# Sistemi 4

Vedremo le system call per gestire file e processi. Un processo non può parlare direttamente con l’hardware, il modo che ha per fare una richiesta al sistema operativo è mediante le system call, che fanno passare la CPU in modalità privilegiata e fa proseguire l’esecuzione all’interno del kernel.

Per un programmatore C una chiamata di sistema non è diversa da una di funzione, perché la libreria C per ogni system call possiede una funzione wrapper (visto che per fare una system call diretta, bisognerebbe scriverla in assembly). Alcuni esempi sono la open(), la creat(), la openat(), ecc.

Le funzioni wrapper preparano gli argomenti per le system call, e, poiché ad ogni system call è associato un numero corrispondente (codice di software interrupt). La trap viene eseguita dalla wrapper (che equivale alla chiamata di INT 0xNumero in base 16 corrispondente alla system call). Il kernel controlla la validità degli argomenti, esegue la richiesta e scrive il risultato in un registro, poi ritorna alla modalità utente tramite l’istruzione IRET. La funzione wrapper va a controllare il risultato: se ci sono stati errori imposta a errno il codice di errore corrispondente (per il momento facciamo finta che sia una variabile globale) e ritorna -1, altrimenti, restituisce il risultato al chiamante.

Una syscall è molto più costosa di una semplice chiamata a funzione.

Per cercare di ottimizzare questo costo, alcune librerie bufferizzano gli input (per esempio printf bufferizza gli output e se si fanno tante printf piccole in realtà queste verranno bufferizzate e verrà chiamata una sola syscall sull’intero buffer di output).

Col comando strace -e si vede quante volte un programma chiama una syscall.   
Si può usare alternativamente un debugger.

Quando noi chiamiamo una syscall, bisogna sempre controllare se il valore di ritorno è -1: in tal caso bisogna guardare il valore di errno (errno NON va controllata se la syscall va a buon fine, perché non è detto che errno venga azzerata).

Perror(num) permette di stampare l’errore corrispondente (strerror(num) converte il numero dell’errore in stringa)).

I Processi sono identificati dal PID. UNIX diversi usano interi di dimensione diversa per rappresentare il PID (può essere di 32, 64, … bit). Questo ci causa problemi sulla portabilità del codice. Per scrivere codice di questo tipo ci si appoggia a dei typedef: in particolare per i PID si utilizza il tipo pid\_t. Questo funziona bene finché non si deve stampare il pid, perché in C alla printf bisogna dire la dimensione di quello che stampiamo. Un approccio standard è di castare pid\_t a un tipo di intero molto grande (come long long): il cast migliore è quello a intmax\_t (o uintmax\_t) stampandolo con %jd (o %ju).

Vediamo il debugger. Per vedere dove un programma si schianta basta lanciare un debugger prima di avviare un programma. Se quando si compila il programma si usano opzioni come -ggdb si aggiungono al compilato le informazioni di debugging, così quando si lancia il debugger possiamo sapere esattamente a quale linea di codice il programma è esploso. Altrimenti si può usare gdb -nx ./a.out che è molto più limitata ma fa il suo lavoro.

Il debugger permette di aggiungere dei break point (che fermano il codice e permettono di leggerlo), così quando si fa il run il programma si ferma lì e diventa possibile controllare il valore delle variabili locali in quel dato momento. Quando il programma è fermo, con “n” si esegue l’esecuzione del comando successivo (poi ripremendo invio si ripete l’ultima istruzione). Con “s” si fa la stessa cosa ma fa anche l’esecuzione passo passo all’interno delle funzioni (mentre con n se c’è una chiamata di funzione prima esegue tutta la funzione e poi ci ridà il controllo). S = step in, N = Step over.

È possibile inserire break points che si attivano solo con determinate condizioni. È inoltre possibile mettere un break point sull’esecuzione, ma anche sulle modifiche di un indirizzo di memoria.  
Ci sono debugger anche negli IDE.

IO su File.

Tutto l’IO avviene tramite file descriptors. I fd sono interi non negativi che identificano un file aperto. “file” è un termine abbastanza generale, non si intendono solo file regolari, ma anche connessioni di rete, pipe, dispositivi (in UNIX quasi tutto è visto come un file). Ogni processo ha i propri file descriptor.

Tre fd speciali sono (per convenzione):

0 = STDIN\_FILENO <-> stdin/cin

1 = STDOUT\_FILENO <-> stdout/cout

2 = STDERROR\_FILENO <-> stderr/cerr

Le costanti sono definite in unistd.h

Per aprire un file, basta usare la syscall open(), che prende o 2 o 3 argomenti: questo fa un po’ schifo perché rende possibile passare il numero sbagliato di argomenti. Quando non creiamo un file ne vuole 2, se lo creiamo ne vuole 3. Gli argomenti che vuole sono il pathname del file come sequenza di caratteri e i flags in or bit a bit sul modo in cui vogliamo aprire il file. I flag si possono recuperare e modificare con fcntl. La maggior parte dei flag riguarda il file:  
O\_APPEND, O\_NONBLOCK, …

Ma esistono (per ora uno: FD\_CLOEXEC) flag associati al fd. CLOEXEC vuol dire “chiudi in caso di exec” (exec è un’altra syscall)

C’è un terzo argomento, mode, che viene usato solo quando viene creato un file, tale argomento è un altro flags, che contiene O\_CREAT o O\_TMPFILE. Possono essere compresi i permessi di lettura, scrittura e esecuzione, che possono essere specificati in ottale (nota: tali permessi varranno solo dall’apertura successiva, se il fd è aperto lo si può usare come si vuole).

I permessi di lettura, scrittura e esecuzione dei file sono visibili in gruppi di 3 bit che indicano i permessi per il proprietario, i permessi per gli appartenenti al gruppo e i permessi per gli altri (per vederli si usa ll filepath).

C’è un utente speciale, che è root, che può eseguire (a patto che il file sia eseguibile) e leggere e scrivere qualsiasi file.

Esercizio per casa: Scrivere un programma che crea un file chiamato pippo che sia leggibile solo dal proprietario (non scrivibile da nessuno). Cosa succede se il file esiste già? La syscall va a buon fine? Il contenuto del file esistente viene preservato?

Una volta che abbiamo il file descriptor possiamo usarlo per leggere, scrivere, o chiuderlo. La funzione di lettura (read) prende come argomenti il fd, un puntatore a un buffer e quanto vogliamo leggere (in genere la dimensione del buffer). La read restituisce quanti byte ha letto veramente. La write fa l’esatto opposto della read, la close chiude il file descriptor. La close può fallire, se il file è già chiuso (per esempio). In ogni caso, il fd viene rilasciato.  
  
Altro esercizio per casa: scrivere un “cat dei poveri”, ovvero un programma che:

se invocato senza parametri legge da stdin e butta su stdout. Altrimenti, legge dai file specificati / scrive su di essi.

A ogni file aperto è associato un file offset/”cursore” che indica in che posizione leggere / scrivere. Appena aperto il file, questo offset è 0. Dopo ogni operazione di lettura/scrittura, questo offset è spostato della quantità di byte che sono stati letti o scritti (è usato per i file regolari, non ha senso andare avanti e indietro nei socket, dispositivi ecc.).

Oltre che da read e write, si può spostare questo cursore tramite lseek.

Off\_t lseek(int fd, off\_t offset, int whence); //Dove whence può essere: SEEK\_SET, SEEK\_CUR o SEEK\_END.

Il cursore può essere spostato oltre la sua fine.

I metadati di un file (quando è stato visualizzato l’ultima colta, chi è il proprietario, ecc.) si usa:

int stat();

oppure lstat (che è uguale ma non segue i link simbolici)

fstat(); che usa direttamente un file descriptor anziché un path.

Ogni tipo di filesystem ha dei limiti, per cui la lunghezza massima di un nome file può variare a seconda della directory considerata (per controllare tale limite si può usare getconf NAME\_MAX /directory).

Esistono delle costanti \_POSIX\_ come \_POSIX\_NAME\_MAX che corrispondo alle misure MINIME (per esempio \_POSIX\_NAME\_MAX è 14, che non ha senso come dimensione massima per il nome di un file).