

File

giovedì 22 ottobre 2020 11:30

File descriptors

Tutto l'input/output Posix avviene tramite i file descriptors.

I file descriptor sono interi non negativi che identificano un file aperto (in Unix sono trattati come file connessione di rete, pipe, dispositivi...).

Ogni processo ha i propri file descriptor, ma 3 sono utilizzati da tutte le funzioni di libreria per convenzione:

- 0 STDIN_FILENO <-> stdin/cin
- 1 STDOUT_FILENO <-> stdout/cout
- 2 STDERR_FILENO <-> stderr/cerr

System call open

Per lavorare con dei file dobbiamo avere dei file descriptor. Per ottenerne uno, dobbiamo utilizzare la syscall open.

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

Open prende 2 o 3 parametri:

- Il primo è il percorso del file che voglio aprire, può essere relativo o assoluto;
- Il secondo è un intero, i flags: questi sono dei bitmask dove possiamo mettere in OR bit a bit più informazioni diverse (se vogliamo leggere, scrivere, creare il file, se vogliamo che sia bloccante, non bloccante, definire cosa succede se il file esiste già...). Se tra i flag c'è la creazione del file, open va a vedere il terzo parametro;
- Il terzo parametro costituisce la modalità di creazione del file: definisce i permessi associati al file. mode_t mode è un intero e spesso conviene indicarlo in ottale. Vale solo per le prossime aperture: possiamo scrivere in un file appena creato anche se è stato definito il permesso per la sola lettura. Una volta chiuso il file, se lo andiamo a riaprire, potremmo solo leggere.
- Utilizzando la open stiamo inoltre ignorando umask(2), ossia una maschera che definisce permessi che si applicano a tutti i file. Ciò significa che potremmo cercare di attribuire dei permessi ad un file che vengono successivamente negati dalla maschera.

Se tutto va a buon fine, viene restituito un file descriptor e la open garantisce che viene restituito il più piccolo disponibile.

Permessi sui file

Ogni permesso (read, write, execute) è costituito da flag rappresentati da 1 bit.

	u	g	o	
				754
access	r w x	r w x	r w x	
binary	4 2 1	4 2 1	4 2 1	
enabled	1 1 1	1 0 1	1 0 0	
result	4 2 1	4 0 1	4 0 0	
total	7	5	4	

Il numero intero rappresenta 9 bit, formati da gruppi di 3 che rappresentano permessi per il proprietario del file, permessi per il gruppo e permessi per tutti gli altri (cani e porci).

Ognuno di questi 3 flag corrisponde a 1 (2 alla zero), 2 (2 alla uno), 4 (2 alla due). Di conseguenza, per indicare i permessi abilitati, si utilizzano i numeri in ottale.

Questo sito permette di calcolare il numero in automatico specificando con i checkbox cosa vogliamo fare: <https://chmodcommand.com/>

Per cambiare i permessi si utilizza il comando chmod

ESERCIZIO

Scrivete un programma che crea un file (regolare) chiamato pippo

- Che sia leggibile solo dal proprietario (non scrivibile da nessuno, proprietario compreso)
- Cosa succede se il file esiste già?
 - La syscall va a buon fine?
 - Il contenuto del file esistente viene preservato?
 - Potete fare in modo che un file già esistente non venga ri-creato per errore?
- Modificate quel file con un editor di testo (potete farlo? come?)

Makefile

Il makefile è una "ricetta" su come compilare il proprio programma. Permette di descrivere le dipendenze dei file e automatizzare il processo di compilazione, in modo che vengano ricompilati solo i file che dipendono da quelli modificati dall'utente.

Quando viene dato il comando make di default viene eseguita la prima regola.

Per convenzione viene messo un comando all che comprende tutti i file che possono essere compilati. In generale con un makefile possono essere creati più eseguibili: può tornare utile usare delle variabili per contenerli, come EXES.

Un'altra regola presente nei makefile è clean: permette di cancellare tutti i file che si possono ottenere ricompilando.

Possono esserci dei problemi con la regola clean: se nella stessa cartella fosse presente un file chiamato clean, il makefile anziché eseguire la regola proverebbe a compilare il file.

Per evitare questo, si utilizza il comando .PHONY: clean

Questo serve a definire che i target che dipendono da .PHONY (in questo caso solo clean) non sono dei file ma sono dei nomi che abbiamo assegnato ad una serie di azioni.

Scelta del compilatore

All'interno del makefile è possibile utilizzare la variabile \$(CC) per permettere all'utente di scegliere il compilatore da utilizzare. È sufficiente andarla a sostituire a tutti i gcc nel makefile, ed utilizzare all'interno della shell il comando preceduto dall'assegnazione di un valore alla variabile CC

```
CC=gcc make
```

Allo stesso modo possiamo utilizzare una variabile per l'utilizzo di flags di compilazione, come CFLAGS, che permette all'utente di definire non solo il compilatore da utilizzare ma anche eventuali flags aggiuntivi (es -Wall)

```
CC=gcc CFLAGS=-Wall make
```

Sia CC che CLANG possono essere inserite ogni volta dall'utente o definite nel makefile, come nell'esempio.

Esempio: makefile per pippo.c pluto.c pluto.h

```
EXES= pippo  
CC=gcc  
CFLAGS=-Wall -pedantic -Werror
```

```
all: $(EXES)
```

```
pippo: pippo.o pluto.o  
$(CC) $(CFLAGS) -o pippo pluto.o pippo.o
```

```
pippo.o: pippo.c  
$(CC) $(CFLAGS) -c pippo.c
```

```
pluto.o: pluto.c  
$(CC) $(CFLAGS) -c pluto.c
```

```
clean:  
rm -f *.o $(EXES)
```

```
.PHONY: clean
```

Abbreviazioni

Possiamo abbreviare il nome del target, se compreso nella riga di comandi da eseguire, con \$@.

Possiamo abbreviare i file da cui dipende il target, se compresi nella riga di comandi da eseguire, con \$^

Esempio: makefile per pippo.c pluto.c pluto.h con abbreviazioni

```
EXES= pippo  
CC=gcc  
CFLAGS=-Wall -pedantic -Werror
```

```
all: $(EXES)
```

```
pippo: pippo.o pluto.o  
        $(CC) $(CFLAGS) -o $@ $^
```

```
pippo.o: pippo.c  
        $(CC) $(CFLAGS) -c $^
```

```
pluto.o: pluto.c  
        $(CC) $(CFLAGS) -c $^
```

```
clean:  
        rm -f *.o $(EXECS)
```

```
.PHONY: clean
```

Valgrind

Attraverso il makefile e l'esecuzione del programma possiamo non accorgerci di eventuali errori di allocazione e uso della memoria dinamica.

Il programma valgrind permette di effettuare controlli sulle aree di memoria dinamica allocate e su come vengono utilizzate, lanciando il comando valgrind seguito dal nome dell'eseguibile.

```
valgrind ./pippo
```

Nelle informazioni generate da valgrind vengono indicati gli indirizzi esadecimali delle aree dove sono presenti errori di gestione della memoria, ma non il numero di riga dove viene generato il problema all'interno del codice del programma.

Per passare informazioni come numeri di riga o nomi di variabili è necessario aggiungere nel makefile all'interno della lista di flag "-g". Questo permette di lasciare le informazioni di debug.

Valgrind permette di controllare se le aree di memoria allocate vengono liberate, se ci sono accessi al di fuori di aree di memoria allocate, se ci sono più free() riferite ad un solo blocco, ecc.

Si tratta di un'automatizzazione dei controlli che altrimenti dovrebbero essere effettuati manualmente.

Valgrind ha degli svantaggi: trova solo errori nella gestione della memoria tramite malloc() e free(), e rallenta tantissimo l'esecuzione del programma.

Address sanitizer

Switch del compilatore che permette di effettuare in fase di compilazione controlli sugli indirizzi.

Compilare con: -fsanitize=address -g

Permette di trovare più errori ed è più efficiente rispetto a valgrind

Gdb

Permette di trovare il punto in cui avviene un segmentation fault. Può essere utilizzato per eseguire passo a passo il programma tramite dei breakpoint, inserendo il comando:

```
break main.c:8
```

In questo caso inseriamo un breakpoint nel file main.c alla riga 8

Per eseguire il programma utilizziamo il comando run

Per procedere per singoli step (anche all'interno di funzioni) utilizziamo il comando s

Per procedere ad eseguire un'intera chiamata di funzione utilizziamo il comando n (next).

Per separare in due terminali diversi il debug del programma e l'output, è possibile aprire una nuova finestra del terminale e attraverso il comando reptyr -l ottenere il nome di uno pseudo terminale.

Prima di eseguire il programma lancio su gdb il comando tty /dev/pts/*num*

System call read, write e close

Una volta aperto un file attraverso la syscall open() otteniamo un file descriptor che può essere utilizzato per leggere e scrivere sul file, attraverso le system call read e write.

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

Entrambe restituiscono un ssize_t, ossia un intero con segno, che indica il numero di byte letti/scritti. In caso di errore restituiscono -1.

Prendono in input un file descriptor, il puntatore ad un buffer su cui andare a leggere/scrivere, e il numero di byte da leggere/scrivere.

Il valore restituito da read e write può essere inferiore del numero di byte che è stato richiesto di leggere/scrivere. Non si tratta di un errore. 0 indica end of file.

Una volta terminato di utilizzare il file è possibile chiuderlo attraverso la syscall close() passandole il file descriptor del file.

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

Chiudere il file permette di "liberare" il file descriptor che può essere utilizzato per un altro file.

Anche la close può fallire, per esempio nel caso in cui si passi come argomento un file descriptor già chiuso o mai aperto. In ogni caso, anche se fallisce, il file descriptor non è più valido.

Strace

Attraverso il comando strace è possibile vedere le system call effettuate da un processo. Attraverso l'attributo -e è possibile specificare le system call che vogliamo vedere.

ESERCIZIO

Scrivete un "cat dei poveri", ovvero un programma che:

- se invocato senza parametri legge da standard-input
 - altrimenti, dal/dai file specificati da riga di comando
- scrivendo tutto quello che riesce a leggere su standard-output.

Cosa succede se specificate:

- file che non esistono?
- file che esistono ma di cui non avete il permesso di lettura?

Usare system call read() e write(). Utilizzare un buffer di medie dimensioni (circa 1kB) per non effettuare troppe system call (costose!).

File offset

A ogni file aperto è associato un file offset (un "puntatore" tenuto dal kernel) che indica in che posizione siamo arrivati a leggere/scrivere nel file. Viene utilizzato solo per i file regolari, perché non ha senso andare avanti o indietro su un socket, un dispositivo ecc..

Dopo la open del file l'offset è 0; ogni volta in cui viene fatta una read o una write si sposta per tanti byte quanti sono stati letti/scritti (effettivi, non richiesti). È possibile spostare anche il puntatore autonomamente senza fare read o write, tramite un'apposita system call:

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

Dove whence può essere: SEEK_SET, SEEK_CUR o SEEK_END

L'offset può essere anche negativo, per tornare all'indietro nel file. Questo naturalmente non è possibile per i file non regolari (connessioni di rete, stampanti ecc..)

L'offset può essere spostato oltre la fine del file: questo non cambia la dimensione del file

Un file può avere "buchi" che corrispondono (logicamente) a byte a 0. Quando facciamo una read dei buchi leggiamo 0, ma questi zeri non sono effettivamente scritti, per cui non occupano spazio.

Metadati di un file

Per recuperare i metadati di un file (utente proprietario, gruppo, dimensione, data in cui è stato aperto l'ultima volta...) possono essere utilizzate le seguenti system call:

- int stat(const char *pathname, struct stat *statbuf);
- int lstat(const char *pathname, struct stat *statbuf);
- fstat(int fd, struct stat *statbuf);

Tutte queste system call prendono un puntatore alla struttura statbuf precedentemente creato dall'utente, e aggiornano le informazioni della struttura associata al puntatore.

```
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
}
```

Major e minor identificano il dispositivo su cui è presente il file, inode è una struttura che rappresenta un file sul file system...

Da shell è presente un comando stat che stampa queste informazioni.

Introduzione ai sistemi operativi

martedì 22 settembre 2020 16:53

<http://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf>

Il componente hardware che effettivamente esegue le istruzioni è la CPU, attraverso le fasi di fetch, decode, execute:

1. Durante la fase di fetch, attraverso un registro chiamato EIP (Extended Instruction Pointer), la CPU accede alla cella di memoria contenente la prossima istruzione da eseguire.

Registri della CPU

Celle di memoria a cui è associato un nome anziché un indirizzo. Essendo interni alla CPU è possibile accedervi direttamente, senza dover effettuare operazioni di accesso al BUS.

Differenza tra RAM e ROM:

La memoria ROM non è volatile, mantiene il contenuto anche dopo assenza di corrente. È possibile accedervi solo in lettura.

La memoria RAM è una memoria volatile, a cui è possibile accedere sia in lettura che in scrittura.

All'accensione del calcolatore la fase di fetch della CPU parte dalla ROM. Attraverso il BIOS (programma su scheda madre) è possibile avviare un programma salvato in memoria come il sistema operativo o il boot loader (programma che permette di selezionare il sistema operativo da avviare).

Applicazioni e sistema operativo

Utilizzando una singola CPU, con un singolo core di esecuzione (viene eseguita un'istruzione per volta), può essere eseguito un solo programma per volta. Tuttavia attraverso il sistema operativo è possibile "illudere" l'utente, facendo credere di poter eseguire più applicazioni simultaneamente.

Il sistema operativo (kernel) funge da intermediario tra l'hardware e i programmi mandati in esecuzione dall'utente. Permette così di virtualizzare e gestire le risorse presenti sul disco, impedendo l'accesso diretto dei programmi alle risorse hardware. Gestisce inoltre l'utilizzo della memoria e l'utilizzo del disco (SSD e Hard Disk). Tutto questo avviene attraverso delle chiamate di sistema, che possono eseguire azioni specifiche del sistema operativo.

Attraverso la virtualizzazione ogni processo (programma mandato in esecuzione) crede di avere una CPU e una memoria a se dedicata. Nel creare questa illusione il sistema operativo deve tener conto del costo della virtualizzazione e dell'equità della suddivisione delle risorse fisiche tra i vari processi.

Esempio

- *Linux*: kernel, programma del sistema che alloca le risorse hardware ai programmi mandati in esecuzione
- *GNU/Linux*: sistema operativo (compreso di interfaccia utente)

Il sistema operativo permette inoltre di gestire l'interazione tra il programma mandato in esecuzione e le periferiche di sistema (scheda grafica, tastiera...) attraverso l'utilizzo di device drivers.

Se il programma dovesse tener conto delle singole periferiche utilizzate, sarebbe compatibile solo con quelle. Se venisse mandato in esecuzione su dispositivi con periferiche diverse, non riuscirebbe ad essere eseguito.

Device Drivers

Parti di software che girano all'interno del kernel. Permettono l'utilizzo di dispositivi e periferiche da parte del sistema. In modo dinamico il kernel identifica le periferiche utilizzate e carica i driver corrispondenti.

Questo processo di virtualizzazione delle risorse hardware da parte del sistema operativo permette inoltre di fornire protezione ed isolamento fra i diversi processi, che possono accedere solo ad un'area di memoria a loro riservata, e tra i processi e il sistema operativo stesso.

Viene così preservata la confidenzialità e l'integrità dei dati di utenti diversi.

Per fare questo è necessario che il sistema operativo sia robusto e affidabile: un bug nel kernel potrebbe essere sfruttato da applicazioni malevoli e potrebbe permettere ad un processo di prendere il controllo del sistema.

Storia

Primi sistemi

Nei primissimi computer il sistema operativo consisteva in una libreria dove erano implementate le funzioni principali. La scelta del processo da mandare in esecuzione era fatta da un operatore umano, che andava ad inserire manualmente le schede perforate nella macchina.

In questo contesto non vi era alcuna distinzione tra il codice del sistema operativo e il codice del programma, che poteva accedere alle risorse hardware e poteva andare a sovrascrivere parti del sistema operativo.

Per evitare questi problemi è necessario prevedere due modalità di esecuzione distinte, in cui vengono eseguite istruzioni del sistema operativo/kernel oppure viene eseguito codice utente.

Serve quindi poter decidere quale modalità di esecuzione del codice da parte della CPU utilizzare.

Se ci fossero due istruzioni macchina, una per passare alla modalità sistema e una per passare alla modalità utente, qualunque programma potrebbe eseguirla, passando così in modalità privilegiata.

Serve quindi una system call, ossia un'operazione analoga ad una chiamata di funzione, che viene fatta dall'applicazione al sistema. La system call deve prevedere un cambio del livello di privilegio di esecuzione. Deve tuttavia prevedere un controllo, per evitare operazioni illecite da parte dei processi.

La syscall viene implementata attraverso il meccanismo delle interrupt/trap. Quando un programma utente deve passare il controllo al sistema, deve passarlo ad un particolare punto del sistema.

Esempio:

Tutte le volte in cui un programma vuole scrivere su disco, deve farlo attraverso delle system call.

Dentro al codice del sistema operativo ci sarà il codice relativo alle system call open (che permette di aprire il file), read/write e close.

Alla chiamata della system call open su un file, viene effettuata una verifica dei permessi di accesso a tale file. Se il programma potesse saltare a qualsiasi punto del sistema operativo, potrebbe saltare i controlli accedendo a qualsiasi file del sistema.

L'utilizzo di interrupt handler differenti, che rispondono a codici differenti, permette di distinguere e assegnare dei livelli di privilegio a diversi dispositivi. Alcune interrupt potrebbero quindi avere privilegi inferiori ed essere ignorate (temporaneamente) dalla CPU.

Supponiamo che vi siano due interrupt relative allo stesso oggetto: se durante la gestione della prima arrivasse e venisse gestita la seconda, l'oggetto su cui l'interrupt handler lavora potrebbe non essere consistente.

È necessario terminare la gestione prima per poter poi passare alla seconda.

Multiprogrammazione

La multiprogrammazione viene introdotta con l'avvento dei minicomputer, con l'idea di sfruttare i "tempi morti" della CPU, quando vi sono richieste di input/output. Permette di avere più processi in memoria allo stesso tempo, alternando la CPU tra di loro.

Questa alternanza permette di illudere l'utente di poter far girare più programmi in parallelo con una singola CPU e un singolo core.

È sempre necessario tenere conto dei problemi di protezione dei dati e della concorrenza (richieste contemporanee di accesso alle periferiche da parte dei processi).

Unix/Posix

Unix è un sistema operativo nato negli anni '60. Si basa sull'idea di avere poche funzionalità semplici, che possono comporsi per sviluppare funzioni molto potenti.

Si tratta del primo sistema operativo scritto in C (e un po' di assembler): avendo il codice scritto in un linguaggio ad alto livello poteva essere compilato e portato su macchine diverse.

I linguaggi precedenti, scritti completamente in assembler, dovevano essere adattati ad ogni singolo processore.

Fu inoltre il primo sistema operativo distribuito a prezzo ridotto alle università.

Oggi Unix è un marchio registrato, ma ci sono diversi sistemi operativi Unix-like non certificati: nascono come derivati di Unix (Linux, BSD)...

Verso la fine degli anni '80 vi furono diversi tentativi di creare degli standard per interoperabilità tra sistemi diversi. I due standard principali sono POSIX e Single UNIX.

Utilizzeremo per le esercitazioni Linux.

Studieremo all'interno del corso una nostra versione derivata da Xv6, un sistema operativo didattico sviluppato al MIT.

Entrambi non sono sistemi POSIX al 100% ma sono POSIX-oriented.

Introduzione alla shell

giovedì 22 ottobre 2020 12:27

La shell è un interprete di comandi, che può essere utilizzato sia in modalità interattiva (dando dei comandi) o attraverso degli script.

Il nome shell deriva dal fatto che si tratta di un programma utente che offre una interfaccia ad alto livello alle funzionalità del kernel. La shell si occupa infatti di tradurre i comandi dell'utente in system call al kernel.

Faremo riferimento alla GNU bash (Bourne-Again SHeLL). In generale, bisogna fare riferimento al man:

<https://www.gnu.org/software/bash/manual/>

Ad ogni sezione del man è associato un numero. Il numero di sezione può essere utile quando una parola compare in più sezioni differenti.

Per saperne di più leggere il libro "The Linux Command Line"

Terminale

Per utilizzare la shell è necessario un Terminale.

Il terminale è un'evoluzione della telescrittore (in inglese TeleTYpe), un dispositivo meccanico nato per il telegrafo alle fine del 1800. Ancora oggi chiamiamo i terminali TTY, in riferimento alla telescrittore.

La telescrittore era un dispositivo di input/output: l'operatore poteva inserire dei dati attraverso una tastiera e poteva ricevere risposte stampate su un foglio di carta.

I terminali sono poi evoluti negli anni '60, quando la carta viene sostituita da schermi a tubo catodico.

Oggi, in Unix quasi tutto è un file (molte componenti del sistema vengono astratte in file) e file speciali corrispondono ai terminali collegati. Ad ogni terminale fisico viene quindi associato un file.

Il nome TTY è rimasto: /dev/tty è sinonimo del terminale associato ad un processo.

Qualunque processo apra questo file, apre un canale di comunicazione con il proprio terminale: ciò significa che, nonostante il file sia lo stesso, a ogni processo corrisponde un terminale diverso (vedi man 4 tty).

Facendo man tty viene stampata di default la pagina tty della sezione 1 del man (sezione dei comandi). Esiste infatti un comando tty che stampa a schermo il nome del terminale che stiamo utilizzando.

In Linux sono presenti diverse virtual console (ttyn), terminali virtuali che condividono la tastiera e lo schermo. Si può cambiare la virtual console attiva premendo ALT + Fn in modalità testuale o CTRL + ALT + Fn dalla modalità grafica; nelle versioni più recenti di Ubuntu tty1 è il login manager, tty2 l'ambiente grafico e le consolle da tty3 a tty6 sono in modalità testo.

All'interno degli ambienti desktop (come Gnome) si può usare un emulatore di terminale, cioè un programma che emula un terminale testuale.

Il collegamento fra l'emulatore di terminale e un'istanza della shell avviene tramite l'uso di pseudo terminali detti pty.

Pseudo-terminali

I pty (pseudo-terminali) sono coppie di dispositivi a caratteri, coppie di file detti master e slave, che possono emulare i terminali.

- Quando apriamo un terminale grafico, per esempio attraverso Gnome lanciando uno Gnome-terminal, viene creato il lato master dello pseudo terminale
- Al lato master corrisponde uno slave che si comporta come un terminale vero. I processi possono aprire il corrispondente slave e interagire con esso come fosse un terminale vero e proprio. Il lato slave ha nome /dev/pts/... (pts= pseudo terminal slave).

Vedi pty(7) e pts(4).

File descriptor

Un file descriptor è un numero non negativo che corrisponde ad un file aperto dal processo. Ogni volta che un processo vuole utilizzare un file lo deve aprire attraverso la system call open(). Se va a buon fine, open restituisce un numero che costituisce il file descriptor che può essere utilizzato per leggere o scrivere quel file.

I numeri 0, 1, 2 per convenzione hanno un significato particolare.

Ogni processo utilizza tre file descriptor:

- 0 standard input (stdin/cin)
- 1 standard output (stdout/cout)
- 2 standard error (stderr/cerr)

Attraverso questi file descriptor è possibile indirizzare output, input ed errori di un processo.

Normalmente tutti e 3 corrispondono al terminale, ossia quando un processo legge da standard input normalmente legge l'input scritto da tastiera su quella finestra (se legge su uno pseudo terminale). Allo stesso modo output ed error scrivono sul terminale corrispondente.

Differenziando input, output ed errori è possibile stampare ed effettuare letture da sorgenti diverse. Per esempio è possibile indirizzare l'output di un programma su file, mentre eventuali errori vengono stampati sulla finestra dello pseudo terminale corrispondente.

Inoltre, per questioni di performance, gli standard output vengono bufferizzati, mentre gli standard error no.

File system virtuale

Su Linux esiste un file system virtuale chiamato /proc.

Si dice virtuale poiché non risiede su disco, ma viene utilizzato dal sistema per mostrare all'utente informazioni sul sistema stesso. In particolare, per ogni processo del sistema, esiste una sub-directory all'interno di /proc.

Ogni processo del sistema è identificato da un numero intero chiamato PID (Process Identifier).

Digitando il comando echo \$\$ è possibile vedere il PID della shell utilizzata in quel momento. All'interno di /proc/\$\$/fd (comando ls -l /proc/\$\$/fd) è possibile vedere i file descriptor aperti di questo processo.

Notiamo come i file descriptor corrispondano allo pseudo terminale, che si comporta da input/output ed error della shell.

Indirizzare standard output e standard input

È possibile indirizzare lo standard output attraverso il comando >

Un comando per scrivere semplicemente del testo è echo. Scrivendo il comando echo > /dev/pts/... è possibile scrivere del testo su un terminale e vedere il testo stampato su un altro.

Attraverso il comando cat è possibile copiare da standard input a standard output. Può essere utilizzato per indirizzare l'output su un pts diverso attraverso il comando cat > dev/pts/ ...

Allo stesso modo possiamo utilizzarlo per ottenere l'input da un'altra shell, tuttavia entrambe entrano in competizione per l'acquisizione dei caratteri dallo stesso terminale.

cat < dev/pts/...

È possibile utilizzare il comando reptyr -l per creare un nuovo pseudo terminale che non viene utilizzato dalla shell. In questo modo utilizzano il comando cat per l'acquisizione di caratteri le due shell non entrano in conflitto tra di loro.

Esempio: lanciare comando da finestra 1 per leggere input da finestra 2 e stamparlo su finestra 3

Finestra 1 /dev/pts/0
cat < /dev/pts/3 > /dev/pts/2

Finestra 2 /dev/pts/1
reptyr -l
(Opened a new pty: /dev/pts/3)

Finestra 3 /dev/pts/2

Indirizzare standard error

Eseguendo il seguente comando:

```
ls -l questo_file_non_esiste > pippo 2> pluto
```

La shell proverà ad elencare il contenuto di un file inesistente, stampando l'output nel file pippo e l'errore generato nel file pluto

Eseguendo questo comando:

```
ls -l questo_file_non_esiste > pippo
```

L'output verrà stampato su pippo e l'errore verrà mostrato a terminale.

Directory di un sistema Unix

/	Radice del file system
/bin e /sbin	Comandi essenziali e comandi per l'amministrazione
/boot	File per il boot di sistema
/dev	file speciali che corrispondono a dispositivi (es: terminali)
/etc	File di configurazione del sistema
/home e /root	Home degli utenti non root e root
/lib*	Librerie
/media e /mnt	Mount-point per i media removibili
/proc e /sys	File system virtuali che forniscono un'interfaccia alle strutture dati del kernel
/tmp	File temporanei, spesso un ramdisk nei sistemi moderni Ramdisk= disco che sta in ram, viene resettato ad ogni riavvio del sistema
/usr	Gerarchia secondaria (/usr/[s]bin, /usr/lib*, . . .), che può essere condivisa in sola lettura fra più host per risparmiare spazio

Disciplina di linea

Tra pseudo terminali e processi c'è la disciplina di linea. Ciò significa che in alcune modalità il kernel si interpone tra il terminale e il processo e gestisce ciò che viene scritto.

Quando scrivo qualcosa sul terminale, ha effetto solo quando premo invio. Questo permette di editare la linea senza che il processo debba intervenire.

Esistono programmi che non vogliono avere un buffering, ma vogliono ricevere immediatamente ogni comando. Per disattivare la versione canonica si usa il comando stty -icanon.

Per riattivarla si utilizza stty icanon.

- Normalmente si usa la versione cooked/canonical che gestisce il buffering, l'editing, l'echo, caratteri speciali (backspace, CTRL + C/D/Q/H/S/Z...) vedi stty(1)
- Alcune applicazioni come Vi, utilizzano la versione raw
- Provare stty -icanon && cat

La modalità canonical permette la gestione di backspace, buffering etc da parte del kernel e non dai singoli programmi.

Programma esempio in c

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main()
{
    ssize_t n;

    do
    {
        char buf[1024];
        memset(buf, 0, sizeof(buf));
        n= read(STDIN_FILENO, buf, sizeof(buf)-1);
        printf("Got %ld bytes: %s\n", (long)n, buf);
    } while (n>0);

}
```

/* read è una system call che permette di leggere un certo numero di byte da un file e salvarli in un buffer utilizzando un file descriptor. Per leggere e scrivere da file su Unix abbiamo bisogno di un file descriptor. La read restituisce il numero di byte che effettivamente è riuscita a leggere */

Sequenze di escape

Sequenze di caratteri che iniziano con Esc (ASCII 27) che servono per mandare dei comandi al terminale.

Esempio: echo -e '\x1b[38;5;123mciao'

Stampa ciao in azzurro

Attraverso il comando -color è possibile colorare l'output

Esempio: ls -color, ls -color | less -r

Job control

giovedì 22 ottobre 2020 12:29

Job control è un meccanismo che permette di gestire più lavori con uno stesso terminale, sospendendo/riprendendo l'esecuzione di gruppi di processi, chiamati anche job.

Si tratta di un meccanismo importante per i terminali veri, un po' meno per gli emulatori di terminale, dove basta aprire un'altra finestra.

Possiamo per esempio, quando mandiamo in esecuzione un comando molto lungo, fermarlo temporaneamente con `ctrl+z`. In questo modo il job è fermo, e possiamo farlo riprendere in background con `bg`. Di default, il comando `bg` senza ulteriori argomenti, manda in background l'ultimo job fermato. Con il comando `fg` possiamo riportarlo in foreground.

Attraverso il comando `jobs` possiamo vedere lo stato di tutti i job presenti sul terminale. Con il comando `kill %numero` possiamo terminare i job in background. Per esempio, se il nostro job è il numero 1, possiamo terminarlo con `kill %1`.

I job sono raggruppati in sessioni, una per terminale.
Per ogni pipeline inserite nella shell, viene creato un job.

Sessioni

Quando apriamo un terminale c'è una cosiddetta sessione associata, all'interno della quale possiamo avere tanti gruppi di processi (anche detti job).

Ogni volta in cui lanciamo un comando dalla shell viene creato un gruppo di processi: per esempio quando lanciamo il comando `ls | grep pippo` viene creato un gruppo di processi contenente al suo interno i processi per `ls` e `grep pippo`.

È possibile che questi comandi generino a loro volta dei processi, che restano sempre all'interno dello stesso gruppo (job). Possiamo quindi gestire tutto il gruppo, mandandolo in background o facendogli arrivare segnali come `SIGINT` (`ctrl+c`).

Ci può essere solo un gruppo di processi in primo piano, mentre tutti gli altri sono in background.

Quando chiudiamo la finestra del terminale il kernel manda un segnale `SIGHUP` alla shell, che a sua volta lo manda a tutti i gruppi di processi, terminandoli.

È possibile mandare in esecuzione dei processi indicando esplicitamente di ignorare il segnale `SIGHUP` tramite il comando `nohup` posto prima del nome dell'eseguibile: il processo continuerà ad essere eseguito anche dopo la chiusura della finestra di terminale e il suo output si troverà nel file `nohup.out`.

Bug tipici

domenica 20 dicembre 2020 14:56

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf>

Del codice corretto in un mondo single-threaded, può avere molti problemi quando si passa a in un mondo multi-threaded. Di seguito vengono riportati alcuni esempi.

Violazione dell'atomicità

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

Questo codice è corretto e funzionante in un processo con un singolo thread, ma in un processo con più thread potrebbe avvenire una violazione dell'atomicità: se un thread entrasse nell'if, ma prima di eseguire l'istruzione contenuta all'interno del blocco un altro thread impostasse il puntatore a null, si avrebbe un errore.

Questo potrebbe essere risolto mettendo l'acquisizione di un mutex prima dell'if , che viene rilasciato dopo la sua esecuzione. In questo modo l'operazione viene resa atomica.

Violazione dell'ordine

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

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

Supponiamo di avere una variabile globale glob_thread che corrisponde ad un certo thread. Tramite la chiamata alla funzione create_thread viene creato il thread associato alla variabile globale e la sua esecuzione viene fatta partire dalla funzione thread_func. All'interno di questa funzione viene letta una variabile di questo thread.

Il problema è che, nel momento in cui viene eseguita la funzione create_thread, viene creato un thread pronto per essere eseguito. Dal punto di vista dello sceduler ci sono quindi due thread pronti per essere eseguiti, ed entrambi possono andare in esecuzione.

Se viene mandato in esecuzione il thread che ha effettuato la chiamata, allora viene inizializzata la variabile globale e il nuovo thread potrà accedere ai suoi campi. Viceversa, se venisse mandato in esecuzione per primo il nuovo thread, questo proverebbe ad accedere ai campi di una variabile non ancora inizializzata.

Deadlock

Consideriamo due thread:

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

Questi due thread lavorano su due strutture globali, ognuna delle quali ha il proprio mutex.

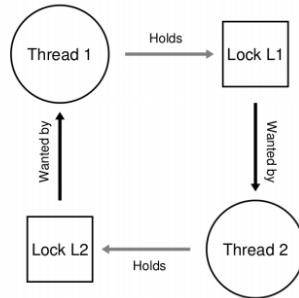


Figure 32.2: The Deadlock Dependency Graph

Può succedere che il Thread-A acquisisca il lock m1, e poi venga deschedulato e venga mandato in esecuzione il Thread-B.

Il Thread-B acquisisce il lock m2, ma vorrebbe anche il lock m1. Allora viene nuovamente mandato in esecuzione il Thread-A, che tuttavia si blocca perché vorrebbe il lock m2.

Se succede questa catena di richieste circolari si entra in un deadlock, e non c'è possibilità di uscirne. Il sistema si blocca (senza consumare CPU, perché tutti i thread restano in attesa).

Condizioni necessarie per i deadlock:

- mutual exclusion: i thread hanno controllo esclusivo
- hold-and-wait: i thread mantengono le risorse acquisite mentre ne richiedono/aspettano altre
- no preemption: le risorse (lock acquisiti) non possono essere tolte
- circular wait: ci deve essere un ciclo nelle attese

Come prevenirlo? I modi più semplici sono:

- Imporre un ordine sui lock e acquisirli in ordine (no circular wait)
- Fare in modo che i lock vengano sempre acquisiti tutti assieme, atomicamente (no hold-and-wait)

Il problema nell'esempio è che i thread prendono i lock in ordine inverso (uno prende prima m1 e poi m2, il secondo prende prima m2 e poi m1). Se entrambi i thread prendessero i lock nello stesso ordine non si creerebbe una situazione di attesa circolare.

Introduzione

lunedì 7 dicembre 2020 17:44

Abbiamo visto come il kernel virtualizzi le risorse: in particolare virtualizza la CPU (ogni processo crede di avere la CPU per sé) e virtualizza la memoria tramite la paginazione (ogni processo ha il proprio spazio di indirizzamento isolato dagli altri).

I thread permettono di avere più flussi di esecuzione in un processo: in un certo senso creare un thread significa creare una "CPU virtuale" che gira assieme ad altre nello stesso spazio di indirizzamento.

La differenza principale fra thread e processi è la condivisione (o meno) dello spazio di indirizzamento: due processi diversi hanno due CPU virtuali e due spazi di indirizzamento diversi, mentre due thread in un processo vedono lo stesso spazio di indirizzamento.

Da un certo punto di vista questo meccanismo è vantaggioso, perché è possibile scambiare puntatori e indirizzi di memoria tra i thread, ma dall'altro necessita di sincronizzazione per gestire correttamente l'area di memoria condivisa.

I thread, come i processi, vanno schedulati. Linux non fa differenza tra thread e processi: entrambi sono per lui oggetti schedulabili.

Il context-switch fra thread costa meno del context-switch fra processi: questo perché condividendo lo spazio di indirizzamento tutti i thread dello stesso processo utilizzano la stessa tabella delle pagine.

Inoltre ogni thread, corrispondendo ad un flusso di esecuzione, ha il proprio stack (ma condivide codice e dati con gli altri thread dello stesso processo). Attenzione: tutti gli stack sono contenuti nello stesso spazio di indirizzamento: teoricamente un thread potrebbe accedere allo stack di altri thread - ma non si deve fare!)

Perché usare i thread

I thread permettono di sfruttare il concetto di parallelismo: supponiamo di avere una CPU con 8 core e dover fare dei conti utilizzando un core. Non avrebbe senso lasciare gli altri 7 inutilizzati: avrebbe più senso suddividere il lavoro in più thread che lavorano su più aree del problema.

Esempio: array con un miliardo di elementi da decrementare

Utilizzando un solo thread questo deve agire da solo su tutti gli elementi, usando più thread ognuno agisce su una porzione dell'array, completando l'operazione in meno tempo.

Il numero di thread da utilizzare dipende dal numero di core della CPU: se ho 4 core, non ha molto senso avere più di 4 thread.

Un altro caso in cui è utile usare i thread è quello di evitare di bloccarsi quando si fa I/O: i thread permettono infatti di continuare ad eseguire operazioni mentre si attende l'input dall'utente (ad esempio). Non si tratta dell'unico modo per continuare a lavorare durante le operazioni di I/O ma è il più semplice.

Creazione di thread

Per la creazione di thread consideriamo le API standard di POSIX

```
#include <pthread.h>
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);

// Compile and link with -pthread
```

Argomenti:

1. Puntatore a pthread_t (analogo al pid dei processi): identifica il nuovo thread che viene creato
2. Puntatore a pthread_attr_t: permette di specificare ulteriori esigenze – normalmente si può indicare null
3. Puntatore a una funzione che prende un void* e restituisce un void*: corrisponde alla funzione da cui parte l'esecuzione nel nuovo thread
4. Argomento passato alla funzione da cui parte l'esecuzione nel nuovo thread



Valore di ritorno

Restituisce 0 quando ha successo. Se c'è un errore restituisce direttamente il codice di errore (non lo scrive in errno).

Terminazione e attesa

Per uscire con un certo valore da un thread possiamo:

- Fare return dalla funzione iniziale;
- Chiamare all'interno di una qualsiasi funzione eseguita dal thread la syscall
`void pthread_exit (void *retval);`

Per attendere poi la terminazione con la system call

```
int pthread_join (pthread_t thread, void **retval);
```

Quando creiamo due thread, questi diventano immediatamente ready, e sta allo scheduler decidere quale mandare in esecuzione. Potrebbe quindi decidere di mandare in esecuzione il primo che è stato creato o il secondo. Da notare che, per ogni processo esiste anche un thread di default: in totale i thread sono quindi 3.

Quando il thread di default fa una join si mette in attesa della terminazione di un altro thread precedentemente creato. Potrà proseguire solo dopo la sua terminazione.

Esempio threads.v0.c

Programma che, dato un valore passato come argomento, crea due thread che eseguono la stessa funzione incrementando lo stesso contatore fino al valore passato come argomento.

Il programma funziona per valori dell'ordine di grandezza di 10, 100, ma a partire da 1000 comincia a non funzionare correttamente.

Questo perché, se uno dei due thread viene deschedulato e viene mandato in esecuzione l'altro (che esegue una serie di incrementi), quando viene nuovamente mandato in esecuzione il primo questo riparte da dove era rimasto, sovrascrivendo il contatore e cancellando gli incrementi dell'altro thread.

Sezioni critiche e race-condition

Una sezione critica è un frammento di codice che accede ad una risorsa condivisa.

Quando si hanno più flussi di esecuzione si parla di race-condition quando il risultato finale dipende dalla temporizzazione con cui vengono schedulati ed eseguiti i flussi di esecuzione: questo avviene per esempio quando due thread eseguono più o meno allo stesso tempo una sezione critica, e il risultato comporta che la computazione sia non-deterministica.

Per evitare questi problemi serve una sincronizzazione tra i thread, alle volte anche di processi diversi. Esistono tante primitive di sincronizzazione. Analizzeremo solo quelle per la mutua esclusione.

Primitive di sincronizzazione

venerdì 11 dicembre 2020 14:28

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf>

Un Lock è un oggetto che permette di sincronizzare due thread e rendere atomiche le operazioni, ossia ci permette di far sì che solo un thread alla volta entri in una sezione critica.

Prima di iniziare l'esecuzione di una sezione critica il thread deve quindi ottenere il lock; se questo non è disponibile deve attendere.

In POSIX i lock vengono implementati tramite i mutex. I lock quindi:

- Vengono dichiarati tramite pthread_mutex_t
- Possono essere inizializzati tramite:
 - Assegnazione della costante PTHREAD_MUTEX_INITIALIZER
 - Pthread_mutex_init
- Possono essere acquisiti (presi/tenuti) da un thread alla volta tramite pthread_mutex_lock
- Vanno rilasciati il prima possibile tramite pthread_mutex_unlock

Numero di lock da utilizzare

In generale il numero ideale di lock da utilizzare in un programma è uno per ogni struttura dati utilizzata. Avere uno stesso lock per più strutture dati, su cui si agisce con funzioni diverse, non avrebbe senso e comporterebbe un rallentamento del programma.

Dall'altro lato bisogna considerare che un numero di lock troppo elevato causa un overhead elevato che comporta forti rallentamenti nell'esecuzione.

Per esempio tutte le strutture dati in Xv6 sono potette da un lock (la process table, la file table...), ossia quando un pezzo di codice vuole agire su una di queste strutture deve ottenere il lock.

Implementazione di lock

Per costruire dei lock e mettere a confronto implementazioni diverse dobbiamo tenere conto di:

- Mutua esclusione (funzionano correttamente?)
- Fairness/starvation
- Performance

Analizziamo le diverse implementazioni.

Disabilitare – Riabilitare le interruzioni

Una prima possibilità è quella di disabilitare gli interrupt quando vogliamo fare il lock e riabilitarli quando vogliamo toglierlo. I problemi che abbiamo visto erano dovuti al fatto che il kernel interrompesse un thread mentre stava modificando una struttura dati condivisa, e non prima o al termine della modifica.

Questa implementazione tuttavia comporta una serie di problemi:

- Le istruzioni per abilitare/disabilitare gli interrupt sono istruzioni privilegiate;
- Permettere a un processo di disabilitare le interruzioni e monopolizzare la CPU è rischioso;
- Nei sistemi multiprocessore/multicore non funziona perché due core possono eseguire la stessa sezione critica.

Dobbiamo quindi pensare ad un'altra implementazione.

Struttura dati

Proviamo ad implementare una semplice struttura dati per la gestione dei lock, che indica se il lock è disponibile (0) o non è disponibile (1).

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}
```



```

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}

```

Supponiamo che due thread stiano aspettando il lock. Nel momento in cui si libera, il primo thread in attesa esce dal while, ma prima che possa impostare il lock lo scheduler lo ferma e fa partire il secondo thread, che prende il lock. Nel momento in cui lo scheduler fa ripartire il primo thread, ci troviamo con due thread con il lock ad 1, che possono agire contemporaneamente su sezioni critiche.

Questa struttura dati non garantisce quindi la mutua esclusione, e di conseguenza non funziona.

Supporto hardware

Per implementare correttamente il meccanismo dei lock abbiamo bisogno di primitive atomiche fornite dai processori.

Diversi processori forniscono primitive atomiche quali:

- Test-and-Set (equivale alla funzione lock, ma si tratta di un'istruzione atomica)
- Scambi atomici (scambio di valori in modo atomico, implementato da Xv6)

Su x86 abbiamo l'istruzione XCHG che corrisponde a:

```

/* pseudo-code (nel x86, istruzione XCHG) */
int AtomicExchange(int *ptr, int new)
{
    int old = *ptr;
    *ptr = new;
    return old;
}

```

Con queste primitive atomiche possiamo implementare gli spin-lock

Spin-lock - implementazione

```

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;
}

```

Questo meccanismo garantisce mutua-esclusione.

Tuttavia non è fair: un thread in attesa non ha qualche garanzia di ottenere, prima o poi, il lock, ed è possibile starvation .

Per quanto riguarda la 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
- multipli core: funziona discretamente bene, perché se i thread sono stati scritti bene non tengono il lock a lungo.

Ha senso aspettare in un loop (consumando CPU)? Se si aspetta poco sì, perché i context-switch costano. Negli anni ci sono state varie proposte di approcci ibridi, ovvero "lock in due fasi": si fa un po' di spin cercando di ottenere il lock, poi eventualmente se non si ha successo si va in sleep. Il thread verrà poi risvegliato quando verrà liberato il lock.

Spin lock – Inversione delle priorità

L'inversione delle priorità è un fenomeno insolito che ha a che fare con un'interazione tra scheduler e lock (spin lock). Nei sistemi complessi possono esserci meccanismi che funzionano perfettamente di per sé, ma generano interferenze se utilizzati con altri meccanismi.

Consideriamo uno scheduler a priorità e due thread:

- T1 bassa priorità, T2 alta priorità
- Supponiamo che T2 sia in attesa di qualcosa, va in esecuzione T1
- T1 acquisisce un certo lock L
- T2 torna ready
- Lo scheduler deschedula T1 e manda in esecuzione T2
- T2 va in spin-wait per il lock L
- Game Over

Per risolvere questo problema si fa "ereditare" al thread che possiede il lock, la priorità del thread in attesa (se maggiore di quella corrente). In questo modo il thread che possiede il lock va in esecuzione, fa quello che deve fare, e una volta terminato rilascia il lock e torna alla sua priorità precedente.

Dispositivi a blocchi

lunedì 21 dicembre 2020 12:49

<http://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf>

Supporti di memoria

Chiamiamo comunemente "dischi" i dispositivi a blocchi di memoria secondaria come:

- dischi magnetici (hard/floppy disk di vario tipo)
 - La formattazione a basso livello prepara questa struttura logica sui dischi magnetici (ma, da decenni, sono venduti "pre-formattati")
- dischi ottici (DVD, Bluray, . . .)
- "pennette" USB
- SSD: Solid-state Storage Device/Drive/Disk/. . .

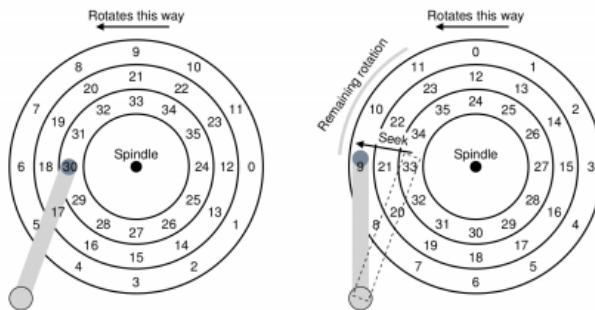
Dal punto di vista logico/astratto, sono tutti una sequenza di settori, tipicamente da 512 byte l'uno.

Dischi

Una volta questi dispositivi erano dischi, cioè oggetti piatti e di forma circolare.



In questi dischi accedere ai dati più vicini costava meno che accedere ai dati più lontani, poiché vi era un ritardo dovuto alla rotazione fisica del disco e allo spostamento della testina da una traccia all'altra.



Questa è la ragione per cui si accede a blocchi: se leggessimo un byte alla volta, per leggere quello successivo dovremmo di nuovo aspettare un'intera rotazione.

Interfaccia

Un SSD moderno non è rotondo, non ruota ecc.

Tuttavia alcune convenzioni, la terminologia e l'interfaccia sono sostanzialmente rimaste: un sistema operativo potrebbe usare un SSD che esporta un'interfaccia da HD senza saperlo (anche se non è ottimale).

L'interfaccia semplificata che consideriamo è un disco con n settori/blocchi che possiamo indirizzare con indirizzi da 0 a n-1. La granularità di accesso è il settore da 512 byte.

Buffer cache

L'I/O su disco è ordini di grandezza più costoso dell'accesso in RAM.

Per minimizzare le letture e le scritture su disco i sistemi Unix-like utilizzano una (unified) buffer cache. Si tratta di una memoria cache tenuta dal sistema operativo, in cui vengono copiati i settori su cui si vuole agire.

Una volta le cache erano due:

- buffer-cache per i blocchi disco acceduti tramite read/write
- page-cache per i file mmappati (system call mmap) in memoria; che, a sua volta, si appoggiava sulla buffer-cache

Su Linux, per forzare la scrittura dei dati/metadati in cache si usa la system call sync(2).

La presenza di memorie cache è la ragione per cui è importante "espellere" (=smontare) i drive prima di scollarli

Partizioni

Su un dispositivo a blocchi potrebbero voler essere installati sistemi operativi diversi. Per evitare che i sistemi operativi cerchino di prendere il controllo dello stesso disco, c'è la possibilità di suddividere il disco in più partizioni.

Partizionare un disco significa suddividerlo in parti logiche che si comportano come drive a sé stanti che possono essere usati in modo separato. Questo potrebbe essere fatto per avere più sistemi operativi diversi, oppure per avere dei dati condivisi tra i diversi sistemi operativi che risiedono su una partizione particolare.

Per partizionare un disco possono essere utilizzati due standard:

- **Master Boot Record (MBR):** si tratta dello standard più antico ma il più compatibile. Prevede l'uso di 4 partizioni primarie (identificate tramite il numero corrispondente) più altre partizioni estese.
- **GUID Partition Table (GPT):** non ha limiti di partizioni e ogni partizione viene identificata da un UUID (Universally Unique Identifier – Identificatore univoco universale).

Utilizzare un identificatore univoco è più efficiente: permette di non avere problemi ad identificare le partizioni nel momento in cui una di queste viene eliminata (quando due partizioni vengono unite tra di loro); oppure, se le partizioni sono legate al disco che viene utilizzato, nel momento in cui questo viene fisicamente staccato e collegato ad un altro bus, senza l'uso di un UUID vediamo dei numeri diversi che non rendono facile identificare una partizione.

L'identificatore univoco viene generato ogni volta che viene creata una partizione, e la probabilità di conflitto è talmente bassa che possiamo assumere che sia unico al mondo, per cui ci permette di identificare una partizione indipendentemente dal bus su cui è collegato il disco.

Nei sistemi Unix-like anche i dispositivi di I/O vengono gestiti tramite dei file speciali, che possono essere creati per dispositivi a caratteri o per dispositivi a blocchi. I file speciali identificano un dispositivo tramite due numeri, il **major number** e il **minor number**.

Il **major number** identifica la classe del dispositivo: in un certo senso identifica il driver, per esempio tutti gli hard disk sata presenti avranno lo stesso major number.

Il **minor number** identifica il particolare dispositivo di quella categoria. Per esempio, se abbiamo 7 hard disk, avremo 7 file speciali con lo stesso major number e minor number rispettivamente 1, 2, 3...7.

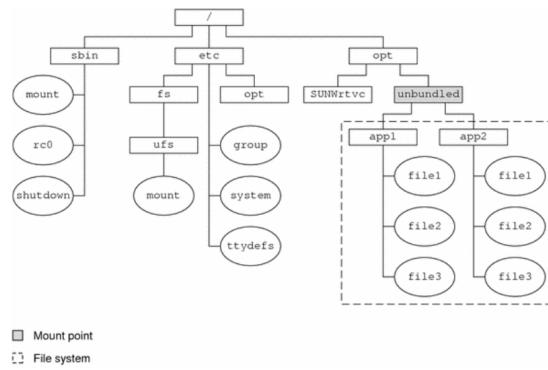
In Linux /dev/sda corrisponde ad un intero disco, mentre /dev/sda1 , /dev/sda2 ecc. corrispondono alle partizioni (vedi comando ls -l /dev/sda*).

Siccome identificare una partizione tramite un numero non è bello, all'interno di /dev/disk/ troviamo diverse caratteristiche che ci permettono di identificare il disco e le relative partizioni.

Questi file che identificano dispositivi possono essere creati utilizzando il comando mknod da root. È importante che possano essere gestiti solo da root o dagli utenti del gruppo disk perché questi file permettono di interagire direttamente con il disco, senza nessuna mediazione del sistema operativo. Per utilizzare un disco su Unix questo deve prima essere montato tramite mount. Per essere montato deve prima essere formattato.

Mounting/unmounting

L'operazione di mount permette di agganciare l'albero delle directory e dei file del disco all'albero globale, su un mount point.



Quando il sistema parte monta in automatico il root file system. A questo file system possiamo montare altri file system di dispositivi come hard disk, chiavette ecc., e va fatto se vogliamo accedere ai file contenuti all'interno di quei dispositivi. Il nuovo file system verrà agganciato ad una particolare directory già presente all'interno del file system attuale.

In questo modo su Unix si avrà sempre un unico file system, al contrario di quello che succede su Windows. Su un sistema Linux moderno di default i file system vengono montati in automatico.

Gestione file speciali

Con l'aumentare del numero di dispositivi supportati dal kernel e l'arrivo dei dispositivi hot-plug (ad es. chiavette USB), il sistema mostrava i suoi limiti: venivano creati migliaia di file all'interno della directory /dev a priori, perché supportati dal kernel. Inoltre vi erano problemi riguardo alla gestione dei dispositivi hot-plug, perché non era possibile prevedere la creazione dei relativi file all'interno di /dev per questi dispositivi.

Per questi motivi erano necessari degli aggiornamenti.

Ad oggi il sistema parte con uno /dev (che è un tmpfs, ossia fa finta di essere un file system ma risiede in ram) vuoto. Quando vengono collegati o scollegati dei dispositivi il kernel genera degli eventi che vengono monitorati da processi utente, che tramite udev(7) possono creare o rimuovere dei file speciali. Questi processi utenti possono anche andare a caricare o scaricare dei moduli del kernel che corrispondono ai driver per quel dispositivo.

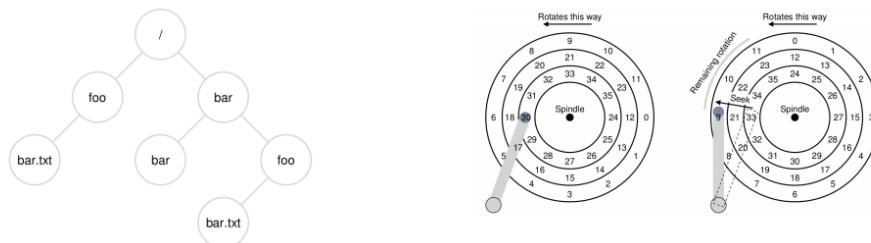
Questo meccanismo può andare a notificare il file manager, per cui i dispositivi possono essere montati automaticamente appena collegati, tipicamente sotto /media/username/label (dove username= nome utente, label= nome dispositivo).

File e directory

Gli utenti di un sistema operativo usano le astrazioni di file (sequenze di byte) e di directory (contenitori logici di file e directory - ricorsivamente) a cui sono associati dei nomi.

Come possono essere implementate le astrazioni di file/directory su un dispositivo a blocchi?

L'utente quando va a scrivere un file non si preoccupa del settore in cui andrà a scrivere e del numero di byte che questo può contenere, ma scrive e legge semplicemente una sequenza di byte.



Trovare settori non utilizzati e andare a scrivere al loro interno tante sequenze di byte sarà compito del sistema operativo.

File system

lunedì 21 dicembre 2020 18:02

Implementazione

<http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf>

Un file system è una struttura dati che risiede su un dispositivo a blocchi.

Ci permette di memorizzare sul dispositivo a blocchi dati e metadati (nome del file, dimensione, ultima modifica, dove sono le sequenze di byte...). Possono esserci strutture dati diverse per gestire queste informazioni e come vengono memorizzate su disco, per cui esistono tanti diversi tipi di file system.

Formattare, rispetto ad un certo formato, un disco significa andare a preparare questa struttura dati sul disco. Andiamo a scrivere la struttura dati relativa al disco vuoto, per cui saranno presenti metadati che indicano che non sono presenti informazioni sul disco ed è presente una certa quantità di spazio libero per scriverci dentro.

Per formattare su Linux si usano i comandi mkfs.*

Esistono vari formati come FAT, FAT32, NTFS, ext2/3/4, ...

Noi consideriamo una versione semplificata di un file system "alla Unix", che il libro chiama vsfs (Very Simple File System).

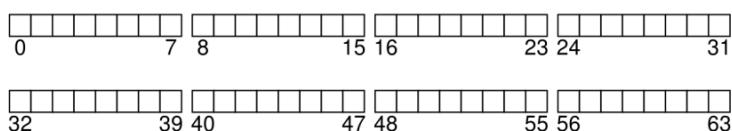
Non dimentichiamoci che sono necessarie anche altre strutture dati, per gestire l'accesso ai file da parte dei processi utente. Per esempio, quando un processo usa open, fra le altre cose il kernel deve:

1. Recuperare l'inode (struttura su disco) corrispondente al percorso specificato come stringa
2. Allocare una struttura che corrisponde al file aperto, che "punta" a (1)
3. Allocare un FD nel PCB del processo, che "punta" a (2)

Ci sono quindi strutture che stanno su disco (come inode, struttura dati che contiene i metadati di un file, che risiede su disco e viene portata in RAM quando serve), ma anche altre strutture come la tabella dei file descriptor e la tabella dei file aperti che risiedono solamente in RAM.

Organizzazione

Assumiamo di avere un disco, o partizione, di 64 blocchi da 4KB. Spesso infatti i settori da 512 byte si raggruppano in blocchi logici, detti cluster. La misura dei blocchi viene fatta appositamente coincidere con la dimensione delle pagine in x86.

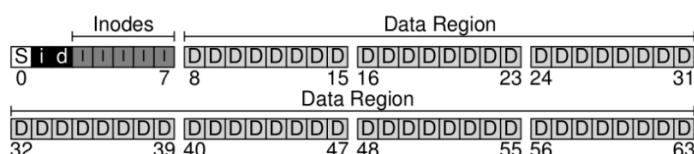


In un disco non possiamo utilizzare tutti i blocchi disponibili per i dati, poiché vi andranno inseriti anche i metadati. Deve quindi essere diviso almeno in due zone: una per i dati (Data Region) e una per il resto. Questa informazione è importante perché stabilisce quanto spazio viene riservato per ogni categoria di informazioni da memorizzare.

Dati e metadati

Per ogni file su disco c'è una struttura dati chiamata **inode**.

Gli inode sono contenuti all'interno della **tabella degli inode**: una serie di blocchi sarà riservata alla sua memorizzazione. Ci sarà quindi da fare una scelta sulla dimensione da riservare per questa tabella, e questo implica il numero massimo di inode che possono esserci e, di conseguenza, il numero file che possono essere memorizzati nel sistema.



Per ogni inode bisogna sapere se è stato usato o meno. Analogamente, per ogni blocco dati, bisogna sapere se è già stato utilizzato o se è disponibile. Nei file system alla Unix sono utilizzate delle bitmap per memorizzare queste informazioni: ci sarà una *inode bitmap* ed una *data bitmap*.

Le bitmap sono delle sequenze di bit, dove ad ogni bit corrisponde un inode o un blocco dati. Infine, all'inizio di un file system, è presente un superblocco per identificare il tipo di file system e le sue caratteristiche (numero di inode, numero di blocchi dati ecc..).

Vengono utilizzate delle bitmap al posto di strutture dati come delle liste, poiché l'accesso al disco è molto costoso ed effettuare la lettura di una lista da disco avrebbe un costo troppo elevato.

Inode

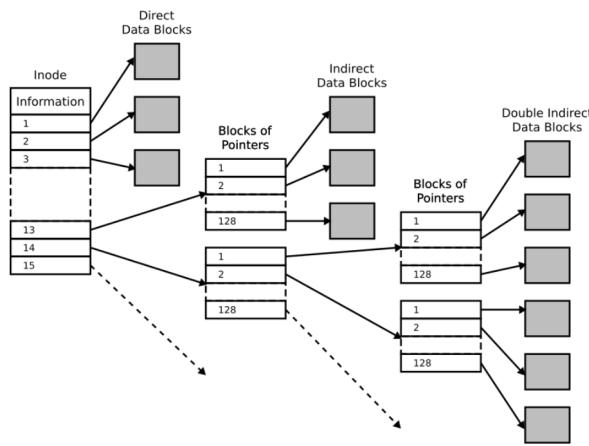
Ogni inode contiene i metadati relativi a un file (ma NON il nome):

- tipo di file, che può essere:
 - regular file
 - directory
 - symbolic link
 - FIFO (tipi di pipe)
 - socket (di tipo locale, corrispondono ad una specie di pipe)
 - character device
 - block device
- UID/GID di proprietario e gruppo
- dimensione in byte
- maschera dei bit relativi ai permessi
- date di creazione/modifica/...
- numero di (hard) link — ne parliamo fra poco
- "puntatori" ai blocchi dati
- ...

Puntatori ai blocchi dati

Un file può essere più o meno lungo, e può quindi usare un numero arbitrario di blocchi dati. Tuttavia la tabella degli inode è di dimensione fissa, e in particolare ogni inode ha una dimensione fissa. Dobbiamo quindi usare un metodo che ci permetta di memorizzare un numero arbitrario di informazioni all'interno di uno spazio di dimensione fissa.

Siccome la maggior parte dei file ha una dimensione limitata, per i primi n blocchi abbiamo un array che ci porta direttamente ai blocchi dati. Un elemento di questo array sarà utilizzato per puntare ad un altro blocco che ha la sola funzione di contenere puntatori ad altri blocchi dati. Questa operazione viene ripetuta a più livelli, creando così diversi livelli di indirezione per l'accesso ai blocchi dati.



Questa struttura ha un livello di efficienza che dipende dalla dimensione del file: più il file sarà grande, meno questa struttura dati sarà efficiente. Siccome la maggior parte dei file è piccola, viene ottimizzato il caso più comune.

Directory, link e cancellazione

Le directory sono file gestiti dal sistema che contengono associazioni nome-inode #

In questo modo il sistema non deve distinguere i file che contengono dati dalle directory, perché queste ultime sono dei file che contengono una serie di queste associazioni. Tra queste associazioni sono presenti anche le associazioni con i nomi . (directory corrente) e .. (directory parent).

Queste associazioni vengono chiamate hard link.

Tramite il comando ls possiamo vedere il numero di hard link associati ad un inode. Si tratta del numero dopo i permessi del file.

```
martina@martina-xps-13:~/pippo$ ls -l
totale 4
-rw-r--r-- 1 martina martina 0 gen 11 14:02 paperino
drwxr-xr-x 2 martina martina 4096 gen 11 14:02 pluto
```

Ciò significa che il file che noi identifichiamo come paperino ha un solo nome all'interno del file system che lo identifica. La directory pluto ha invece due hard link che identificano quel file: uno è il nome pluto, l'altro è il nome . con cui viene identificata al suo interno.

Possiamo aggiungere dei nomi creando degli hard link tramite il comando ln

```
martina@martina-xps-13:~/pippo$ ln paperino clarabella
martina@martina-xps-13:~/pippo$ ls -l
totale 4
-rw-r--r-- 2 martina martina 0 gen 11 14:02 clarabella
-rw-r--r-- 2 martina martina 0 gen 11 14:02 paperino
drwxr-xr-x 2 martina martina 4096 gen 11 14:02 pluto
```

I nomi clarabella e paperino corrispondono allo stesso file, con lo stesso inode (visible tramite comando stat). Quando usiamo il comando rm (ad esempio rm clarabella) cancelliamo un hard link (tramite la system call unlink(2)). Viene quindi rimosso solamente il nome, mentre il file (e il relativo inode) continuano ad esistere con gli altri nomi. Se cancello l'unico nome che fa riferimento ad un file, se non ci sono file descriptor aperti, cancello il file e il relativo inode che potrà essere riutilizzato.

Tramite il comando ln -s possiamo creare dei link simbolici: questi sono file (viene allocato un nuovo inode), il cui contenuto corrisponde ad un percorso (non necessariamente esistente).

```
martina@martina-xps-13:~/pippo$ ln -s paperino topolino
martina@martina-xps-13:~/pippo$ ls -l
totale 4
-rw-r--r-- 2 martina martina 0 gen 11 14:02 clarabella
-rw-r--r-- 2 martina martina 0 gen 11 14:02 paperino
drwxr-xr-x 2 martina martina 4096 gen 11 14:02 pluto
lrwxrwxrwx 1 martina martina 8 gen 11 14:15 topolino -> paperino
```

Nel momento in cui accediamo al contenuto del file topolino, veniamo reindirizzati al contenuto del file paperino. Se eliminiamo paperino, topolino continua ad esistere, e conterrà un percorso non esistente.

Limitazioni

Possiamo creare dei link simbolici verso file system diversi, ma possiamo creare degli hard link solo all'interno dello stesso file system. Questo perché il numero di inode ha senso solo all'interno di un file system. Su un altro file system lo stesso numero di inode potrebbe essere usato per qualcosa'altro.

Directory e permessi

Per le directory

- r: posso leggere? Ovvero, listare i file contenuti
- w: posso modificarne il contenuto? Creare/cancellare/rinominare/. . . nomi contenuti
- x: posso "entrare"/accedere?

Ciò significa che per creare/eliminare nomi da una directory d, dovete avere il permesso di scrittura su d. Ad esempio possiamo cancellare file read-only di root se la directory è nostra.

Risoluzione dei percorsi – path_resolution(7)

1. Nel PCB ci sono inode di root r e directory corrente c
 - a. il percorso inizia con "/"? Allora si tratta di un percorso assoluto, d = r
 - b. altrimenti si tratta di un percorso relativo, d = c
2. Per ogni componente (separata da /) non-finale, diciamo n
 - a. *Si hanno i permessi di ricerca in d?*
Questo significa verificare se è root, oppure verificare i permessi legati all'effective UID riguardo a quella directory. Se non si hanno i permessi di ricerca → errno=EACCESS
 - b. *Si cerca n in d*
 - i. non si trova → ENOENT
 - ii. altrimenti si recupera inode corrispondente i
 - c. *i è una directory?*
Se sì, si riparte da lì: d = i
 - d. *i è un link-simbolico?* Si risolve a partire da d
 - i. Il risultato non è una directory? → ENOTDIR
 - ii. Se lo è, d', si continua da lì: d = d' — un contatore evita loop infiniti
 - e. ENOTDIR
3. Per la componente finale non si pretende che sia una directory.
4. . e .. hanno significato speciale
 - a. Anche se il sottostante FS non li memorizza esplicitamente
 - b. Ma non si può salire sopra la radice: ../../

Integrità

Consistenza del file system

Cosa succede a livello di file system quando viene creato un file?

- Bisogna allocare un inode, andando a scrivere un bit all'interno della bitmap per indicare che quel particolare inode è stato utilizzato.
- Bisogna aggiungere l'associazione nome-numero inode nella directory corrente, modificando un blocco dati della directory corrente

Se ci sono varie parti della struttura dati da aggiornare (e.g. blocco dati, bitmap e puntatori ai blocchi), cosa succede se manca la corrente a metà? Potremmo aver aggiunto l'associazione tra nome e inode senza averlo marcato, oppure potremmo aver marcato l'inode come occupato senza effettuare l'associazione con un nome.

C'è un ordine migliore di altri?

Sicuramente avere delle associazioni all'interno di una directory senza avere riservato l'inode è peggio: nel momento in cui un altro processo creerà un file andando ad utilizzare quell'inode, avremo due file completamente diversi che andranno ad usare lo stesso inode. È come avere degli hard-link, ma il numero di hard-link rimane ad 1.

Al contrario, avere un inode occupato quando non lo è, elimina la possibilità di avere quel file, ma non genera conflitti.

Fsck

Per controllare l'integrità di un file system (smontato, non in uso in quel momento) fsck(1):

- controlla che il superblocco sia "ragionevole"
- consistenza tra bitmap blocchi liberi e puntatori ai file
 - più inode puntano a uno stesso blocco? fix: si può duplicare il blocco, dando a ognuno la sua copia
 - un blocco puntato risulta libero? Fix: si indica come occupato sulla bitmap
 - un blocco non puntato risulta usato? Fix: si indica come libero sulla bitmap
- stato degli inode (e.g. tipo valido)
- link count
 - scandendo ogni directory recupero il numero di link per ogni inode e controllo che corrisponda con il link count. Nel caso lo aggiusto
 - recupero file senza nome
- puntatori fuori dal range dei blocchi
- directory
 - ognuna ha il suo . e ..?
 - qualcuna è collegata più di una volta nell'albero?

Journaling

Abbastanza ovviamente, il controllo di integrità è molto lento, e cresce sempre di più al crescere della dimensione dei dischi. Ha quindi un costo elevato, ma vorremmo effettuare dei controlli di integrità ad ogni avvio del sistema. C'è una possibile ottimizzazione: effettuare un controllo di integrità solo quando il sistema è andato in crash. Un file system smontato "in modo pulito" non viene controllato a ogni mount.

Come viene realizzata? Viene inserito all'interno del superblocco un indicatore che indica che il file system è "sporco" (dirty). Se il file system non viene smontato correttamente, questo indicatore rimane all'interno del superblocco e la volta successiva in cui si prova a montare questo disco verrà effettuato un controllo di integrità.

Approccio moderno è quello che prevede l'uso del Journaling (AKA write-ahead logging). Questo meccanismo permette di rendere atomiche le (sequenze di) operazioni.

Per fare questo, prima di fare le modifiche, vengono segnate in un log; se vengono correttamente eseguite vengono cancellate dal log, altrimenti se c'è un crash rimangono salvate nel log e vengono riapplicate al mount.

Per approfondire potete vedere: <http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf> e capitoli seguenti

Processi (API)

martedì 10 novembre 2020 18:48

Le API sono le system call che hanno a che fare con i processi.

(P)PID

```
pid_t getpid(void)  
pid_t getppid(void)
```

Le system call getpid e getppid restituiscono rispettivamente il process identifier del processo stesso e del processo parent.

I processi formano un albero: ogni processo ha un solo parent. La radice dell'albero è data dal processo con pid=1 chiamato init (nei sistemi moderni systemd). Si tratta dell'unico processo creato in automatico dal sistema, nel momento in cui viene avviato. Tutti gli altri processi vengono creati tramite system call.

La gerarchia dei processi può essere visualizzata tramite il comando pstree.

Attenzione:

I PID identificano i processi in un particolare momento: il PID è unico ma può essere associato in momenti diversi a processi diversi, poiché i PID vengono riutilizzati. Quando un processo termina il suo PID potrà essere riassegnato prima o poi ad un altro processo, in base a fattori di carico del sistema (più processi vengono creati più velocemente vengono riassegnati i PID) e numero di bit per esprimere i PID (più bit -> più PID utilizzabili -> meno riutilizzo).

Le informazioni sui processi possono essere viste utilizzando lo pseudo-filesystem /proc. Non corrisponde a dei file su disco ma rappresenta un modo per interrogare il kernel e ottenere informazioni sui processi. Su /proc troviamo una directory per ogni processo.

Contiene vari file, di cui la maggior parte in sola lettura. Tra questi troviamo maps, che indica la struttura dello spazio di indirizzamento del processo

Vedere proc(5).

Directory di lavoro

Ogni processo ha una directory di lavoro, utilizzata per tutti i percorsi relativi (che non iniziano per /) e una directory root, identificata da / (i percorsi assoluti sono quelli che iniziano con /).

Non è detto che la root del file system sia la stessa della root del processo: processi diversi potrebbero avere root diverse.

La system call getcwd(2) restituisce la directory di lavoro corrente, chdir(2) e fchdir(2) la modificano.

API per la gestione dei processi

Le principali system call per la gestione dei processi sono:

- **fork** crea un nuovo processo
- **_exit** termina il processo chiamante
- **wait** aspetta la terminazione del processo figlio
- **execve** esegue un nuovo programma nel processo chiamante, sostituendo l'intero spazio di indirizzamento

La chiamata di execve spesso segue la chiamata di fork. Ad esempio, quando dalla bash viene lanciato il comando ls, viene fatta una fork. Viene creato un nuovo processo, all'interno del quale verrà fatto execve di /bin/ls. Nel frattempo nel processo chiamante viene fatta una wait, per aspettare che ls termini. Nel momento in cui ls (e quindi wait) termina, viene stampato il prompt.

Attenzione: è importantissimo che la chiamata di execve di /bin/ls avvenga solamente dopo la fork, altrimenti si tratterebbe di un "suicidio" della bash, poiché verrebbe sostituito il suo spazio di indirizzamento.

Fork

```
pid_t fork(void)
```

La system call fork clona un processo: crea una copia del processo che la invoca.

Per questo motivo init è speciale: non viene creato attraverso una fork, ma viene creato dal kernel in modo speciale.

Il nuovo processo creato dalla fork viene detto "figlio" del processo chiamante. Questo poiché il PPID del processo figlio coincide con il PID del processo chiamante.

La copia del processo chiamante è quasi identica: cambiano PID e PPID ma i file descriptor (compresi i flag) vengono duplicati (ciò significa che possono essere utilizzati dal processo figlio per leggere e scrivere lo stesso file) e l'intero spazio di indirizzamento viene copiato.

Dal punto di vista logico tutte le pagine di indirizzamento del processo chiamante dovrebbero essere duplicate (non efficiente), ma nei sistemi moderni viene fatta una pagina logica grazie al concetto di copy-on-write.

Ciò significa che non viene effettuata una copia delle pagine dello spazio di indirizzamento in RAM (soprattutto inutile se dopo viene fatta una execve) ma il processo figlio condivide le stesse pagine in memoria fisica del processo chiamante, che vengono marcate come readonly. Questo significa che i processi non possono modificare contemporaneamente eventuali dati presenti nelle pagine.

Inoltre, il kernel tiene traccia del numero di processi che hanno contemporaneamente accesso ad una pagina. Nel momento in cui resta solo un processo, può ripristinare i permessi originali.

Nel momento in cui un processo che condivide una pagina fisica (con permesso readonly) prova a modificare dei dati contenuti al suo interno, viene generata una trap. Il kernel, vedendo che è stata generata una trap a seguito di una richiesta di modifica di un'area di memoria condivisa, genera una copia della pagina, a cui fa puntare il processo che ha richiesto la modifica, consentendogli di effettuarla.

Questo significa che processo chiamante e processo figlio non condivideranno più quella particolare pagina dello spazio di indirizzamento, che è stata copiata a seguito di una richiesta di scrittura (copy-on-write).

Una volta creata la copia la system call fork (chiamata una volta) ritorna due volte (su processo padre e processo figlio). Al padre restituisce il PID del figlio, al figlio restituisce 0. Possono essere quindi effettuate azioni diverse (su padre e figlio) in base al valore restituito dalla system call.

Fork in gdb

Normalmente gdb continua a seguire solo uno dei due processi: di default non è possibile seguire sia padre che figlio. È possibile scegliere quale seguire con il comando:

```
set follow-fork-mode [child|parent]
```

Nel caso in cui si vogliano seguire entrambi (sconsigliato perché si fa casino) è necessario usare il comando

```
► set detach-on-fork off
```

per vedere i processi collegati a gdb: info inferiors
per selezionare quello corrente: inferior id

Esempio – double fork

```
#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);
}
```

In questo programma il processo parent effettua una fork(), creando un processo figlio. Dopodiché entrambi effettuano nuovamente una fork(), creando a loro volta altri due processi (in totale abbiamo 4 processi). Al termine viene stampato "pippo\n" una volta per ogni processo.

Totale "pippo\n" = 4

Exit

```
void exit(int status) //libreria C
void _exit(int status) //syscall (in Linux, exit_group(2))
```

Esistono due varianti della exit:

- La prima, funzione della libreria C, prima di uscire:
 - Chiama le funzioni registrate con atexit(3) e on_exit(3)
 - Svuota i buffer di I/O
 - Elimina i file temporanei creati con tmpfile(3)
- Entrambe le funzioni:
 - Chiudono/rilasciano le risorse del processo (file descriptor, memory mapping, ...)
 - Fanno terminare il processo con exit status uguale a (status & 0xff): ciò significa che viene preso solo il byte meno significativo dello status, che può essere recuperato dalla shell con \$?
 - Eventuali figli di questi processi vengono adottati, storicamente da init, con PID=1

Restituire n dal main corrisponde ad exit(n). Lo standard C definisce le costanti EXIT_SUCCESS (=0) ed EXIT_FAILURE (=1).

Wait

```
pid_t wait(int *wstatus)
```

La system call wait permette di aspettare che succeda qualcosa in un processo figlio. Attende un cambio di stato, che può corrispondere alla terminazione del processo figlio oppure allo stop/ripartenza tramite segnali.

La wait di default aspetta un figlio qualsiasi, e restituisce il pid di quello che ha aspettato. Nel caso in cui si voglia avere più controllo su quale figlio aspettare, vedere waitpid(2).

La wait prende in input anche un puntatore ad intero, dove scrive cosa è successo al processo.

Ha senso guardare cosa c'è nell'area di memoria puntata dal puntatore solo se la wait è andata a buon fine.

All'interno dell'area puntata dal puntatore ci può essere scritto:

- WIFEXITED (se il figlio è terminato con exit(). Possiamo recuperare l'exit status con WEXITSTATUS)
- WIFSIGNALED (se il figlio è terminato con un segnale, che possiamo recuperare con WTERMSIG)

Zombie

Un processo terminato, non aspettato dal padre, è uno zombie.

ATTENZIONE:

- Orfano: processo in esecuzione il cui parent è morto
- Zombie: processo terminato che non viene aspettato dal parent. Diventa zombie finché il parent non lo aspetta

Il sistema può rilasciare alcune risorse ma non può dimenticarsi del processo: siccome il parent è ancora in vita, prima o poi potrebbe aspettare quel figlio leggendo il PID e l'exit-status del figlio, che devono continuare ad essere memorizzati nel sistema.

Per questo motivo i processi orfani vengono adottati da init: senza qualcuno che li aspetta il sistema si riempirebbe di processi zombie.

Allo stesso modo, se il padre termina senza aspettare il figlio zombie, quest'ultimo viene adottato da init che lo aspetta.

Quando un processo termina, viene inviato al padre il segnale SIGCHLD. Di default questo segnale viene ignorato, ma può essere catturato per effettuare la wait dei figli.

Siccome il segnale (non realtime) non si accoda (il sistema non tiene conto del numero di segnali che arriva), potrebbero esserci più figli terminati per un singolo segnale. Per questo è buona norma effettuare un ciclo while per la wait.

Exec

La system call execve permette di eseguire un nuovo programma all'interno di un processo (creato per esempio con una fork).

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

Prende come argomenti il percorso del programma da eseguire, gli argomenti da passare al programma e le variabili d'ambiente da passare.

Esistono poi una serie di funzioni di libreria che usando execve e sono più o meno comode a seconda di come vogliamo passare i parametri:

- Dove c'è v gli argomenti vengono passati come vettore
- Dove c'è l gli argomenti vengono passati come lista
- Dove c'è p viene ricercato nel path un nome che corrisponde al file che stiamo chiedendo
- Dove c'è e viene passato l'ambiente

L'exec sostituisce lo spazio di indirizzamento del processo: prepara il nuovo spazio di indirizzamento e se questa operazione va a buon fine butta via quello vecchio e lo sostituisce con quello nuovo.

Lascia invece invariati PID, PPID e file descriptor (a meno che su un file descriptor non sia abilitato il flag FD_CLOEXEC)

Nuovo spazio di indirizzamento

Quando eseguiamo un programma il sistema (kernel+linker) prepara uno spazio di indirizzamento dove inserisce:

- codice, mappato r-x (.text)
- dati a sola lettura, mappati r-- (.rodata)
- dati, mappati rw- (.data, .bss e heap)
- stack, mappato rw-
 - in fondo allo stack (quindi, a indirizzi più grandi) il kernel copia:
 - le variabili d'ambiente
 - gli argomenti della riga di comando
 - (altra "roba")
- (il kernel, "invisibile")

In caso di linking dinamico il kernel fa la stessa cosa, ricorsivamente, per il codice e i dati delle librerie necessarie. Quando viene creato un thread, viene allocato un nuovo stack.

Nei sistemi moderni codice e dati vengono portati in RAM on demand.

Valore di ritorno

La exec, se va a buon fine, sostituisce completamente il codice che l'ha chiamata con il programma che vogliamo eseguire, e non ritorna nulla. Se la exec restituisce un valore, qualcosa è andato storto.

Se l'eseguibile ha i bit set-user-ID/set-group-ID abilitato, succedono "cose" (ne parleremo in un altro momento).

Programmi binari e processi

venerdì 30 ottobre 2020 12:33

Dopo la scrittura di un programma in un linguaggio ad alto livello, questo viene tradotto dal compilatore in un programma binario che è possibile mandare in esecuzione.

Cosa sono questi file binari? Che cosa contengono? Cosa deve fare il sistema per poterli mandare in esecuzione?

Innanzitutto, siamo abituati a chiamare gcc compilatore. In realtà si tratta di un programma che pilota l'intero processo di compilazione, lanciando preprocessore, compilatore, assemblatore e linker.

- **Compilatori e assemblatori producono i file oggetto/rilocabili (*.o)**

L'input per i compilatori sono i file c e i file .h. Il preprocessore espande le macro, fa gli include e produce un unico file c, su cui il compilatore produce un file .s (file assembler).

Sulla base del file .s l'assemblatore produce un file .o (file oggetto). Questo passo intermedio serve per poter scrivere un unico assemblatore per tutti i linguaggi di programmazione, che prima di essere tradotti in file oggetto vengono tradotti in un formato comune.

- **Id, il link editor (AKA linker statico), mette assieme file-oggetto/librerie, eseguendo rilocazione e risoluzione dei simboli**

Il link editor si occupa di unire tutti i file .o con le librerie (per esempio la libreria c) e crea un unico eseguibile.

Il linking può essere statico o dinamico. Di default è dinamico.

Linking statico

Quando il linking è statico Id unisce tutto insieme e crea dei programmi auto-contenuti. Ciò significa che se il nostro programma chiama una funzione della libreria C, Id prende il codice della funzione di libreria e lo unisce al nostro codice. I pezzi di libreria vengono quindi copiati all'interno del programma.

In questo modo gli eseguibili funzionano su tutte le macchine, ma tendono ad essere molto pesanti in termini di occupazione di spazio su disco e di occupazione della memoria.

Linking dinamico

Quando il linking è dinamico, Id prepara l'eseguibile "annotando" le dipendenze esterne alle librerie. Tra le dipendenze vi è anche il suo interprete, il linker dinamico Id.so

Il linker dinamico Id.so mette assieme i pezzi mancanti a tempo di esecuzione.

Questa tipologia di linking viene utilizzata di default sui sistemi moderni.

Permette di risparmiare spazio, sia su disco sia in memoria fisica.

Questo perché, supponiamo venga utilizzata la libreria C. Questa viene copiata una sola volta su disco e non in ogni singolo eseguibile. Inoltre, se tutti i programmi che usano la libreria fanno riferimento allo stesso file, possiamo caricare in memoria il codice della libreria una sola volta, ma mappare i page frame in tante page table diverse, una per processo.

Processi diversi possono avere pagine che fanno riferimento agli stessi frame in memoria fisica, poiché i frame sono accessibili in sola lettura. Questo non genera conflitti e permette il risparmio di memoria.

Questa modalità facilita anche l'aggiornamento delle librerie: per aggiornare una libreria non devo andare a modificare tutti gli eseguibili che la utilizzano, ma è sufficiente andare ad aggiornare l'unica versione presente sul disco. Automaticamente tutti i programmi che utilizzano quella libreria utilizzeranno la versione aggiornata.

Tuttavia, a differenza del linking statico, attraverso il linking dinamico può essere problematico portare l'eseguibile da una macchina a un'altra, a causa di possibili differenze di librerie presenti sui sistemi. Ci sono anche lievi svantaggi in termini di efficienza, perché bisogna andare a linkare a runtime le funzioni di libreria utilizzate, al giorno d'oggi si tratta di ritardi totalmente irrilevanti.

Cosa contiene un eseguibile?

Gli eseguibili, così come file rilocabili e librerie, possono contenere:

- Codice macchina: corrisponde alla traduzione del codice scritto ad alto livello. Viene inserito in una sezione chiamata .text
- Dati e dati a sola lettura: quando dichiariamo delle variabili e le inizializziamo, il valore di inizializzazione deve essere portato nell'eseguibile. Se invece dichiariamo una variabile senza inizializzarla nell'eseguibile viene riportata solo la sua dimensione. Queste informazioni vengono riportate nelle sezioni .data e .rodata
- Metadati
 - architettura (Intel/ARM/..., 32/64 bits, little/big endian, ...)
 - entry-point; no, non è il main, ma è la libreria C
 - quanto spazio riservare per variabili globali non inizializzate → .bss
 - ...

In un eseguibile le sezioni vengono raggruppate in segmenti, i “pezzi” che vengono mappati in memoria (tramite paginazione) dal s.o./linker-dinamico per l'esecuzione.

Attenzione: questi segmenti NON necessariamente corrispondono ai segmenti della CPU/MMU. Infatti, non è così in Linux, Windows e Xv6.

Executable and Linkable Format - ELF

Tutte queste informazioni devono essere contenute in un unico file. Ci deve quindi essere una specifica che indica come scrivere queste informazioni nel file. Questa specifica si chiama ELF (Executable and Linkable Fromat).

Il formato ELF può essere utilizzato per gli eseguibili, ma anche per i file oggetto (.o), per le librerie dinamiche (.so) e altre tipologie di file.

Un file ELF inizia con un elf header che identifica il file. L'elf header può puntare a delle tabelle contenute nel file. A seconda della tipologia di file che stiamo analizzando (eseguibile, file oggetto ecc) ci interessano solo le sezioni (descritte da un section header) o solo i segmenti (descritti dal program header).

Nei file eseguibili ci interessano solo i segmenti. Nel program header sono indicate le posizioni dei segmenti, all'interno dei quali troviamo le sezioni. Quando mandiamo in esecuzione il programma il sistema operativo/linker va a vedere dove sono i segmenti indicati dalla program header table. Queste informazioni ci servono a costruire lo spazio di indirizzamento del processo.

Spazio di indirizzamento di un processo

Lo spazio di indirizzamento è l'astrazione della memoria fisica.

I processi utilizzano indirizzi logici/virtuali che vengono tradotti in indirizzi fisici tramite la MMU (Memory Management Unit). Questa traduzione può avvenire tramite segmentazione, paginazione oppure tramite delle combinazioni dei due metodi.

È possibile vedere come un processo utilizza indirizzi virtuali lanciando 3 volte lo stesso programma contemporaneamente, che stampa l'indirizzo di una stessa variabile. In tutti e 3 i processi l'indirizzo della variabile sarà lo stesso.

Cosa serve per far girare un programma

Per far girare un programma non basta mappare codice e dati.

Supponiamo di voler mandare in esecuzione un programma: attraverso la chiamata alla system call execve il programma viene mandato in esecuzione, viene riservato uno spazio di indirizzamento nuovo dove vengono inseriti codice e dati. Questo non basta, è necessario avere anche:

- **Stack**: struttura dati runtime necessaria per salvare variabili locali, temporanee, parametri, indirizzi di ritorno ecc. Viene utilizzato in modo automatico dalla CPU ogni volta che effettuiamo una chiamata di funzione (anche system call). Lo stack viene anche utilizzato per il passaggio di parametri alle funzioni tramite il registro stack pointer (\$esp).
- Lo stack cresce verso indirizzi decrescenti: l'operazione di push, utilizzata per inserire un valore nello stack, decrementa lo stack pointer di 4 byte. La push degli argomenti di una funzione nello stack viene fatta da destra verso sinistra: in questo modo il primo argomento si troverà sempre a 4 byte dallo stack pointer.

Se, quando è in esecuzione un processo, ne viene mandato in esecuzione un altro, viene creato un altro stack. Questo perché ogni processo vede il proprio spazio di indirizzamento e di default i processi non condividono memoria. All'interno dei processi possiamo creare più flussi di esecuzione, detti thread. I thread condividono la memoria e lo spazio di indirizzamento, ma ognuno di essi ha il proprio stack.

- **Heap**: memoria dinamica, ottenuta a livello di codice utente con malloc() e free(). Non conosciamo a priori la quantità di memoria dinamica che verrà utilizzata, e per questo è necessaria un'altra regione. Storicamente lo heap viene messo sotto allo stack, e viene fatto crescere verso l'alto (indirizzi crescenti). Questo funziona finché stack e heap non si incontrano.
- **Kernel**: all'interno dello spazio di indirizzamento di un processo deve esserci anche il kernel, che viene mappato ad indirizzi alti sopra allo stack. Il kernel viene mappato all'interno dello spazio di indirizzamento (ossia nella tabella delle pagine del processo) ma queste pagine sono marcate come accessibili solo dalla modalità kernel (e non dalla modalità utente).

Logicamente ogni processo ha il suo spazio di indirizzamento ed è completamente isolato da tutti gli altri processi. Tuttavia ci sono delle ottimizzazioni possibili che comprendono la condivisione del codice. Ad esempio, per lanciare 2 volte lo stesso programma, non ha senso portare in memoria fisica 2 volte lo stesso codice, poiché normalmente il codice non viene modificato. Questo significa che le stesse pagine di memoria fisica vengono mappate in diversi spazi di indirizzamento relativi a processi diversi (anche ad indirizzi logici diversi).

Inoltre, quando viene mandato in esecuzione un programma, non è necessario che venga immediatamente caricato tutto in RAM. Se una pagina del processo non è presente in memoria quando viene mandata in esecuzione, viene generata una trap e il kernel provvede al caricamento. Nel momento in cui una pagina non serve più può essere sostituita.

Redirezione dell'I/O

domenica 15 novembre 2020 15:06

Come viene interpretata la redirezione di input e output dalla shell?
La prima cosa da considerare è la system call dup

Dup

```
int dup(int oldfd);
```

System call che crea un nuovo file descriptor equivalente ad uno che abbiamo già. Entrambi i file descriptor "punteranno"* allo stesso file aperto

* I file descriptor sono in sé degli interi, indici di una struttura dati del kernel. Per ogni processo che c'è nel sistema il kernel terrà una struttura dati chiamata Process Control Block (struct del C che contiene le informazioni legate ad ogni processo). All'interno della struct è presente un array legato ai file descriptor. Questa struttura dati, a seconda di come è implementato il sistema, può essere un array di puntatori, di interi, di struct...

Dal punto di vista logico il file descriptor ci collega ad un'altra struttura dati che corrisponde al file aperto nel sistema. Questa conterrà informazioni quali il nome del file, l'offset utilizzato per leggere/scrivere, i flag e un inode (struttura dati che rappresenta un file nel file system).

Il nuovo file descriptor restituito dalla dup è il più piccolo disponibile (come nel caso della open). Quando creiamo il file descriptor equivalente non vengono copiati i flag del file descriptor (da non confondere con i flag del file). Ciò significa che il flag del vecchio file descriptor FD_CLOEXEC (se attivo) non sarà quindi più attivo sul nuovo file descriptor: in caso di exec il nuovo file descriptor rimarrà, a differenza del vecchio, aperto.

Esiste una seconda versione della system call dup

```
int dup2(int oldfd, int newfd);
```

La dup2 prende due file descriptor: oltre a quello vecchio da "duplicare" prende anche il nuovo file descriptor che vogliamo ottenere. Non restituisce quindi il più piccolo disponibile ma, se va a buon fine, restituisce quello che le abbiamo passato come secondo argomento.

Se per caso il newfd era già aperto ed associato ad un file, lo chiude e lo associa allo stesso di oldfd (se oldfd è valido). Se oldfd==newfd non fa nulla.

Relazione tra file descriptor e file aperti (POSIX)

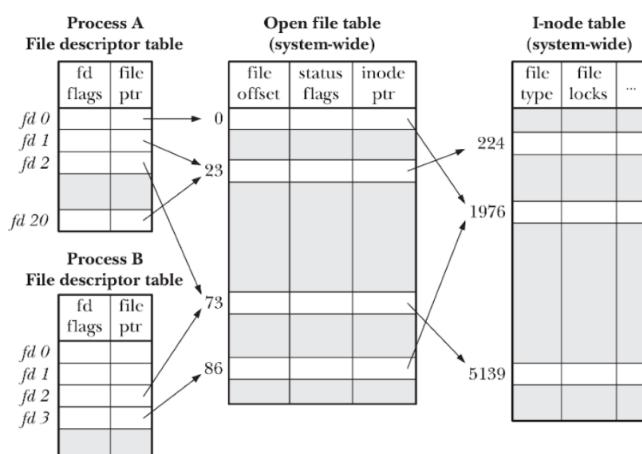


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Ogni processo ha la propria tabella dei file descriptor, contenuta all'interno del Process Control Block. Il file descriptor è un indice all'interno della tabella, che contiene i flag relativi ai vari file descriptor e i puntatori che mettono in collegamento il file descriptor e il file aperto.

Nel caso in cui venga effettuata una dup avremo più file descriptor che puntano allo stesso file aperto.

Nella tabella dei file aperti per ogni file è presente l'offset (dove siamo arrivati a leggere/scrivere), i flag di stato del file e un qualcosa che collega il file aperto con un inode (struttura dati che rappresenta un file nel file system).

Nel caso in cui uno stesso file venga aperto in due momenti diversi da processi diversi (non attraverso fork() o dup()) saranno presenti due campi diversi nella tabella dei file aperti. Questo perché, per esempio, potrebbero esserci dei permessi diversi (apertura in sola lettura/ apertura in scrittura e lettura...). Ciò comporta anche offset diversi. La inode rimarrà invece la stessa.

Possiamo avere situazioni in cui due file descriptor diversi dello stesso processo puntano allo stesso file aperto (ottenuto tramite una dup), ma anche una situazione in cui gli stessi file descriptor di processi diversi puntano allo stesso file aperto. Questo succede quando viene effettuata una fork(): tutti i file descriptor del processo padre vengono mantenuti nel processo figlio.

Infine, possiamo avere file descriptor diversi di processi diversi che puntano allo stesso file (stessa entry nella tabella dei file aperti). Questo succede grazie ad una fork() + dup().

Cosa c'entra con la redirezione dell'I/O

Sappiamo che quando facciamo ls > pippo, lo standard output di ls anziché essere stampato a terminale, viene scritto sul file pippo.

Potremmo pensare che sia il comando ls che, quando si trova sugli argomenti della linea di comando un > vede che segue un nome di file, lo apre, ci scrive dentro... Questo non è efficiente, poiché ogni singolo comando dovrebbe implementare questo meccanismo.

Un modo più semplice per risolvere questa cosa è far credere al comando ls che il suo output sia finito su standard output. ls continuerà quindi ad utilizzare il file descriptor 1. Se cambiamo il significato del file descriptor 1 automaticamente cambiamo la posizione in cui i comandi andranno a scrivere.

Cosa succede quando lanciamo il comando ls:

1. La bash effettua una fork: viene creato un processo figlio (con gli stessi file descriptor 0, 1, 2) del parent;
2. Sul processo figlio viene effettuata una exec() mandando in esecuzione il comando ls (vengono lasciati invariati i file descriptor)
3. Il processo parent (bash) effettua una wait() per attendere il termine dell'esecuzione di ls sul processo figlio

Cosa succede quando lanciamo il comando ls > pippo:

1. La bash effettua una fork(): viene creato un processo figlio (con gli stessi file descriptor 0, 1, 2) del parent;
2. Sul processo figlio viene effettuata una open() del file pippo: al file descriptor 3 corrisponderà il file pippo aperto in scrittura;
3. In seguito alla open() sul file descriptor 3, viene effettuata una dup2 tra il file descriptor 1 e il file descriptor 3. In questo modo lo standard output punta al file pippo;
4. In seguito alla dup() il file descriptor 3 non viene più utilizzato e può quindi essere chiuso tramite una close();
5. Dopo di che sul processo figlio viene effettuata una exec() mandando in esecuzione il comando ls (vengono lasciati invariati i file descriptor)
6. Il processo parent (bash) effettua una wait() per attendere il termine dell'esecuzione di ls sul processo figlio

In alternativa alla dup() potremmo chiudere il file descriptor 1, ed effettuare la open del file pippo, che restituisce il file descriptor più piccolo (1). Non si tratta di una mossa molto intelligente perché nel caso in cui la open fallisca, non abbiamo modo di ripristinare il file descriptor 1.

Pipe (anonime)

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

La pipe è un canale unidirezionale di byte (anonimo). È simile ad un socket TCP ma a differenza del socket è unidirezionale (c'è un file descriptor in cui si può scrivere, dove solitamente agisce un processo, e un file descriptor in cui si può leggere, dove agisce l'altro processo).

Come il socket TCP anche la pipe è un canale byte-stream: non importa il numero di scritture che vengono effettuate, ma solo il ciò che viene scritto. Non c'è un concetto di messaggio o di corrispondenza tra numero di letture/scritture.

Di per sé la pipe è un buffer all'interno del kernel. Alla system call pipe viene passato un array di 2 interi. Se la syscall va a buon fine viene allocato nel kernel questo buffer, e vengono restituiti 2 file descriptor (uno per scrivere, l'altro per leggere dal buffer). pipefd[0] corrisponde alla lettura, pipefd[1] corrisponde alla scrittura.

Quando il buffer è pieno, le write vengono messe in attesa (o falliscono se flag O_NONBLOCK è abilitato). Simmetricamente se la pipe è vuota e qualcuno prova ad effettuare una read(), viene messo in attesa.

Il kernel tiene traccia di tutti i file descriptor aperti in lettura e in scrittura sulla pipe. All'inizio sono 1 e 1, ma a seguito di fork() e dup() potrebbero moltiplicarsi.

Quando tutti i fd di scrittura sono chiusi e si prova ad effettuare una read() di una pipe vuota, la read restituisce EOF (valore 0), perché non c'è niente da leggere e non ci sarà mai.

Viceversa, se vengono chiusi tutti i fd di lettura e si prova ad effettuare una write(), questa solleva un segnale SIGPIPE. Questo perché è inutile scriverci qualcosa perché nessuno ci leggerà mai.

Vedere pipe(2) e pipe(7).

Come funzionano le pipe in correlazione al comando | sulla bash

Quando viene dato il comando | sulla bash, viene creata una pipe (un buffer del kernel), su cui viene reindirizzato l'output del comando presente a sinistra di |.

Viceversa l'input del comando a destra deve essere rediretto in lettura del buffer.

Ciò significa che i due comandi (a sinistra e a destra) possono procedere in modo parallelo: man mano che il comando di sinistra produce dell'output, il comando di destra effettua delle letture del buffer processando i dati letti come input.

Cosa succede quando lanciamo il comando A / B

1. La bash() crea la pipe tramite la syscall pipe(). Supponiamo che restituisca fd 3 in pipefd[1] per la scrittura, fd 4 in pipefd[0] per la lettura dalla pipe;
2. La bash effettua due fork(), uno per ogni processo;
3. Vengono chiusi i file descriptor 3 e 4;
4. Viene effettuata la wait() dei processi figli;
5. Processo A
 - a. Viene effettuata una dup2 dei file descriptor 1 e 3. In questo modo allo standard output corrisponde il file descriptor in scrittura della pipe;
 - b. Vengono chiusi i file descriptor 3 e 4;
 - c. Viene effettuata la exec() per mandare in esecuzione il processo A;
6. Processo B
 - a. Viene effettuata una dup2() dei file descriptor 0 e 4. In questo modo allo standard input corrisponde il file descriptor in lettura della pipe;
 - b. Vengono chiusi i file descriptor 3 e 4;
 - c. Viene effettuata la exec() per mandare in esecuzione il processo B;

Introduzione

mercoledì 2 dicembre 2020 09:31

Questa parte si riferisce a <http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf>

Quando abbiamo tanti processi che devono e possono essere eseguiti, deve essere presente un codice per decidere quale di questi mandare in esecuzione. Se c'è una sola CPU con un solo core può esserne mandato in esecuzione solo uno, e la scelta è determinata.

Esistono diversi algoritmi di scheduling per effettuare questa scelta, ognuno dalle prestazioni diverse. Non esiste in generale un algoritmo migliore dell'altro ma dipende dal contesto e dal carico di lavoro.

Nota

I processi che girano su un sistema sono detti workload. Nel contesto dello scheduling i processi sono spesso chiamati job.

Assunzioni

Partiamo con assunzioni irrealistiche:

1. Ciascun job dura lo stesso tempo
2. Tutti i job arrivano allo stesso momento
3. Una volta iniziato un job lo si porta fino in fondo (senza interruzioni)
4. Tutti i job usano solo CPU, no I/O
5. Il tempo di ciascun job è noto a priori

Elimineremo queste assunzioni mano a mano

Metriche

Per misurare la "bontà" di un algoritmo di scheduling abbiamo bisogno di metriche. Inizialmente ci concentreremo sulle performance, ma un altro aspetto da tenere in considerazione è la fairness (quando le risorse sono equamente divise tra i vari processi che devono andare in esecuzione. Se un processo non va mai in esecuzione poiché l'algoritmo non gli rilascia mai l'uso della CPU, si parla di starvation).

Turn-around Time

Si tratta di una misurazione che tiene conto solamente delle performance.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Siccome, per assunzione, tutti i processi arrivano nello stesso momento, possiamo assumere che $T_{arrival} = 0$ e di conseguenza per il momento $T_{turnaround} = T_{completion}$

Misureremo il turnaround medio sommando tutti i tempi di completamento e dividendo per il numero di job.

Response Time

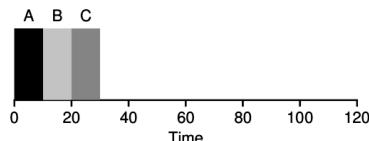
Metrica che misura il tempo che passa da quando arriva un job a quando inizia a fare qualcosa. Per esempio, quando clicchiamo su un'icona ma non riceviamo immediatamente il feedback, significa che il response time è elevato (il sistema non è in grado di mandare in esecuzione quel processo in quel momento, cliccare altre 300 volte è anche peggio).

Nei sistemi interattivi il response time può avere importanza maggiore del turnaround, per dare all'utente la sensazione che effettivamente venga fatto qualcosa.

Tipologie di algoritmi (completare con foto dalle slide)

First In-First Out

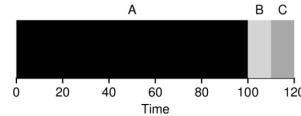
Il primo processo ad entrare è il primo che esce. Supponiamo che i processi arrivino tutti nello stesso momento e abbiano tutti la stessa durata: Il turnaround sarà:



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

Supponiamo ora che i processi non abbiano tutti la stessa durata, e che venga eseguito prima quello più lungo. Generiamo il cosiddetto effetto convoglio.

Se A dura 100, B e C 10

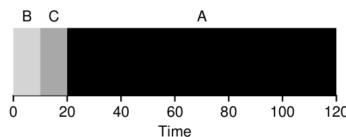


Il tempo medio diventa:

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

Shortest-Job First

Eseguendo invece prima quelli con tempi più corti otteniamo un turnaround ottimale.

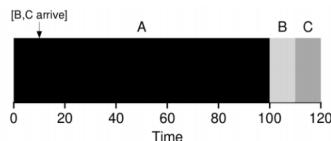


$$T_{\text{turnaround}} = \frac{10 + 20 + 120}{3} = 50$$

Shortest- Time-To-Completion First

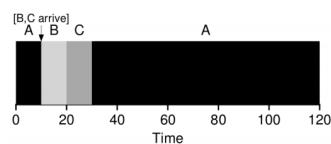
Supponiamo ora che i processi arrivino in momenti diversi e che all'istante iniziale ci sia solo quello più lungo. Otteniamo nuovamente un turnaround molto alto.

Se A arriva all'istante 0, B e C arrivano all'istante 10:



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

Per ovviare a questo problema è necessario introdurre la possibilità di interrompere un processo per eseguirne un altro. In questo modo l'algoritmo si rivela ottimo.



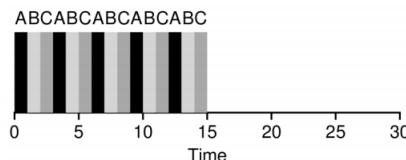
$$T_{\text{turnaround}} = \frac{120 + (20 - 10) + (30 - 10)}{3} = 50$$

In ogni caso questi tempi non sono realistici: non si tiene conto del tempo necessario a deschedulare e rischedulare i processi, non si tiene conto della starvation (questo algoritmo non è fair, se continuano ad arrivare job corti il più lungo non verrà mai eseguito), e nel caso di sistemi interattivi non si tiene conto del response time.

Round-Robin

Algoritmo che tiene conto del response time. Utilizzato per abbassare il tempo di risposta e dare l'illusione che tutto effettivamente proceda.

L'idea è che ogni processo prenda un po' di tempo, e possa eseguire al massimo quella quantità di tempo. Dopotutto viene interrotto e sostituito.



Il tempo medio di risposta per 3 processi del Round-Robin è:

$$T_{\text{response}} = \frac{0 + 1 + 2}{3} = 1$$

Mentre per gli algoritmi Shortest-Job First e Shortest-Time-To-Completion First (ottimali dal punto di vista del turnaround) è:

$$T_{\text{response}} = \frac{0 + 5 + 10}{3} = 5$$

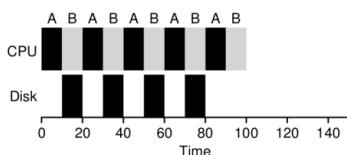
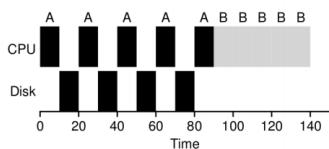
Il tempo di risposta dipende quindi dal numero di job presenti e dalla quantità di tempo che viene data ai processi. Questa quantità di tempo deve essere abbastanza ristretta da permettere dei tempi di risposta brevi. Tuttavia non deve essere troppo breve a causa dell'overhead generato dal context-switch, che rischia di prevalere sugli effettivi tempi di esecuzione dei processi.

Inoltre, l'algoritmo Round-Robin si comporta malissimo nei confronti del turnaround: l'esecuzione viene diluita e il tempo di completamento si allunga. In generale, gli algoritmi fair tendono ad evitare la starvation ma penalizzano i job corti.

Input-output

Non abbiamo ancora considerato i casi in cui i processi facciano input/output.

Mentre il processo attende che vengano effettuate le operazioni di input o di output (ad esempio che il disco legga o scriva dati da file), è bloccato. È inutile lasciargli a disposizione la CPU mentre ci sono altri processi che potrebbero essere eseguiti.



Multi-level Feedback Queue

mercoledì 2 dicembre 2020 10:47

<http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf>

I processi, in momenti diversi della loro esecuzione, possono essere I/O bound (utilizzano prevalentemente meccanismi di input-output, come per esempio per leggere da disco), oppure possono essere CPU bound (utilizzano prevalentemente la CPU).

Tra i processi I/O bound rientrano tutti i programmi interattivi. Nei sistemi che interagiscono con l'utente è molto importante che vi siano dei feedback veloci agli input dell'utente (anche se non si tratta della risposta finale). Per questo motivo, e perché una volta richiesto un input dall'utente i processi si fermano in attesa di ottenerlo, i processi I/O bound devono avere un'alta priorità.

I processi CPU bound invece continuano ad usare la CPU, e anche se vengono eseguiti per più tempo non comportano dei grossi cambiamenti dal punto di vista dell'utente.

Tuttavia non è possibile sapere a priori a quale categoria appartiene un processo: bisognerebbe avere un meccanismo automatico che determina le caratteristiche di un processo e premia i processi I/O bound rispetto a quelli CPU bound, mettendoli in coda per essere eseguiti appena diventano ready.

Questa idea è alla base dell'algoritmo Multi-Level Feedback Queue. Questo algoritmo prevede più livelli di code, e a seconda del comportamento dei processi cerca di prevedere il suo funzionamento posizionandolo all'interno dei livelli diversi di code.

Questo algoritmo cerca di ottimizzare sia il turnaround time, facendo eseguire prima i job corti, sia il response time.

Idea di funzionamento del MLFQ

- Avere diverse code, con diversi livelli di priorità. Quando due processi hanno la stessa priorità tra di loro, viene usato il Round-Robin;
- La priorità di un processo viene determinata dinamicamente a seconda del suo comportamento;
- L'idea è quella di usare la storia di un processo per predirne il comportamento futuro.

Regole

1. Se la priorità di A è maggiore della priorità di B, allora viene mandato in esecuzione A
2. Se A e B hanno la stessa priorità, vengono eseguiti con round-robin.

Supponiamo di avere 4 processi: A e B con priorità 8, C con priorità 4 e D con priorità 1.

Usando solo queste due regole i processi C e D vanno in starvation, perché continuano a girare A e B.

Proviamo allora ad aggiungere le seguenti regole:

3. Un nuovo job entra con priorità massima;
4. Se un job usa tutto il suo quanto di tempo, allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità).

Tuttavia anche con queste regole un processo potrebbe barare: se invece che usare il 100% del suo tempo ne usasse il 99%, resterebbe comunque ad alta priorità e continuerebbe a usare gran parte della CPU. Inoltre non c'è modo di risalire la coda: un processo che inizia l'esecuzione usando molto la CPU viene messo in una coda a bassa priorità. Se in un secondo momento cominciasse a eseguire solo I/O si troverebbe comunque in fondo alla coda di priorità.

Infine con queste regole non viene risolto il problema della starvation: se continuano ad arrivare job nuovi, quelli che sono stati declassati prima non gireranno mai.

Aggiungiamo quindi una nuova regola:

5. Ogni s secondi spostiamo tutti i job alla priorità più alta (resetiamo le priorità).

▶ Questa regola risolve il problema della starvation, e se un processo che era finito a priorità bassa comincia a fare I/O, rimane a priorità alta.

Per risolvere il problema di un processo che cerca di ingannare il meccanismo usando il 99% della CPU, riscriviamo la regola 4:

4. Quando 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.

Conteggiando tutte le quantità di tempo in cui è andato in esecuzione su quel particolare livello di priorità, e non la singola quantità di tempo, riusciamo a risolvere questo problema.

A questo punto con queste regole l'algoritmo funziona: non abbiamo starvation, i job nuovi hanno response time minimo possibile (massimo livello di priorità all'inizio) e riportare tutti i processi a priorità massima dopo un certo periodo di tempo permette di redefinire il comportamento del programma.

Resta il problema di definire il numero di code, quanto vale il tempo t e ogni quanto si resettano i livelli di priorità. A seconda del carico di lavoro, del numero di processi e di molti altri fattori, può avere senso utilizzare valori diversi. Una soluzione può essere quella di inserire dei valori di default e lasciare i parametri modificabili dall'admin.

Proportional Share

mercoledì 2 dicembre 2020 10:48

<http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf>

Si tratta di una categoria di scheduler che, invece di cercare di ottimizzare i tempi di turnaround o response, si basano sull'idea che ogni processo debba avere la sua percentuale di CPU.

Nice

Ogni processo su Unix ha un valore di nice, che indica la sua "bontà". Il valore di nice va da -20 a 19 (di default è 0). Rappresenta una priorità utilizzata dallo scheduler per definire la percentuale di CPU da attribuire ad un processo. Più è basso il valore di nice, più CPU viene assegnata.

I processi normali possono solo aumentare il valore di nice, mentre il superuser può anche diminuirlo. È possibile cambiare il valore di nice tramite la system call nice().

Completely Fair Scheduler – CFS

Questo scheduler è quello attualmente in uso su Linux. Conteggia il tempo usando un concetto di virtual runtime: nel caso più semplice questo tempo è proporzionale al tempo vero, ma varia in base alla priorità dei processi (il virtual runtime di un processo ad alta priorità passa più lentamente rispetto a quello di un processo a più bassa priorità).

Il virtual runtime (vruntime) misura il tempo di utilizzo della CPU.

Quando c'è da mandare in esecuzione un processo il CFS sceglie quello con il vruntime più piccolo (il processo che ha usato meno CPU).

L'algoritmo CFS usa vari parametri, uno dei quali è lo sched_latency, che misura la latenza dello scheduler. Ogni sched_latency si domanda quanti processi ha che sono pronti per essere eseguiti. Dopodiché lo sched_latency viene diviso in n parti, una per ogni processo pronto, e viene mandato in esecuzione il processo con vruntime più basso per un tempo pari a sched_latency/n.

- ▶ Questa operazione viene fatta per un numero di processi contenuto, ossia per uno sched_latency/n non inferiore al parametro min_granularity. Nel caso in cui sia inferiore, viene assegnato il tempo min_granularity ai vari processi. Questo viene fatto per contenere i tempi di context switch, evitando che i tempi di esecuzione diventino più bassi dei tempi necessari a cambiare il processo in esecuzione.

Ogni processo k si prende quindi una fetta (slice) che è data o da min_granularity oppure da una proporzione di sched_latency in base al peso di ogni processo ready (al suo valore di nice).

Il virtual runtime viene invece aggiornato sulla base dell'uso effettivo della CPU da parte di quel processo, scalato per il suo peso (nice).

Efficienza – Alberi rosso/neri

Per mandare in esecuzione un processo tra quelli ready, questo algoritmo sceglie quello con il vruntime più piccolo. Diventa quindi importante utilizzare una struttura dati che permetta di trovarli in tempi molto ridotti.

Per questo motivo si utilizzano gli alberi rosso/neri (alberi binari bilanciati). Questi alberi permettono di trovare il processo con vruntime più piccolo tra n processi e di aggiornare la priorità in un tempo di $\log(n)$.

CFS – Input/Output

Bisogna considerare cosa succede quando un processo viene messo in attesa per fare I/O. Quando torna ready il suo virtual runtime viene aggiornato e viene impostato uguale al minimo tra tutti gli altri processi. La stessa cosa avviene quando arriva un nuovo processo.

Questo permette di non avere squilibri tra i valori che comportano l'uso esclusivo della CPU da parte di un processo (evita starvation). Tuttavia si ripercuote sui processi che fanno I/O frequentemente per breve tempo, perché si "bruciano" i tempi di esecuzione a loro riservati. Tuttavia si tratta di una piccola ripercussione, tanto che l'algoritmo viene comunque definito Completely Fair.

Scripting

giovedì 22 ottobre 2020 12:28

La shell è un linguaggio: oltre a dare i comandi in modo interattivo è possibile dare comandi più complessi formati da cicli, if, else etc. Scrivendo degli script che vengono poi eseguiti dalla bash.

Comandi

I comandi che può prendere la bash sono:

- Semplici: sequenze di comandi separati da spazi
- Pipeline: sequenze di comandi separati da | (lo standard output del primo comando viene dato come input al secondo) e |& (invia sia lo standard error che lo standard output al comando successivo)
- Liste: sequenze di pipeline, separati da:
 - ; separatore simile a ; in C
 - & comandi vengono eseguiti in background in parallelo
- && || And e or possono essere usati per comporre pipelines, in cui i comandi successivi vengono eseguiti in base alla validità del primo (cmd1 && cmd2 – cmd2 viene eseguito solo se viene eseguito cmd1) (cmd1 || cmd2 – cmd2 viene eseguito solo se non viene eseguito cmd1)

Po^{ss}sono essere opzionalmente terminate da ;, &, o newline. Possono essere racchiuse fra (e), o { e }, per applicare un'unica redirezione a tutti i comandi nella lista.

Le parentesi tonde eseguono il comando in un'altra shell, le parentesi graffe nella stessa.

Exit Status

Ogni comando termina con un codice di uscita, l'exit status, che viene memorizzato nella variabile \$?.

Il comando echo &? ci dà l'exit status del comando precedente.

Per convenzione l'exit status 0 corrisponde a "nessun errore", un codice diverso da zero significa che ci sono stati dei problemi.

In un programma C quello che viene restituito dal main corrisponde all'exit status.

Segnali

I comandi possono essere tipicamente interrotti premendo Ctrl+C

Quando un terminale è in modalità canonica, coi settaggi di default, Ctrl+C corrisponde a inviare al comando un segnale SIGINT (2). Tecnicamente, viene inviato al gruppo di tutti i processi in foreground (non in background). Il comportamento di default in seguito alla ricezione del segnale è la terminazione.

Per l'elenco dei segnali vedere signal(7)

Per inviare segnali si può usare il comando kill. Non tutti i segnali comportano la terminazione del programma, possono anche essere utilizzati per inviare delle informazioni.

Tuttavia di default attraverso il comando kill viene inviato il segnale SIGTERM(15) che (di default, a meno che non venga esplicitamente ignorato dal processo) comporta la terminazione.

Tra i vari segnali, SIGKILL(9) termina il processo senza poter essere bloccato o ignorato.

Attraverso il comando ps -e possiamo vedere il pid di tutti i processi del sistema.

Per terminare un processo possiamo fare kill *pid del processo*

Quando un processo viene terminato da un segnale non ha un proprio exit status. L'exit status viene calcolato come 128 + numero del segnale (esempio: exit status di un segmentation fault è 139 -> SIGSEV(11)).

Funzioni

La bash è un linguaggio, ed è possibile definire delle funzioni:

```
function name { cmd-list }
function pippo { echo "arg1= $1 arg2= $2 numero argomenti= $#"; }
```

La chiamata alla funzione pippo senza argomenti restituisce arg1= arg2= numero argomenti=0

Gli argomenti sono \$1, \$2, ...

Il numero di argomenti è dato da \$#.

Può essere usato il comando return come in C.

L'attributo local può essere utilizzato per dichiarare variabili locali.

Condizioni

Per effettuare il confronto tra valori inizialmente sono stati implementati due comandi chiamati test e [.

I loro file si trovano nella directory /bin.

Questi permettevano di implementare un valutatore di espressioni al di fuori della shell.

Esempio:

- [1 -eq 2] verifica l'uguaglianza tra 1 e 2 (restituisce 1 – falso)
- [-f pippo] verifica se nella directory corrente è presente il file pippo
- [-w /Scrivania/pippo] verifica se il file pippo sulla scrivania è scrivibile
- [-r pippo] verifica se è possibile accedere al file in lettura

Nella shell moderna, nonostante il programma sia ancora presente, viene implementato all'interno della shell attraverso il comando [[...]]

Le doppie quadre sono state introdotte per eseguire operazioni come la comparazione booleana.

Questo perché nel comando [1 < 2] il minore < viene interpretato come redirezione di input.

Attraverso il comando [[1 < 2]] riusciamo ad effettuare la comparazione booleana. Attenzione, questo minore confronta le stringhe e non gli interi!

Per confrontare gli interi si utilizza [[1 -lt 2]]

A differenza delle quadre singole, le doppie quadre danno errore se si prova ad espandere una variabile non esistente o se si prova ad eseguire il comando -f senza passare ulteriori argomenti.

Costrutti condizionali e cicli

if

Una volta definite le espressioni booleane possiamo definire dei costrutti condizionali come l'if.

La sintassi è la seguente:

```
if  tst-commands; then
    consequent-commands;
[elif more-tst-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
```

elif sta per else if. In qualsiasi caso l'if viene sempre chiuso da fi.

For

La sintassi del for è la seguente:

```
for name in words; do commands; done
```

Per esempio il comando

```
for i in qui quo qua; do echo $i; done
```

Stampa qui quo qua.

Possiamo comporre le parti del for con espressioni che abbiamo già visto, come il risultato dell'esecuzione di un comando:

```
for i in $(ls); do echo $i; done
```

Esiste anche il for versione C:

```
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

Esempio: stampa dei colori da 0 a 255

```
for i in {0..255}; do
    printf "\x1b[38;5;%dm%d " $i $i
done
echo
```

La shell ha un linguaggio di programmazione abbastanza vecchio, con delle peculiarità diverse dai linguaggi moderni. Sul sito <http://mywiki.wooledge.org/BashPitfalls> sono presenti gli errori più comuni fatti dagli utenti nella programmazione di script.

Un altro modo per controllare la correttezza degli script è utilizzare il comando shellcheck.

Quando gli script diventano più complessi conviene passare a linguaggi di programmazione come python.

Autenticazione

lunedì 21 dicembre 2020 09:34

Identificazione e autenticazione

Nei sistemi Unix-Like tipicamente gli utenti si identificano con uno username e si autenticano con una password. Esistono due file per gestire questi meccanismi:

- **etc/passwd** file di testo dove ogni riga corrisponde ad un utente del sistema (vedi man passwd(5)). Questo file viene utilizzato dal sistema per autorizzare il login. Una volta conteneva come secondo campo di ogni riga un hash della password, ma essendo leggibile da chiunque questo campo è stato sostituito da una x, ed è nato il file etc/shadow
- **etc/shadow** file con la stessa struttura del precedente, ma contiene l'hash delle password. Questo file è leggibile solo dal proprietario (root) oppure dagli utenti del gruppo shadow.

Il kernel identifica ogni utente con un numero intero (UID), e analogamente i gruppi con un GID. C'è un gruppo principale di appartenenza, e dei gruppi secondari specificati nel file in /etc/group. Da riga di comando è possibile usare il comando id per vedere UID, GID e gruppi secondari a cui si appartiene.

L'UID zero corrisponde a root, l'amministratore di sistema. In molti sistemi corrisponde a Dio, quando si fa qualcosa con UID zero non si va nemmeno a controllare i permessi.

Tradizionalmente i processi si dividono in:

- Privilegiati UID=0
- Non privilegiati UID≠0

In pratica le Linux capabilities permettono una maggiore granularità, si veda capabilities(7).

Autorizzazione

lunedì 21 dicembre 2020 09:53

<http://pages.cs.wisc.edu/~remzi/OSTEP/security-access.pdf>

L'autorizzazione consiste nel decidere se un certo soggetto può accedere o meno in qualche modo ad una risorsa. Il sistema deve avere delle informazioni per poter decidere. I modi per memorizzare queste informazioni sono due:

- **Access Control List:** corrispondono ad un buttafuori ad una festa privata con una lista degli invitati
- **Capabilities** (non c'entrano nulla con le Linux Capabilities): corrispondono a delle chiavi che permettono a chiunque le possieda di accedere alle relative risorse.

L'idea generale è semplice, ma poi vanno considerati i dettagli:

- Dove memorizzare queste informazioni/quanto spazio serve
- Cosa bisogna aggiornare quando aggiungiamo/eliminiamo utenti

Unix utilizza una versione ottimizzata di Access Control List in cui usa 9 bit per codificare tutta la lista. È molto ottimizzata ma è ridotta: ad ogni file sono associati 3 bit di permessi per ogni categoria (r w x per proprietario, gruppo, altri). Con le ACL di Unix non è possibile indicare permessi specifici per un utente, se non creando un gruppo per ogni granularità e associare gli utenti ai gruppi in base ai permessi desiderati (macchinoso).

MAC vs DAC

Indipendentemente dall'uso di ACL/capabilities, chi decide chi può/non-può accedere a una risorsa? Se è il proprietario, si parla di **DAC: discretionary access control**.

Se invece è una qualche autorità impone le regole (e.s. ambito militare), si parla di **MAC: mandatory access control**. In questo caso per esempio si potrebbe distinguere tra i file Secret e i file Top Secret, e questa suddivisione implica i permessi di accesso ai file.

I sistemi più comuni seguono il DAC, anche se non al 100%. Per esempio, tradizionalmente root può leggere qualsiasi file.

Esistono varianti MAC di Linux; per esempio, <https://github.com/SELinuxProject>

Su Linux è infatti possibile aggiungere sistemi che impongono ulteriori restrizioni oltre a quelle già presenti.

Principio del minimo privilegio

Definizione

In ogni momento, ciascuna entità dovrebbe avere il minimo privilegio, che gli permetta di eseguire i suoi compiti legittimi. Detto diversamente, nessun processo dovrebbe mai poter accedere a più risorse del minimo indispensabile per il suo corretto funzionamento.

Questo perché, se vengono commessi errori, il danno è limitato ai file accessibili dai permessi associati a quel processo. Se decidiamo di cancellare tutti i file contenuti in bin come root, non ci ferma nessuno e facciamo fuori tutti i comandi, se proviamo a farlo come utenti normali questo ci viene impedito e il danno viene limitato. Questo potrebbe essere frutto non solo di errori ma anche di attacchi da parte di software malevoli.

In generale è importante considerare il ruolo che abbiamo in quel momento: lo stesso utente può avere diversi ruoli. Per esempio, noi siamo amministratori del nostro sistema (abbiamo la facoltà di agire come tali) ma non lo siamo in ogni momento e non è nemmeno necessario esserlo (non serve essere amministratori per aprire un pdf).

In Unix, login(1) è il comando che permette di fare login. Questo processo deve poter leggere etc shadow per verificare che la password sia corretta, deve poter accedere alla home directory, lanciare la nostra shell... Questo processo gira quindi con privilegi elevatissimi, ma quando termina la fase di login il nostro utente non ha più questi privilegi.

Per aumentare i privilegi storicamente ci sono due comandi che permettono di farlo. Storicamente si usava il comando su(1) (che permette di diventare superuser). Ad oggi si usa molto di più il comando sudo(1).

La differenza principale è che su(1) per farti diventare superuser ti chiede la password del superuser. Ciò significa che utenti diversi devono sapere la password di root (che non è una buona idea). Nei sistemi moderni sudo(1) chiede invece la password dell'utente, e se il nostro utente è specificato tra quelli che possono eseguire questo comando, diventiamo amministratori di sistema al fine di eseguire un singolo comando.

Si tratta quindi di un'implementazione del principio del minimo privilegio.

UIDs

Ad ogni processo corrispondono 3 UID, potenzialmente diversi:

- **real UID** — il proprietario del processo, sostanzialmente chi può usare kill verso il processo;
- **effective UID** — identità usata per determinare i permessi di accesso a risorse condivise, per esempio ai file;
- **saved UID**
- (solo Linux: FS UID, lo ignoriamo; si veda credentials(7))

Al login tutti e 3 coincidono.

Tramite setuid(2) un processo può modificare l'effective UID, facendolo diventare come il real o il saved. Se il chiamante è privilegiato, può modificare tutti e tre come vuole.

Un nuovo processo "eredita" gli id del parent e di solito, execve non cambia gli id del processo chiamante. Tuttavia c'è un bit particolare: se il file eseguibile ha il bit set-userid abilitato, allora effective UID e saved UID diventano quelli del proprietario del file. Questo solitamente accade per i file di proprietà di root.

Questo si vede perché tra i permessi che vediamo con ls -l c'è una s. Ciò significa che quando eseguiamo questo comando diventiamo automaticamente root.

Per esempio: supponiamo che un processo venga lanciato con effective UID=0 (con bit set-userid). Se il programma è scritto bene, quando fa azioni che non richiedono privilegi da root, lo cambia impostandolo uguale a quello real. Quando invece deve svolgere azioni che richiedono il privilegio, può reimpostare l'effective a zero "copiandolo" dal saved.

Esempio incApache

Se un processo ha bisogno di fare delle azioni con privilegi elevati, ma solo all'inizio, è meglio se quando ha terminato di eseguire le azioni privilegiate rilascia completamente i privilegi da root ossia riporta sia effective che saved uguali a real.

Per esempio i privilegi servono per fare il bind su porte inferiori a 1024 o per fare la chroot. Dopo queste azioni rilascia i privilegi: se ci fosse un bug nel programma in questo modo non potrebbe essere sfruttato per attaccare il sistema.

Chroot jails e namespace

chroot(2) è una syscall che permette di modificare il significato di "/" nella risoluzione dei percorsi assoluti, ossia di limitare l'accesso al file system. La modifica è per il processo e tutti i (futuri) figli. Per eseguirla è necessario che il processo sia privilegiato. Nota: i FD aperti non vengono toccati.

Chroot permette di creare le cosiddette **chroot jail** limitando la visibilità del file-system ad una directory (e sotto-directory) per questioni di sicurezza. Si tratta di un'applicazione del principio del minimo privilegio, ma se il processo rimane privilegiato e/o la syscall non è correttamente associata a una chdir(2) è possibile "uscire di prigione".

Per esempio, anche in caso di bug, in incApache un client non può guardare fuori dalla www-root.

Queste chroot jail corrispondono ad antenati dei namespace, spazi di nomi che permettono di partizionare processi, utenti, network ecc. All'interno di un sistema.

Container

Il meccanismo moderno per creare isolamento sono i container (usati, per esempio, dal famoso Docker).

Introduzione

domenica 20 dicembre 2020 15:28

<http://pages.cs.wisc.edu/~remzi/OSTEP/security-intro.pdf>
<http://pages.cs.wisc.edu/~remzi/OSTEP/security-authentication.pdf>

Il sistema operativo si trova alla base di tutte le applicazioni. Costituendo le fondamenta delle applicazioni, deve garantire dei sistemi di sicurezza che impediscono ad una applicazione di prendere il controllo influendo sul funzionamento dell'intero sistema.

- Allo stesso modo non possiamo scrivere un sistema operativo sicuro se il livello inferiore (hardware) non è sicuro. Supponiamo che una qualsiasi applicazione possa chiamare un'istruzione macchina per passare a livello kernel e fare ciò che vuole. Non avrebbe senso implementare un sistema operativo che cerca di impedire questa interazione.

In generale per descrivere l'idea di sicurezza ci sono 3 proprietà, descritte dall'acronimo CIA

- **Confidentiality** (Segretezza)
- **Integrity** (Integrità)
- **Availability** (Disponibilità)

L'idea di segretezza è quella di avere cose che non possono essere lette se non dai proprietari (ad esempio file). L'integrità è quella caratteristica per cui nessuno possa sovrascrivere, manomettere, corrompere i dati di un altro utente (senza permesso). La disponibilità è quella proprietà per cui, se c'è un servizio, deve essere reso disponibile ai legittimi utenti.

Attacchi relativi alla disponibilità possono essere per esempio i DDoS (Denial Of Service), in cui si impedisce ad un utente di accedere ad un certo servizio.

In generale quello che gli utenti vogliono è una condivisione controllata delle risorse.

Contesto

Per fare qualsiasi cosa che coinvolga l'utilizzo di risorse un processo deve fare delle system call.

Il kernel quando viene attivata una system call:

- Controlla che la richiesta sia sensata, ossia ci sia un numero valido di system call e che i parametri delle chiamate siano corretti;
- Rispetta la politica di sicurezza.

Terminologia:

- L'entità che fa la richiesta è chiamato principal o subject
- Le richieste sono relative a una risorsa, l'object
- E, tipicamente, una modalità di accesso; per esempio, lettura o scrittura

Nel valutare se eseguire o meno bisogna considerare il contesto: aprire un file in lettura può avere risposte diverse rispetto a chi fa la richiesta. Questo in generale è vero per i sistemi desktop (Windows/Linux): quando viene mandato in esecuzione un programma su questi sistemi, generalmente ha gli stessi permessi dell'utente che lo ha mandato in esecuzione.

Sui sistemi mobili invece ogni applicazione ha i propri pacchetti di permessi (fotocamera, contatti, posizione...), anche se è lo stesso utente a mandarla in esecuzione.

Chiaramente un'utente non interagisce direttamente con il kernel, ma servono:

- Un'associazione fra utenti e processi;
- Un modo per verificare l'identità degli utenti.

Per fare questo vengono utilizzate le AAA

AAA

- **Authentication:** Autenticazione – Verifica dell'identità
- **Authorization:** Autorizzazione – Applicazione di una politica di sicurezza, decidere se accettare/rifiutare le richieste
- **Accounting:** Contabilità: logging e gestione del consumo delle risorse

Uso interattivo della shell

martedì 29 settembre 2020 15:58

Cosa fa la shell?

- In modalità interattiva stampa un prompt, una sequenza di caratteri che indica che è in attesa di comandi
- Legge l'input
 - Lo spezza in token: parole e operatori
 - Espande gli alias – Per esempio alias ls='ls -color=auto'
 - Fa il parsing in comandi semplici e composti
- Esegue varie espansioni (\${ } \$ *?[])
- Esegue le varie redirezioni dell'input/output
- Esegue il "comando", tipicamente un eseguibile/script esterno (ma può essere anche una funzione o un comando built-in) - Per esempio il comando ls si trova in /bin/ls
 - Utilizzare comandi esterni permette una maggiore flessibilità: è possibile comporre più comandi implementati in file separati per ottenere comandi più complessi
- Tipicamente ne aspetta la terminazione
 - Per non far aspettare la terminazione del comando, inserire & al termine del comando.
Permette di mettere in background l'esecuzione di alcune operazioni.
- Ricomincia

Comandi della shell

Man: contiene tutte le pagine di manuale dei comandi esterni

Help: contiene tutte le pagine dei comandi interni alla shell

Per sapere se un comando è esterno o interno è possibile utilizzare il comando type (esempio type alias).

Differenza tra comandi esterni e interni:

- I comandi interni vengono eseguiti direttamente dalla shell;
- I comandi esterni vengono eseguiti attraverso un nuovo processo in cui viene invocato il comando. Al termine dell'esecuzione del comando il controllo ritorna alla shell.

Escaping e quoting

Alcuni caratteri hanno significati speciali (per esempio, lo spazio) e devono essere messi fra virgolette o preceduti da un backslash per essere utilizzati.

- I singoli apici permettono l'inserimento di qualsiasi carattere, a parte gli apici stessi
- Le doppie virgolette trattano in maniera speciale \$ e \
- La forma '\$...' permette di espandere i \... alla ANSI-C; per esempio: echo '\$'ciao\nmondo\x21' → ciaomondo!

Variabili

Attraverso la shell è possibile creare delle variabili ed espandere il valore di queste variabili.

Per definire una variabile è necessario inserire il nome (per convenzione tutto maiuscolo) e associargli un valore.

PIPPO=pluto

Attenzione a non inserire degli spazi, la shell cercherebbe di eseguire le varie parti come comandi.

Operazioni su variabili

Per espandere il valore di una variabile (stampare il suo valore) si stampa \$nomeDellaVariabile (esempio echo \$PIPPO).

Per stampare stringhe di seguito all'espansione della variabile, si usano le parentesi graffe:

```
echo ${PIPPO}altra_stringa  
plutoaltra_stringa
```

Stampare il valore della variabile se esiste, altrimenti stampare un'altra stringa

```
echo ${PIPPO:-altra_stringa}
```

Variante: se non esiste stampa l'altra stringa, poi crea la variabile e assegna l'altra stringa come valore

```
echo ${PAPERINA:=altraroba}
```

Variabile \$\$

Una variabile particolare è la variabile \$\$, che corrisponde al PID (Process Identifier) della bash.

Per stampare l'elenco dei PID di tutti i processi è possibile utilizzare il comando ps.

Esporre una variabile

Le variabili di ambiente sono visibili solamente alla shell ma non ai programmi che vengono mandati in esecuzione. Per esportare una variabile e renderla visibile ai programmi mandati in esecuzione si usa il comando `export`.

In questo modo è possibile modificare il comportamento dei programmi che vengono mandati in esecuzione.

Variabile \$PATH

La variabile `$PATH` contiene una serie di percorsi separati da :

Specificata tutte le directory del file system dove sono presenti i comandi eseguibili esterni alla shell. Scrivendo un comando, la shell ricerca il comando nel primo percorso specificato dalla variabile `PATH`, se lo trova esegue il file corrispondente. Se non lo trova procede nella ricerca nel secondo percorso specificato da `PATH`...

Se al termine della ricerca non trova il comando restituisce "command not found".

Per sapere in quale directory si trova un comando è possibile utilizzare il comando `which` (esempio `which ls`).

Startup Script

Supponiamo di modificare il valore della variabile `$PATH` nella shell: nel momento in cui chiudiamo la finestra in cui è stata effettuata la modifica, questa viene persa.

Per far sì che la modifica sia permanente è necessario aggiungere l'`export` all'interno di un file che viene lanciato automaticamente ogni volta che viene aperta una shell.

Lo script in questione della bash si chiama `~/.bashrc`

Espansioni

Dopo il parsing vengono eseguite le varie espansioni dei comandi, prima che vengano passati ai programmi:

- La tilde `~` equivale alla home del nostro utente
 - `echo ~` restituisce `/home/martina`

È possibile aggiungere il nome di un altro utente per vedere la sua home directory (`echo ~root`).

Attraverso il comando `vim /etc/passwd` è possibile vedere l'elenco degli utenti del sistema. Per la maggior parte di essi non è possibile loggarsi nel sistema, servono soltanto a tenere separati i permessi per questioni di sicurezza.

- Un'altra espansione è la seguente:
 - `a{A, B, C}z` restituisce `aAz, aBz, aCz`
 - `a{1..10}z` restituisce `a1z, a2z, a3z, ..., a10z`

Possono essere utilizzate per rinominare un file: anziché fare: `mv pippo pippo.png`

Possiamo fare: `mv pippo{,.png}`

- `$(cmd)` e '`cmd`'
Hanno la semantica di eseguire il comando ed espandere l'output in linea. Per esempio:
 - `echo ls` stampa semplicemente `ls`
 - `echo $(ls)` equivale all'output del comando `ls`

Le espansioni che non sono avvenute all'interno di doppie virgolette (e contengono spazi, tab o newline) vengono spezzate in parole separate.

Per esempio:

`PIPPO="a b c"` -> dichiaro variabile `PIPPO` e assegno valore "a b c"
`echo $PIPP0` -> stampa a b c prendendoli come 3 argomenti separati
`echo "$PIPP0"` -> stampa a b c prendendolo come un unico argomento

- Espansione delle espressioni regolari `* ? [a0b-z]`
Dato il comando `ls read*`, viene stampato `read` e `read.c`
Questo perché l'asterisco corrisponde a prendere tutte le stringhe che iniziano per `read` e continuano con una sequenza di lunghezza arbitraria (anche 0) di qualsiasi altro carattere.
Esempio `g++ *.cpp` -> compila qualunque file .cpp

Allo stesso modo funziona l'espressione regolare `? che però viene sostituito con un solo carattere.`

La notazione compatta `[a0b-d]` funziona come nelle espressioni regolari.

- La shell sa anche contare: `$((espressione))` calcola il risultato dell'espressione

Per convenzione i file che iniziano per `.` sono per convenzione nascosti.

Il comando grep permette di filtrare le linee che contengono una certa espressione regolare.

Esempio: grep a (invio) paperina (invio) pluto (invio)

Redirezioni

- Output
 - > per redirigere in un file sovrascrivendo il contenuto precedente
 - >> per redirigere in un file scrivendo in fondo al contenuto già presente (append)
- Input <
 - Esempio: sequenza di comandi ls -l > pluto grep a < pluto
(si potrebbe anche fare direttamente grep a pluto)

Ci sono file speciali come /dev/null che permette di reindirizzare output su un terminale che verrà ignorato e /dev/zero che permette di leggere zeri all'infinito (come il comando yes – stampa y all'infinito).

Utilizzo delle system call

giovedì 22 ottobre 2020 12:26

Un processo non può interagire direttamente con l'hardware, ma deve farlo attraverso una chiamata di sistema (system call o syscall).

Una system call è una chiamata controllata all'interno del kernel: la modalità di esecuzione della CPU va in modalità kernel ma si può accedere solamente ad un indirizzo prefissato scelto dal kernel. Non si può quindi decidere di accedere ad aree di memoria arbitrarie in modalità kernel.

Per fare delle syscall è necessario utilizzare delle istruzioni assembler, ma la libreria C offre delle funzioni wrapper, che incatolano la chiamata alla syscall.

Funzioni wrapper

Le funzioni wrapper preparano gli argomenti per la system call secondo le convenzioni del sistema. Per esempio in Linux i parametri devono essere caricati in registri, mentre in sistemi come Xv6 i parametri devono essere sullo stack come per le chiamate di funzioni normali.

Specificano poi il numero della system call da chiamare in un registro chiamato EAX, ed eseguono una trap per passare alla modalità kernel.

Il kernel controlla la validità del numero della syscall e degli argomenti. Se questi sono validi, esegue la richiesta e salva il risultato della system call in un registro. Dopodiché ritorna alla modalità utente tramite un'istruzione speciale (IRET).

Una volta terminata la modalità kernel si ritorna alla funzione wrapper, che controlla il risultato inserito nel registro. In caso di errore impone errno (simile ad una variabile globale) e restituisce un codice di errore, tipicamente -1, altrimenti restituisce il risultato al chiamante.

Esiste delle syscall

Quando eseguiamo delle funzioni o delle syscall è necessario controllare sempre il valore di ritorno. In caso di errore errno indica la ragione del fallimento: errno può essere considerata come una specie di variabile globale intera, ma ogni thread (flusso di esecuzione) ha la sua. Non è detto che in caso di successo venga azzerata, modificata.

Perror permette di stampare una versione testuale della ragione per cui è fallita la syscall.
Strerror analogamente dà la versione testuale del codice di errore di perror.

Vedere errno(3) perror(3) e strerror(3).

Tipi di dato

Alcune informazioni vengono memorizzate in int/long/short/... a seconda del sistema.
Per esempio, quando abbiamo parlato di PID abbiamo detto che sono interi: tuttavia non sappiamo se vengano memorizzati a 32 bit, 64 ecc.

Per scrivere del codice portabile non possiamo quindi assegnare un tipo ai PID, perché in sistemi diversi possono essere utilizzati tipi diversi. Si possono utilizzare degli alias definiti dagli header: per esempio getpid (chiamata di sistema che restituisce il pid del mio processo) restituisce un pid_t, che è un typedef del tipo che serve per memorizzare i PID all'interno di questo sistema. In ogni sistema va ad utilizzare il tipo utilizzato da quel particolare sistema.

Tuttavia questo crea dei problemi quando si vogliono stampare con printf e funzioni simili. Un modo per aggirare il problema è fare un cast ad un tipo sicuramente più grande, come long o long long.
Da C99 in poi è possibile utilizzare i tipi interi più grossi possibile (intmax_t e uintmax_t), stampandoli con %jd e %ju, definiti in stdint.h.

Debugging

mercoledì 25 novembre 2020 22:53

- In generale per fare il debug di un kernel si utilizza il remote debugging, ossia il debugger sta in una macchina diversa rispetto a quella dove risiede il kernel di cui vogliamo fare il debug.

Lanciando Xv6 con il comando debug la macchina virtuale Qemu parte in una modalità particolare che attende la connessione con gdb. Una volta stabilita la connessione gdb è in grado di controllare tutta la macchina, e di eseguire quindi il debug non solo del kernel ma anche dei programmi utente, e la transizione tra modalità kernel e utente (e viceversa).

Lista processi

Dalla console di Xv6 ctrl+P mostra per ogni processo: PID, PPID, stato, nome, back trace (elenco di indirizzi all'interno della pila di chiamate nel kernel. Utile per debuggare un processo). Per decifrare la lista di indirizzi della back trace possiamo utilizzare info symbol *indirizzo* all'interno di gdb.

Debug di programmi utente

Dal punto di vista del sorgente abbiamo una serie di nomi (nomi di variabili, di funzioni..) che però non sono utili per la CPU a livello di esecuzione. Questi nomi non vengono infatti riportati nell'eseguibile in fase di compilazione, perché occuperebbero spazio inutile.

In fase di debug però i simboli diventano fondamentali, perché indirizzi e codice assembler non sono riconoscibili dall'utente. Compilare utilizzando il flag (-g oppure –ggdb) permette di includere i simboli del programma all'interno dell'eseguibile, rendendo facile effettuare il debug del programma.

Nel caso di Xv6 il kernel ha i suoi simboli. Tuttavia anche ogni programma utente ha i suoi simboli, che devono necessariamente essere aggiunti se il programma deve essere debuggato.

Se abbiamo già caricato dei simboli da programmi utente, eliminiamoli tutti e impostiamo quelli del kernel:

- symbol-file
- symbol-file kernel/kernel

Aggiungiamo i simboli con add-symbol-file; per esempio:

- add-symbol-file user/_ciao
- b main
- c

Possiamo seguire una system-call anche all'interno del kernel; una delle più semplici è getpid: modifichiamo ciao.c e vediamo cosa succede.

Programmi utente

mercoledì 25 novembre 2020 10:59

Le system call e le funzioni disponibili per scrivere programmi utente sono elencate in user/user.h. Per i casi più semplici basta un solo include: #include "user/user.h".

La libc è veramente ridotta (12 funzioni utente), e anche il numero di system call è piuttosto limitato (22). Alcune delle funzioni della libreria C assomigliano a quelle originali ma non sono uguali: per esempio printf prende come primo argomento un file descriptor per sapere dove andare a scrivere. Questo significa che non ci devono essere inizializzazioni su standard input, output... Di conseguenza l'entry point dell'ELF non è la libc ma è direttamente il main.

Per questo motivo il main deve uscire per forza con exit(). Se facesse return, in condizioni normali tornerebbe alla funzione della libc che lo ha chiamato; in questo caso non è stato chiamato da nulla, quindi fallirebbe.

Le system call sono implementate in modo simile alla normalità con dei wrapper. Vengono implementate in sys_s. Per ogni system call viene eseguita una macro che corrisponde a:

- Inserire nel registro eax un numero che corrisponde a SYS_ seguito dal nome della syscall;
- Fare int T_SYSCALL (genera un interrupt di numero 64).

Aggiungere un programma

Dentro a user creiamo user.c

```
#include "user/user.h"
int main()
{
    printf(1, "Ciao SETI!\n");
    exit();
}
```

Nel Makefile aggiungiamo \$U/_ciao a UPROGS
Eseguiamo make — crea il file user/_ciao che verrà “automagicamente” incluso in fs.img
Si tratta di un ELF, ma girerà solo su Xv6, con Linux si schianta

Avvio

martedì 17 novembre 2020 13:15

Consideriamo un processore Intel i386 a 32 bit.

Quando accendiamo la macchina il processore si trova in modalità reale (o indirizzamento reale). Di fatto questo significa che il processore parte a 16 bit, per questioni di compatibilità: in questo modo dovrebbe essere compatibile con computer molto più vecchi di lui.

L'indirizzo da cui parte l'esecuzione in esadecimale è 0FFFFFFF0H, detto anche $2^{32} - 16$. Questo significa che viene invocato qualunque cosa ci sia alla locazione $2^{32} - 16$. Poiché il processore è in grado di indirizzare 2^{32} indirizzi, invochiamo un indirizzo che si trova nell'ultimo pezzo in alto della memoria fisica.

A quell'indirizzo deve essere mappato un codice chiamato firmware, contenuto in qualche ROM (memoria in grado di mantenere il contenuto anche senza alimentazione). Il codice presente a quell'indirizzo è il BIOS, ed è firmware che arriva con la scheda madre. Svolgerà quindi azioni diverse a seconda della scheda madre utilizzata.

In generale:

- Inizializza l'hardware presente;
- Cerca un dispositivo da cui poter fare il boot, ossia un dispositivo a blocchi (disco) il cui primo blocco termina con la "firma" 0x55 0xAA agli offset 510 e 511. Se lo trova, prova a fare il boot del sistema operativo. L'ordine con cui vengono scansionati i dischi può essere selezionato, ma il boot avviene sul primo disco che soddisfa i requisiti;
- Quando il BIOS trova il blocco (detto boot block) lo carica in memoria all'indirizzo fisico 0x7c00.
- Lo esegue.

Perché l'indirizzo 0x7c00?

Nell'anno 1981 il msDOS 1.0 richiedeva almeno 32 KB di memoria.

Gli sviluppatori del BIOS per il pc 5150 di IBM hanno pensato di lasciare più spazio possibile all'interno di quei 32 KB per il sistema operativo.

Hanno quindi scelto l'ultimo KB disponibile in cima alla memoria, ovvero $32768 - 1$ KB. Viene lasciato un KB perché:

- 512 byte servono per il boot sector;
- 512 byte vengono utilizzati per dati/stack.

Dopo il boot quella zona di memoria non serve più, e può essere riutilizzata dal sistema operativo.

Organizzazione della memoria

Sempre per ragioni storiche il resto della memoria RAM viene così suddiviso:

- I primi 640 KB (indirizzi 0-0xA0000) vengono detti memoria convenzionale: ai tempi del msDOS la memoria disponibile era 640 KB.
- Sopra questa zona, la parte che va da 640 KB – 1 MB (indirizzi 0xA0000-0x100000) è detta memoria alta ed è riservata alle periferiche: significa che ci sono degli indirizzi di memoria che non fanno riferimento alla memoria ma fanno riferimento alle periferiche (ad esempio la scheda grafica).
- Sopra al MB la memoria è detta memoria estesa.

Nei PC un po' più moderni la parte alta dei 32 bit è riservata per altre periferiche.

Boot block

In Xv6 i sorgenti del boot block sono:

- kernel/bootasm.S
- Kernel/bootmain.c

I blocchi devono avere una dimensione massima complessiva di 510 byte

Il makefile li mette insieme e "firma" il boot block tramite sign.pl (una utility in perl che legge il contenuto del boot block, verifica che la lunghezza sia inferiore a 510 byte, fa il padding con degli 0 per farlo arrivare a 510 byte e aggiunge 0x55 0xAA).

Dopodiché crea un'immagine del disco, xv6.img, che contiene:

- Boot block come primo blocco
- Il kernel dal secondo blocco in poi

L'immagine viene creata utilizzando 3 volte il comando dd, che copia dischi (roba a blocchi). La dimensione di default di un blocco è 512 byte. La sequenza di dd restituisce questo risultato:

- Inizialmente crea un disco di 512 blocchi di 512 byte (256 k) pieni di zeri
- Viene scritto il boot block all'inizio del file mantenendo il resto (zeri)
- Viene scritto il kernel a partire dal blocco numero 1 (nello 0 c'è il boot block).

Il disco xv6.img viene utilizzato solo per il boot e non contiene nessun file system.
L'altro disco, fs.img, contiene invece il file system.

Cscope

Siccome ci sono parecchi file, può essere utile usare cscope.

Permette di ricercare variabili, simboli, funzioni chiamate da/che chiamano una funzione, ecc. all'interno di tutti i file della directory.

Può essere usato da terminale - (ctrl+D) per uscire

Bootasm.S

Cosa fa al boot Xv6?

La partenza del processore avviene in modalità "reale", a 16 bit. È in modalità segmentata, ma i segmenti corrispondono alla base shiftata di 4. Questo trucco serve per riuscire ad indirizzare 1 MB, nonostante i 16 bit di indirizzamento.

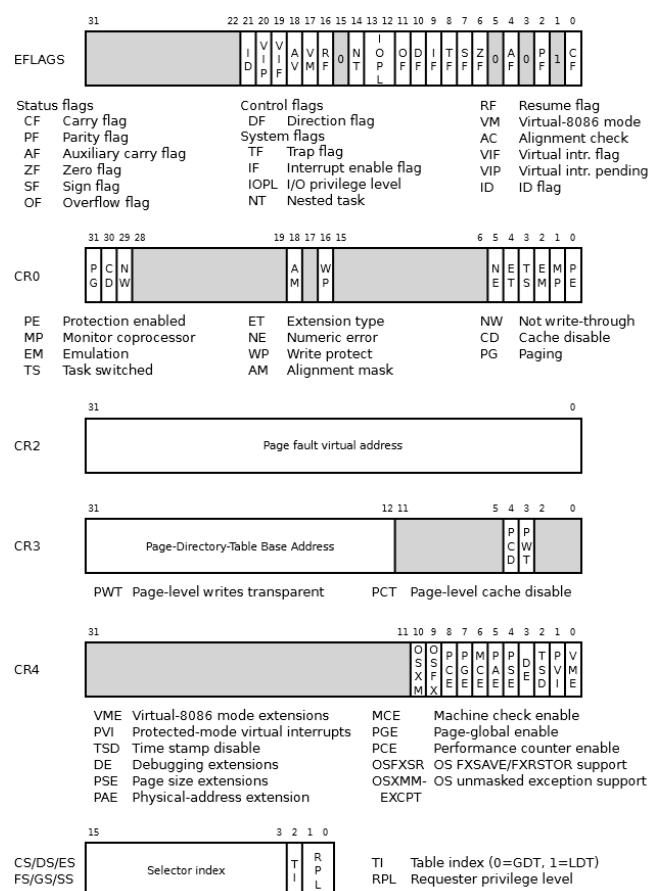
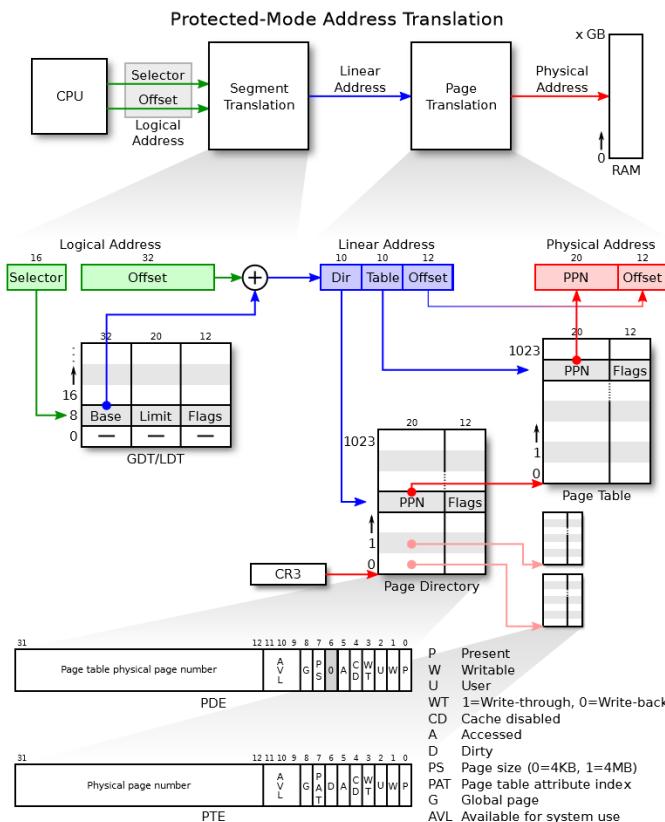
Viene poi disabilitato il gate A20, un trucco creato per supportare programmi che sfruttavano l'idea di avere 1 MB massimo di memoria anche con computer che permettevano di indirizzare dimensioni maggiori.

Avviene poi la preparazione della modalità protetta a 32 bit che prevede che il bios carichi il bootblock in memoria e lo esegua. In particolare:

- LGDT (Load Global Descriptor Table): caricamento della tabella dei segmenti
 - Le modalità protette di Xv utilizzano una tabella dei segmenti e anche se i sistemi moderni non usano la segmentazione, non c'è modo di evitarla. All'interno di un processore intel a partire dall'indirizzo virtuale c'è sempre la trasformazione tramite segmentazione nell'indirizzo che intel chiama lineare, che può opzionalmente ulteriormente essere trasformato tramite paginazione in indirizzo fisico.
 - Se non vogliamo utilizzare la segmentazione utiliziamo per tutti i segmenti base 0 e limite +∞
- Avviene il passaggio alla modalità protetta abilitando il flag PE nel registro CR0
- Viene effettuata un'operazione di salto "far" che prevede l'uso di offset + selettore

A questo punto ci troviamo in modalità 32 bit e il codice assembler prepara uno stack per il codice C (niente heap, niente libc). Infine si passa al codice C, effettuando una chiamata al bootmain.

Gestione della memoria in Xv6



Bootmain.c

Il bootblock tramite il file bootmain.c carica il kernel dal disco (che contiene il boot loader e il kernel).
Esegue le seguenti operazioni:

- Controlla che il kernel sia un ELF
- Carica i segmenti dell'ELF a 1 MB (da 1 MB in su, all'inizio della memoria estesa). Per fare questo non può utilizzare system call come open, read ecc., ma può solo interagire direttamente con il controller del disco.

Non stiamo ancora utilizzando la paginazione: vengono quindi utilizzati indirizzi fisici.

A regime, una volta terminata l'inizializzazione, il kernel sarà nella parte alta della memoria; i 4 GB indirizzabili da un processore a 32 bit saranno così suddivisi:

- I primi 2 GB (più bassi) vengono riservati all'utente
- Gli altri 2 (più alti) al kernel

In particolare il kernel sarà a $2G + 1 \text{ MB} = 0x80100000$

Una volta caricato il kernel si salta all'entry point dell'ELF : `_start`

Il codice di `_start` è in entry.S

Paginazione su x86

Come viene mappato il kernel in memoria?

Con x86 abbiamo 32 bit di indirizzamento e, tipicamente, pagine da 4KB. Si tratta di una dimensione delle pagine equilibrata, che permette di avere una tabella delle pagine di dimensione limitata e di avere poca frammentazione interna (spazio riservato ai processi non utilizzato).

Se, con queste caratteristiche, venisse utilizzata una sola tabella delle pagine, questa sarebbe di dimensione 4MB, poiché i 32 bit di indirizzamento vengono così suddivisi:

- 12 bit per l'offset (pagine da 4KB)
- 20 bit per il numero di pagina (1 MB – 1 milione di pagine)

Per ogni pagina servono 4 byte, poiché 20 bit indirizzamento + flags. Otteniamo quindi 4 MB totali

Per ogni processo, 4MB contigui sono tanti: supponiamo di avere 100 processi, bisognerebbe usare 400 MB (quasi mezzo GB di RAM) riservati solo alla tabella delle pagine.

Una delle soluzioni che possono essere utilizzate è quella di usare delle tabelle multi-livello: non c'è un'unica tabella delle pagine, ma tante tabelle a livelli diversi.

Nel caso del i386 possiamo avere tabelle a 2 livelli con pagine da 4KB (soluzione più utilizzata), oppure un solo livello con pagine da 4MB (soluzione utilizzata solo in partenza, perché permette di mappare il kernel in una singola pagina).

Con i processori più moderni a 64bit, l'approccio è analogo: si basa sempre su tabelle multi-livello, ma in questo caso possiamo avere fino a 4 livelli, con pagine da 4KB, 2MB o 4MB.

Entry.S

Codice in assembler.

Entry.S prepara una tabella delle pagine chiamata entrpgdir, definita in main.c.

In questa tabella abbiamo solo due pagine mappate: la pagina 0 e KERNBASE (2GB).

Quello che succede è che:

- Nei primi 4 MB della memoria fisica abbiamo mappato il kernel (meno di 200 kB) in posizione 1MB;
- Questa pagina da 4MB viene mappata due volte nello spazio di indirizzamento virtuale, nella tabella delle pagine: nei primi 4MB della pagina 0 e nei primi 4MB della pagina KERNBASE.

Al momento stiamo eseguendo il codice nella pagina 0.

La stessa area viene mappata in memoria due volte. Questo perché, utilizzando un indirizzo fisico basso per caricare il kernel in memoria, abbiamo la garanzia di avere sufficiente spazio (potremmo non avere 2 GB di memoria fisica). Per arrivare ad eseguirlo a 2GB nella memoria virtuale dobbiamo mapparlo due volte, poiché mappandolo solo ad indirizzi alti potremmo avere errori quali tentativi di eseguire il codice in aree non mappate, e non ci sarebbe nemmeno un kernel per gestire trap di questo genere.

Dobbiamo quindi prima caricarlo ed eseguirlo nella pagina 0, effettuando correttamente l'inizializzazione della tabella delle pagine, per passare poi in un secondo momento alla parte più alta della memoria virtuale.

A regime ogni processo utilizzerà per sé (user mode) i 2 GB più bassi, mentre quelli più alti saranno utilizzati per il kernel.

La entry.S prepara inoltre uno stack; infine salta al main scritto in C.

Prima tabella delle pagine utilizzata

La prima tabella utilizzata usa pagine da 4MB. Per indirizzare 4 MB servono 22 bit di indirizzamento, che saranno riservati all'offset, mentre i restanti 10 indicheranno il numero della pagina.

All'interno del processore è presente un registro chiamato CR3 che punta all'indirizzo base della tabella delle pagine e ci permette di saltare nella memoria alta. Appena saltiamo alla memoria alta possiamo chiamare il main.

Main

Il main esegue una serie di inizializzazioni: l'allocatore delle pagine, la tabella delle pagine del kernel, altri processori... Poi salta a mpmain(), che effettua altre inizializzazioni, e infine passa allo scheduler(). Lo scheduler() è quella parte di sistema operativo che si occupa di mandare in esecuzione i processi.

Il kernel inizialmente ha bisogno di allocare spazio, per esempio della tabella delle pagine. Non può chiamare malloc(), perché si tratta di una funzione della libc che si appoggia sul kernel. Per allocare spazio, il kernel ha un allocatore che come unità di misura usa le pagine. Nella parte alta della memoria virtuale, all'interno della pagina da 4MB che abbiamo mappato che contiene il kernel, sopra il kernel sono presenti tante pagine da 4KB utilizzabili.

Per ogni pagina controlla: che l'indirizzo sia un multiplo della dimensione di pagina, che la pagina non sia sovrapposta al kernel, che l'indirizzo non sia sopra la fine della memoria fisica. Se questi vincoli sono rispettati, la pagina viene riempita di 1 e viene inserita in una lista delle pagine libere.

Quando all'allocatore del kernel viene richiesta una pagina, viene ricercata all'interno della lista delle pagine libere ed eventualmente restituita.

Dopo aver riempito la lista delle pagine libere viene preparata la tabella delle pagine del kernel: viene effettuato il setup della kernel virtual memory, che deve allocare lo spazio per la tabella delle pagine. La memoria viene così organizzata: il registro CR3 punta alla page directory (quella che stiamo allocando adesso). Ogni entry della page directory può puntare ad una tabella delle pagine, in cui ogni entry punta effettivamente ad un frame.

La traduzione viene così effettuata: vengono analizzati i primi 10 bit dell'indirizzo virtuale e viene ricercata le entry corrispondente nella page directory. Se viene trovata, otteniamo l'indirizzo fisico della page table. A questo punto vengono analizzati i 10 bit successivi che corrispondono ad una entry della page table. Se troviamo la pagina, otteniamo 20 bit dell'indirizzo del frame, a cui vengono aggiunti gli ultimi 12 dell'indirizzo (offset). In questo modo otteniamo l'indirizzo fisico corrispondente.

Per realizzare questo meccanismo bisogna quindi allocare una pagina per la page directory, e per il range di indirizzi che vogliamo allocare, andranno allocate tante page table. Questi range specificati tramite inizio e fine in indirizzi fisici, vengono mappati all'indirizzo virtuale specificato, con una serie di permessi da associare alla pagina.

All'indirizzo virtuale 2GB (KERNBASE) mappiamo il primo MB in lettura e scrittura. Da dove inizia il kernel (1 MB) a dove finisce, mappiamo in sola lettura. Da dove finiscono i dati fino in cima alla RAM mappiamo in lettura e scrittura. Infine in cima è presente una piccola parte riservata ai device accessibile in sola lettura.

Indipendentemente dai processi utente che stanno girando, questa parte resterà sempre invariata. Quando si passa da un processo all'altro si va a caricare una nuova tabella delle pagine dove le entry per la parte alta saranno sempre le stesse, mentre cambieranno le entry per la parte bassa.

Scheduling

mercoledì 25 novembre 2020 22:53

Limited Direct Execution

Come viene virtualizzata la CPU?

Per virtualizzare la CPU si utilizza il time sharing; ovvero, ogni processo riceve l'uso (esclusivo) della CPU per un po', poi si passa a un altro processo e così via.

I problemi di questo meccanismo sono:

- Efficienza: come contenere i costi nel passaggio tra un processo e l'altro? Per togliere la CPU a un processo e darla ad un altro è necessario salvare tutti i valori dei registri nello stack, caricare i valori dei registri del nuovo processo e farlo ripartire nelle condizioni in cui era stato fermato;
- Controllo: come garantire che il processo rilasci l'uso della CPU? Se il processo fa una system call viene avviata la modalità kernel, e il kernel può decidere cosa eseguire. E se invece l'esecuzione rimanesse in modalità utente in un loop?

Il metodo per risolvere questo è far sì che venga utilizzato un timer, che a intervalli periodici invia degli interrupt. In questo modo viene avviata periodicamente la modalità kernel, e il kernel può decidere se continuare ad eseguire il processo in corso o eseguirne un altro.

Meccanismi per lo scambio di processi

Meccanismo cooperativo

Il sistema operativo si fida dei processi, che si impegnano ad eseguire ogni tanto una system call, anche se non ne hanno bisogno. Per questi processi esiste una system call apposita, chiamata yield, che se necessario rilascia l'uso della cpu.

Questo approccio tuttavia non è molto affidabile: i processi potrebbero non eseguire la system call, oppure potrebbero entrare in loop e non riuscire ad eseguirla.

Meccanismo non cooperativo

Meccanismo utilizzato da tutti i sistemi moderni. Prevede che il sistema operativo si riprenda il controllo a forza tramite un timer interrupt.

Per esempio in Xv6 in kernel/trap.c troviamo

```
void trap(struct trapframe *tf)
{
    /* ... */
    if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 +
    IRQ_TIMER)
        yield();
}
```

che viene chiamata da alltraps (definita in kernel/trapasm.S).

Questo pezzo di codice fa la seguente valutazione: se c'è un processo corrente, che sta girando, ed è arrivato un interrupt di tipo timer, allora fai yield (dai la possibilità ad un altro processo di andare in esecuzione).

Salvare e ripristinare il contesto

Quando il SO riprende il controllo, deve decidere se far continuare l'esecuzione del processo corrente o meno. La parte di kernel che prende questa decisione è lo scheduler (ci sono diverse politiche diverse di scheduling, che dipendono dal sistema o dallo stato attuale).

Per cambiare processo, si fa un cambio di contesto (context switch); l'idea è semplice:

1. Si salvano i registri del processo che si sta eseguendo
2. Si caricano i registri del processo che vogliamo mandare in esecuzione
 - a. nota: questo include anche, per esempio, il registro che "punta" alla tabella delle pagine

L'implementazione di questo meccanismo è un po' meno semplice, perché deve tenere conto di tutti i dettagli; la funzione che implementa il context-switch è swtch.

Ottimizzazione del contesto

Quando swtch verrà chiamata, i registri utente saranno già stati salvati nel trap-frame (Struct C che contiene tutti i valori dei registri utente. È contenuto nello stack kernel) del processo corrispondente, quindi basta scambiare i registri che:

- non vengono automaticamente gestiti dal compilatore in seguito a una chiamata di funzione (EAX, ECX ed EDX);
- non sono già salvati altrove; per esempio il PCB (Process Control Block) contiene:
 - pde_t *pgdir; (indirizzo fisico della tabella delle pagine di quel processo)
 - char *kstack; (indirizzo dello stack kernel di quel processo)
- cambiano da un processo all'altro. Per esempio, i registri di segmento sono uguali all'interno del kernel.

Fatte queste considerazioni il contesto si riduce a:

```
struct context {  
    /* definita in kernel/proc.h */  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip; /* Instruction pointer-inserito "automaticamente" da CALL */  
}; /* esp usato per puntare a questa struttura nello stack */
```

Nel PCB (area dove vengono salvate le informazioni relative ad un processo) abbiamo:

```
struct proc {  
    //...  
    struct trapframe *tf;  
    struct context *context; // valore caricato in esp da swtch  
    //...  
};
```

Per ripristinare un contesto esp viene fatto puntare a quel contesto: verrà fatta la pop e verranno ripristinati tutti i registri che contiene e infine si tornerà all'istruzione puntata da eip.

Swtch

Prende due parametri:

- Un puntatore ad un puntatore a contesto, dove verrà salvato il contesto precedente
- Il puntatore al nuovo contesto

Quando viene chiamata una funzione sullo stack gli argomenti vengono aggiunti da destra a sinistra (il primo valore nello stack sarà il secondo argomento). Dopo gli argomenti viene salvato l'eip (instruction pointer) del processo che ha effettuato la chiamata di funzione.

Dopodiché inizia l'esecuzione della funzione:

- eax contiene il valore dello stack pointer+4 (primo argomento)
- edx contiene il valore dello stack pointer+8 (secondo argomento)
- Avviene il salvataggio (push) dei valori contenuti nei seguenti registri:
 - ebp
 - ebx
 - esi
 - edi
- Avviene lo scambio:
 - Il valore di esp (stack pointer) viene messo dove punta eax (contesto precedente)
 - Dentro esp viene caricato edx (nuovo contesto)In questo modo lo stack diventa lo stack del nuovo processo
- Vengono effettuate tutte le pop dei registri del nuovo contesto, per toglierli dallo stack e renderli disponibili per l'esecuzione del processo.

La funzione swtch viene chiamata sul processo A, ma il return avviene sul processo B. In un altro momento, un altro processo chiamerà la swtch e il return verrà effettuato su A.

System call

mercoledì 25 novembre 2020 22:00

Le system call funzionano esattamente come gli interrupt. Su Xv6 l'istruzione che ci permette di fare delle system call è proprio int 64.

Esistono 256 diversi vettori di interrupt e ognuno di loro potrebbe avere un handler diverso.

Quando arriva un interrupt bisogna interrompere quello che stavamo facendo, rispondere alla interrupt in modalità kernel, tornare alla modalità precedente (che può essere sia utente che kernel) e riprendere poi l'esecuzione da dove eravamo rimasti.

Per fare questo abbiamo bisogno di salvare i valori dei registri (in particolare l'instruction register, che indica l'indirizzo della prossima istruzione da eseguire, a cui vogliamo tornare) all'interno dello stack. Tuttavia non possiamo fidarci dello stack del processo, perché lo stack pointer potrebbe non essere ad un valore valido oppure potremmo rischiare di esporre all'utente informazioni che non deve vedere. Per questo motivo ogni processo ha due stack diversi:

- Uno che viene utilizzato in modalità utente (registro esp)
- Uno che si attiva quando si passa alla modalità kernel, salvato in un registro speciale.

Se l'interruzione arriva in modalità utente, la CPU passa alla modalità kernel e di conseguenza anche allo stack kernel, salvando il valore dello stack utente sullo stack kernel. Poi vengono salvati i flag e l'instruction pointer. Altri registri, se vengono modificati dall'interrupt handler devono prima essere salvati sullo stack da quest'ultimo ed essere ripristinati prima di restituire il controllo.

La CPU tramite i flag sa se è avvenuto un cambio di privilegio e se, prima di ritornare alla modalità precedente, è necessario ripristinare lo stack utente.

Qualche interrupt ha un codice di errore associato, qualche altro no. Questo significa che può essere presente sullo stack oppure no. Xv6 corregge questo comportamento per uniformare la gestione, facendolo sempre trovare nello stack.

Vector.S

Il file kernel/vector.S contiene tutti gli handler che rispondono agli interrupt. Viene generato tramite uno script in Pearl. Tutti gli handler hanno la stessa forma: il primo numero di cui viene fatta la push sullo stack è l'error code, il secondo è il numero di interrupt. Se l'interrupt possiede un error code viene fatta solo la push del numero di interrupt.

Dopo aver effettuato le push, salta ad alltraps, definito all'interno di kernel/trapasm.S

Trapasm.S

Questo codice salva una serie di valori di registri di uso generale nello stack tramite push(). In questo modo al termine possiamo invertire il processo risistemando tutti i registri tramite pop(). Al termine del ripristino vengono ignorati tramite una add il numero di interrupt e l'error code precedentemente inseriti nello stack(), infine si esce dalla interrupt tramite l'istruzione iret.

La struttura dati che viene costruita in questo modo sullo stack viene chiamata trap-frame. La struct trapframe viene definita in kernel/x86.h. Contiene il numero di trap, l'error code, e una copia di tutti i registri utente (registri di uso generale, registri di segmento, ciò che c'era sullo stack).

Siccome lo stack cresce ad indirizzi decrescenti e le strutture in C crescono ad indirizzi crescenti, le prime informazioni che troviamo nella struttura sono le ultime salvate nello stack.

Stati di un processo

Alcune system call possono restituire subito il valore di ritorno (ad esempio getpid()). Altre system call richiedono invece un sacco di tempo (ad esempio una read() di un settore del disco non ancora caricato in RAM). Per tutto questo tempo il processo è bloccato, e non può proseguire con l'esecuzione.

In questi casi può avvenire un context switch: il processo che ha fatto la read() viene messo in una coda di attesa, e viene mandato in esecuzione un altro processo. Ciò significa che entriamo nell'interrupt salvando la fotografia di un processo, e usciamo dall'interrupt ripristinando la fotografia di un altro.

Questo è il principio alla base della condivisione della CPU tra processi diversi.

Ogni processo durante la sua vita può essere in stati diversi. Tre stati minimali sono:

- Ready (pronto ad essere eseguito)
- Running (sta girando)
- Blocked, waiting, sleeping (sta aspettando qualcosa, non se ne fa nulla della CPU).

I processi possono partire da una fase di inizializzazione chiamata init o embrion. Dopo questa fase vanno in ready, ossia in attesa della CPU. Lo scheduler, una parte del sistema operativo, decide chi tra i processi running e i processi ready deve utilizzare la CPU.

Quando un processo running viene deschedulato significa che viene salvata una fotografia del suo stato di esecuzione e viene mandato in esecuzione di un altro processo. Questo meccanismo dà l'illusione all'utente che i processi vengano eseguiti in parallelo.

Bisogna tenere conto che l'operazione di (de)schedulazione ha un costo. Ha senso farla 100/1000 volte al secondo, ma non di più, poiché comporterebbe costi troppo elevati.