
    - L'entità che fa la richiesta, chiamata principal o subject;
    - Le richieste sono relative ad un oggetto (l'object);
    - Hanno una modalità di accesso (lettura, scrittura, ecc.);
    - Nel valutare se eseguire o meno una system call, si deve considerare anche il contesto (ad esempio linux, android, ecc.).

Ad esempio, in Android, **ogni app funziona come se fosse un utente unix diverso** (le applicazioni con lo stesso publisher possono runnare nello stesso "utente"). Un utente **non parla direttamente con il kernel**, ma interagisce con il sistema e lancia i processi, che fanno le system call; quindi, deve esserci un'associazione fra utenti e processi, e un modo per verificare l'identità degli utenti.
Nei sistemi di autenticazione ci sono le cosiddette **AAA**:

- Authentication: verifica dell'identità (esempio: utente e password);
- Authorization: decide se accettare o rifiutare le richieste dell'utente, in base alle politiche di sicurezza dell'applicazione;
- Accounting: fare il logging di quello che succede e gestire il consumo delle risorse.

Identificazione e autenticazione: Sono argomenti molto vasti.
Il caso più comune: gli utenti si identificano con uno username, e si autenticano con una password.
All'interno dei sistemi unix ci sono due **file** che **contengono per ogni utente le sue informazioni**; in particolare:

- **L'user id** (uid), che è un numero che identifica l'utente
- **La password**

I file sono **/etc/passwd** dove c'è tutto tranne la password (perché può essere letto da tutti), e **/etc/shadow** dove c'è la password.

UID zero corrisponde a root, che è amministratore del sistema; quindi, i processi che girano con UID0 sono **privilegiati**, possono fare praticamente tutto. Quindi, **UID ≠ 0** sono **non-privilegiati**.

Per **l'autorizzazione** ci sono due approcci:

- **Access Control List**:
  - Ogni oggetto contiene una lista di coppie soggetto/accesso (subject/access);
  - Versione ottimizzata Unix in 9 bit (con r, w, x per proprietario, gruppo e altri).
- **Capabilities** (che non c'entrano con le Linux capabilities):
  - Sono delle chiavi che racchiudono un oggetto in accesso e ci danno il loro privilegio (object/access).

Sono approcci diversi, in Linux e Windows si usa l'Access Control List.
Indipendentemente dall'uso di ACL o capabilities, chi decide chi può e chi non può accedere ad una risorsa?

- Il proprietario, si parla di **DAC**: Discretionary Access Control, controllo degli accessi discrezionale dove il proprietario del file decide chi può accedere ecc.;
- Controllo di accessi imposto da regole, **MAC**: Mandatory Access Control, tipicamente usato in ambito militare, non su sistemi normali.

Un **principio** importante è quello **del minimo privilegio**: l'idea è che in ogni momento, ciascuna identità (quindi ciascun programma, cliente) **dovrebbe avere i permessi minimi per eseguire i suoi compiti**. **Se un programma ha più permessi del necessario, vuol dire che un attaccante può fare più cose.**

In Unix, per aumentare i privilegi si usava storicamente **su(1)** (Super User), che chiede la password dell'amministratore di sistema. Oggi si usa **sudo(1)**, che usa la password dell'user (che sono nel gruppo degli user autorizzati a farlo).

Ogni processo ha associato tre user id (UID), che quando facciamo **login** coincidono:

- Real UID: proprietario del processo, chi può uccidere il processo (kill);
- Effective UID: utilizzato per verificare la politica di sicurezza, identità usata per determinare i permessi di accesso a risorse condivise;
- Saved UID.

Esiste una system call **set-userid** che permette a un processo di **modificare l'effective UID**, facendolo diventare o come il **real UID** o come il **saved UID**. Se il chiamante della funzione è privilegiato, può modificarli tutti e tre.
Come cambiano? Di solito, quando si utilizza la funzione **execve** per eseguire un programma, non cambia gli id del processo chiamante; se però **il file eseguibile ha un bit dal nome set-userid, allora l'effective uid e il saved uid diventano uguali a quelli del proprietario del file.**

Chroot è una system call che ci permette di modificare il significato di **/** quando vengono risolti i percorsi di file. Normalmente **/** fa riferimento alla **root** del file system; con chroot dico "per questo processo e per i suoi figli il file-system non inizia dalla root ma da una cartella che decido sia la nuova root". Limita i danni dei bug. Permette di creare le cosiddette **chroot jail** che permettono di limitare la visibilità del file-system.

File system

25 novembre 2024

Persistenza: Come è implementato il file system. Tipicamente utilizziamo la parola "dischi" per riferirci a dispositivi a blocchi di memoria secondaria, perché storicamente erano dischi (anche se ora sono chip). L'interfaccia con cui il sistema operativo parla è sempre quella di **dispositivi a blocchi**: quando interagisco (leggo/scrivo) con un blocco, non posso solo leggere o scrivere un singolo byte ma devo leggere o scrivere l'intero blocco. Tipicamente sono divisi in 512 byte l'uno (anche a 4k). Esempi di dispositivi a blocchi: dischi magnetici (hard disk, floppy disk), dischi ottici (DVD, bluray, ecc.), pennette USB, SSD, ecc.

Ogni indirizzo di memoria RAM contiene dei byte, mentre sui dischi la questione è diversa. Per indirizzare una memoria molto grossa servono molti bit; se raggruppo in blocchi i dati, bastano meno bit per indirizzare i vari blocchi. Leggere o scrivere un blocco è veloce, ma se voglio cercare un blocco (dove devo scrivere), devo aspettare la rotazione del disco: **ritardo rotazionale**.
La grossa differenza tra un disco rispetto alla RAM è la **persistenza**: quando stacco la corrente non perdo il contenuto del disco, mentre perdo quello della RAM. La memoria secondaria è quindi persistente, molto più economica ma molto più lenta.

Nei sistemi Unix c'è una cache, chiamata **buffer cache**, che **contiene i blocchi del disco che abbiamo letto o che vogliamo scrivere**. Quando leggiamo un settore dal disco, il sistema operativo deve chiedere di leggere il contenuto, poi mette il contenuto del settore in una cache, per fare in modo che, se un altro processo/lo stesso processo chiede di nuovo di leggere quella porzione di disco, se la porzione è già in RAM, non serve perdere tempo leggendo il disco me si legge direttamente dalla cache. Quando scriviamo un settore, il settore finisce nella buffer cache e **non** viene scritto immediatamente su disco, così se viene modificato viene modificato solo in RAM e finisce poi su disco. Problema: se saltasse la corrente e i dati non sono ancora stati salvati su disco, si perdono.

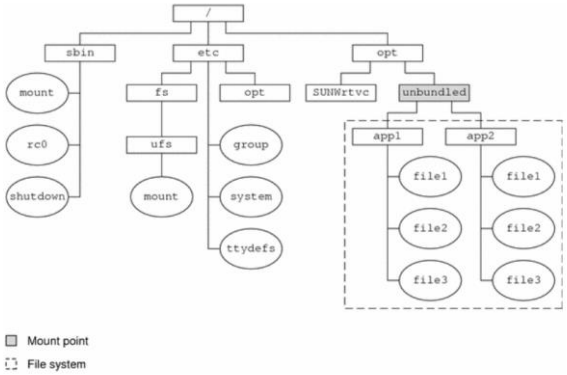
Un **volume** è una **sequenza di blocchi indirizzabili** che potrebbero anche non contigui o appartenenti a dischi diversi, ad esempio un insieme di più dischi economici.
Se vogliamo utilizzare un disco mettendoci ad esempio due sistemi operativi diversi, per farlo **partizioniamo** il disco, ossia creare dei dischi logici che sono una fettina del disco vero.
Per partizionare i pc, esistono due standard:

- MBR -> Master Boot Record, con 4 partizioni primarie + partizioni estese;
- GPT -> GIUD Partition Table, con 128 partizioni identificate da un UUID, Universal Unique Identifier).

Nei sistemi Unix-like esistono dei file speciali che fanno riferimento a dei dispositivi, che possono essere a caratteri (stampante, tastiera, ecc.) o a blocchi. Un file corrispondente ad un sistema a blocchi può fare riferimento all'intero disco oppure alle partizioni.
Questi file speciali che corrispondono a dispositivi possono essere creati con la system call **mknod** da root e sono identificati da due numeri:

- Il **major number** identifica il tipo di dispositivo (classe);
- Il **minor number** identifica uno dei vari dispositivi di quel tipo (istanze della classe).

Da utente, il file system viene visto come **un albero che ha una radice che è lo slash /, delle cartelle con all'interno dei file e altre cartelle ecc.** Su sistema Unix, quando si aggiunge un disco, questo deve essere **montato** su una cartella del file system corrente. Su windows viene invece creata una lettera di unità (esempio c quella principale, attacco una chiavetta che diventa d, ecc.).



Questi file che corrispondono ai dispositivi storicamente vengono inseriti sotto la cartella **/dev**; il problema è che, se io devo creare a priori i file che corrispondono ai dispositivi per tutti i dispositivi che supporta il kernel, mi ritroverò migliaia di file. Nei sistemi moderni **il file che corrisponde al dispositivo viene creato on demand quando viene creato il dispositivo e viene fatto sparire quando lo stacco**: il kernel solleva degli eventi quando i dispositivi vengono collegati e scollegati, e c'è un /dev che si occupa di andar a cercare il driver del dispositivo collegato, caricarli all'interno del kernel e verranno quindi creati i file speciali sotto /dev.

Il file system visto dall'utente è un'astrazione sopra l'utilizzo del disco totale. Di tutti i blocchi che stanno sul disco, una parte conterrà i dati veri e propri e una parte che verrà usata per i metadati. **Un file-system è una struttura dati che sta su un disco, quindi su un dispositivo a blocchi, e quando si formatta un volume si inizializza la struttura dati di quel volume**. Ci sono vari tipi di file system: FAT (File Allocation Table), FAT16, FAT32, 1216, XFAT, NTFS, EXT234, ecc.

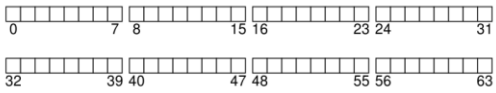
Ce ne sono alcuni più semplici, e hanno limitazioni diverse. A seconda del dispositivo che si usa per leggere il disco, potrebbe non essere supportato.

Implementazione di un file-system:
Un file system è una struttura dati, e quando si va a formattare un volume, si sta inizializzando una struttura dati. Noi consideriamo una versione semplificata, vsfs (Very Simple File-System). La struttura dati principale per il file system alla Unix è l'i-node, che contiene quasi tutti i metadati di un file e per ogni file su disco, c'è il corrispondente i-node.

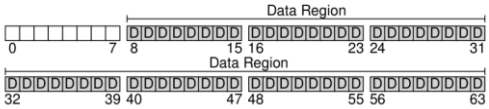
esempi:

Assumiamo di avere un volume di 64 blocchi da 4K.

Spesso i settori si raggruppano in blocchi logici, chiamati **cluster**.



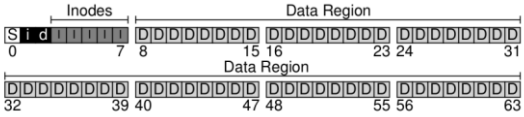
Non potremo utilizzare tutti i blocchi per i dati perché dobbiamo memorizzare anche altre cose, come i nomi dei file (quindi i metadati).



Per ogni file c'è una struttura dati che si chiama i-node, che contiene quasi tutti i metadati (tutti tranne i nomi dei file), e gli i-node sono contenuti in una tabella chiamata tabella degli i-node.

Quanti i-node vado ad allocare quando formatto un disco? A seconda dell'utilizzo del disco può aver senso allocare più o meno i-node.

Dobbiamo definire anche delle strutture dati: **la bitmap degli i-node** e **la bitmap dei blocchi di dati**. La bitmap è un **settore di booleani**, e per ogni i-node ci sarà un bit nella bitmap che mi dirà se l'**i-node** è **occupato o libero**, mentre nella **bitmap dei dati** si saprà se un blocco dati è **utilizzato o libero**. Infine, c'è un superblocco, che è un'intestazione del file-system che identifica le sue caratteristiche (numero di i-node, numero di blocchi dati, ecc.).



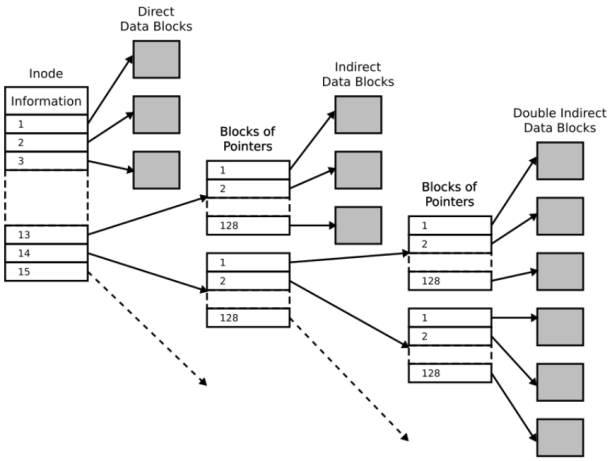
2 dicembre 2024

L'**i-node** è una **struttura dati** fondamentale che mantiene quasi tutti i **metadati** di un file, tranne i nomi. All'interno di un file system alla Unix, possiamo avere più tipi di **file**; il più **semplice** è quello regolare. Dal punto di vista del sistema operativo, **un file regolare non è altro che una sequenza di byte**. Il **formato** del file **non** è un tipo di file, tutti i file tipo jpg, png, pdf, ecc. dal punto di vista del sistema operativo sono file regolari.

Un altro tipo di file molto importante sono le **directory**. A livello utente, una directory è un **contenitore di file regolari o altri directory o altri tipi di file**. Dal punto di vista del sistema operativo, una directory è un **file che contiene le associazioni fra nome e numero di i-node**; quindi, i **nomi dei file** stanno dentro alle directory.

Quando **creo un file**, verrà **allocato un i-node** che non era ancora utilizzato per questo nuovo file; per controllare se un i-node è utilizzato si cerca un i-node con **bit 0** nel suo bitmap. Segno il bit come utilizzato, e dentro alla directory scrivo l'associazione ***nome file* -> *numero i-node***. Possono esserci **più associazioni nome-inode**, ovvero un file può avere tanti nomi.
Un altro tipo di file è il **link simbolico**, un tipo di file il cui contenuto è un **percorso**.
Altri tipi di file sono **FIFO**, detto anche **named pipe**, ossia una pipe con un nome; **socket**; d**ispositivi a carattere** (tastiere, ecc.); **dispositivi a blocchi** (hard disk, ssd, ecc.).

Il primo campo di un i-node specifica il **tipo di file**; poi ci sono un paio di campi che memorizzano **l'utente** e il **gruppo**; poi c'è la **dimensione** in byte; una **maschera** che contiene bit relativi ai **permessi** (R,W,X, set-user-id, set-group-id); delle **date**, come quella di creazione del file, ultima modifica ecc.; dopodiché c'è il numero di **hard link** (più nomi); **puntatori** ai blocchi dati dove trovi i contenuti dei file.
La **dimensione** di un i-node è **fissa**, perché semplifica l'allocazione; questo porta dei problemi quando si devono elencare i puntatori-bocchi dati; un file molto piccolo ha bisogno di uno/due blocchi dati, mentre uno molto grosso ne ha bisogno di molti di più.



Per specificare questi blocchi dati dentro all'i-node si ha un certo numero di **blocchi ad accesso diretto**; per i primi 12 blocchi abbiamo direttamente il numero del blocco che contiene l'informazione. Poi il 13esimo puntatore è utilizzato per indirizzare un blocco che conterrà i puntatori indiretti per altri 128 blocchi. Se non ci bastano questi, abbiamo il 14esimo puntatore che avrà un blocco i cui puntatori punteranno ad altri blocchi ecc. Quindi **ogni volta aggiungiamo un livello di indirezione, e questo ci permette di avere sempre più blocchi indirizzati**.

Le directory sono file, contengono le associazioni tra nome e numero di i-node e in un file system alla Unix ci sono anche due nomi particolari dentro le directory che sono **.** e **. .** :

- **.** rappresenta la **directory corrente**;
- **. .** rappresenta il **parent** della directory.

Il comando **ln(1)** permette di **creare dei link**, sia hard link che link simbolici. Un **hard link** è un'associazione nome-numero di inode; un **link simbolico** è un file che ha un blocco dati associato e quel blocco dati che conterrà un percorso. Si possono creare hard link solo all'interno dello stesso file system; i link simbolici possono essere anche a file che non esistono.

Il comando **rm(1) rimuove il nome di un file**; quando un file non ha più nomi viene eliminato. Se il numero di nomi è 0 ma il file rimane aperto, il sistema lo tiene "in vita" fino a quando il processo che lo usa non termina.

Come vengono risolti i percorsi:
prima di tutto dobbiamo distinguere fra percorsi assoluti e percorsi relativi. Quelli che iniziano con / sono assoluti, tutti gli altri sono relativi. Se un percorso è assoluto, la risoluzione del percorso parte dalla directory root, mentre un percorso relativo parte dalla directory corrente. Ogni processo ha una sua directory corrente e ha una sua root; normalmente la root di un processo corrisponde con la root del file system, ma non è sempre così (chroot). A questo punto, per ogni componente del percorso, separati con /, si cerca nella directory il componente successivo (esempio /qui/quo/qua, sono in root e cerco qui, dentro qui cerco quo, ecc.). Se c'è, vado a recuperarne l'i-node. Se ho un link simbolico, cerco di risolverlo, se riesco procedo, non ci riesco do errore. Dopo un certo numero di tentativi di risolverlo, ci "rinuncia", per evitare loop (tipo A che punta a B, e B che punta ad A, loop).

9 dicembre 2024

Una **singola operazione logica** in realtà potrebbe corrispondere a **tante operazioni di scrittura in posti diversi del file system.**
Cosa succede se, mentre scrivo queste cose su disco, manca la corrente? Il file system può rimanere in uno stato inconsistente. Se io quando vado a creare il file devo allocare un i-inode nuovo, quindi devo modificare la bitmap del i-node, poi manca la corrente, non sono andato a modificare la tabella dell'i-node e la directory; quindi, questo i-node risulterebbe utilizzato anche se non lo è.
Se invece modifico la bitmapdoc, modifico la tabella degli i-node ma non scrivo l'associazione nome-directory, uguale l'i-node risulterebbe utilizzato ma non lo è.
Se mi interrompo dopo aver solo scritto il nome nella directory, mi ritrovo nella directory un'associazione ad un i-node libero, ma quello potrebbe essere usato dopo e quindi avremmo un solo i-node per due associazioni diverse.
Non è possibile trovare un ordine che ci garantisca di mantenere il file system consistente in questo caso.

Una prima soluzione (dal punto di vista storico) è quello di avere dei programmi che vanno a controllare la consistenza, l'integrità di un file system (**fsck**).
Controlla che il superblocco sia "ragionevole"; controlla la consistenza tra i blocchi dati liberi e i puntatori ai file (se ad esempio trovo che un blocco dati sia occupato nella bitmap, ma nessun i-node lo punta, lo marchiamo come libero; se troviamo dei i-node che puntano a dei blocchi che risultano liberi, lo marchiamo come occupato. Se troviamo più i-node che puntano allo stesso blocco, l'unica cosa che si può fare è sdoppiare il blocco e dare ad ogni file la sua copia); si controlla gli stati degli i-node; controlla il link count; controlla che i puntatori non siano fuori dal range dei blocchi: ecc.
Fare questi controlli costa, e più i dischi diventano grossi, più costa; quindi, devono essere fatti solo quando serve. Nel superblocco, tra i tanti metadati che ci sono, c'è anche un dato, **un flag che dice se un file system è montato da qualche parte**. Quando **si spegne il sistema**, lui **smonta** il file system, e poi tira giù la corrente. **Se la corrente manca mente il sistema è in esecuzione, il flag rimane attivo e la prossima volta che si riaccende il sistema il sistema è già montato. Al boot del sistema, esegue l'fsck.**

L'approccio moderno usa il **Journaling**. Alcune sequenze di operazioni devono essere atomiche, o avvengono tutte o non avviene nessuna. Viene creato un **journal**, una **sezione di disco** dove vengono **scritte le operazioni che stiamo facendo**. Quando ho finito la sequenza logica delle operazioni, chiudo la transazione e scrivo sul blocco che è stata chiusa. Se invece scrivo le operazioni che voglio fare ma non le eseguo e chiudo la transazione, poi le faccio davvero. Se il sistema va giù a metà, quando riparte, trova in questo journal cosa volesse fare e ricomincia a farlo fino a quando non chiude la transazione.