Corso di Laboratorio di Sistemi Operativi

Lezione 5

Alessandro Dal Palù

email: alessandro.dalpalu@unipr.it

web: www.unipr.it/~dalpalu

Processi in Unix Approfondimenti:

`http://gapil.gnulinux.it/download/`

Processi in Unix

- In Unix i processi vengono creati dalla syscall `fork()`; il processo figlio è una copia identica del processo padre, ma ha un nuovo pid e viene eseguito in maniera indipendente.

- Il padre può attendere la fine del processo figlio con `wait()` o `waitpid()`

- Usi tipici di `fork()`:

- unico programma con 2 sezioni, una per il padre, l'altra per i figli (tipica del client server)

- il figlio esegue un altro programma mediante le syscall della famiglia `exec()` (tipica della shell)

- Il processo figlio condivide con il padre la lista dei file descriptors.

Syscalls in Unix

- `getenv()`, `getpid()`, `getppid()`, `getuid()`, `getcwd()`
  Informazioni sull'ambiente.

- `putenv()`, `setuid()`, `setgid()`, `chdir()`
  Modificare ambiente del processo.

- `fork()`, `exit()`
  Creazione/terminazione nuovo processo.

- `execve()`, `execl()`, `execlp()`, `execle()`, `execv()`, `excvp()`
  Sostituire il programma in esecuzione con un altro.

- `wait()`, `waitpid()`
  Sospendere esecuzione fino alla terminazione del figlio/processo.

Esempio padre figlio

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int pid,retv;
  switch(pid=fork()){
  case -1: printf("Errore in creazione figlio\n");
           exit(-1);
  case 0 : sleep(2); // figlio
           exit(3);
  default: wait(&retv); // padre
    printf("Figlio ha terminato con exit status %d\n",WEXITSTATUS(retv));
  }
 }
```

Esercizio

- Scaricare il file esempiofork.c
- Modificare il codice, aggiungendo la stampa del proprio pid e ppid (usando `getpid` e `getppid`)
- Compilare con `make esempiofork` (osservare che make possiede delle regole implicite che compilano i files .c etc)
- NON scrivete la documentazione in latex
  (vediamo poi come farla in doxygen)
- Tempo 20 minuti

Doxygen

- Tool per la generazione automatica di documentazione di software (C, Java, ...)
- Documentazione completa `www.doxygen.org`
- Utilizza i commenti inseriti nei files sorgente per impaginare una documentazione.
- Utilizza un file di configurazione `Doxyfile`, in cui si può scegliere il formato di output (html, man, latex, rtf...)

Esercizio

- Scaricare il file `Doxyfile` nella stessa directory di `esempiofork.c` e lanciare `doxygen`
- Entrare nella sottodirectory `doc` creata e osservare (`ls`) i vari output generati
- Copiare i files html (tutta la cartella) in una sottocartella di `~/html/` e creare un link dalla directory della lezione
- Entrare nella sottodir `latex` e lanciare `make`, che compila il manuale in PDF. Analizzare il Makefile generato automaticamente da `doxygen`
- Analizzare il file `Doxyfile` e identificare i punti da toccare per: cambiare nome al progetto, cambiare directory di output, selezionare i tipi di files in output.
- Tempo 30 minuti

Eseguire programmi

- Per eseguire un programma usare la famiglia `exec`.
- Per maggiori informazioni: `man execl` o su gapil.
- Esempio: `execl("/bin/ls","ls","-l",0)`
- `execl` prende il path, seguito dagli argomenti (il primo è il comando vero e proprio). L'ultimo argomento DEVE essere 0, per segnalare il termine della lista.
- Esercizio: includere nel file `esempiofork.c` il comando exec

I files - Accesso bufferizzato

- Linux possiede 2 interfacce per l'accesso ai file: bufferizzato e non bufferizzato.

- Interfaccia standard ANSI C (che avrete già usato).

- E' bufferizzata (da `glibc`) e si basa sul concetto di stream (`FILE *`).

- L'interfaccia è definita in `stdio.h`

- fornisce un alto livello di astrazione e di indipendenza dal sistema operativo (infatti può essere compilata su qualunque sistema).

I files - Accesso non bufferizzato

- Interfaccia standard di Unix.
- E' una interfaccia non bufferizzata basata sui file descriptor (numeri interi).
- L'interfaccia è definita nell'header `unistd.h`.
- E' una interfaccia di piu' basso livello, strettamente legata alle syscalls di Unix, ed e' quindi adatta per la programmazione di sistema.

I file descriptor e i file standard

- I file descriptor sono interi (indici di un vettore) che vengono assegnati progressivamente a partire da 0 ai nuovi file aperti di un processo.

- Ogni nuovo processo viene lanciato con almeno tre file aperti.

- fd0: **standard input**. E' sempre associato al file da cui ci si aspetta di ricevere i dati di ingresso. Nel caso si una shell e' la tastiera del teminale.

- fd1: **standard output**. E' sempre associato al file da cui ci si aspetta debbano essere inviati i dati di uscita. Nel caso si una shell e' il video del teminale.

- fd2: **standard error**. E' sempre associato al file da cui ci si aspetta debbano essere inviati i messaggi di errore. Nel caso si una shell e' il video del teminale.

- Per questi 3 interi in unistd.h sono definite 3 costanti simboliche: STDIN_FILENO STDOUT_FILENO e STDERR_FILENO

Operazioni su files

- Apertura di un file: fd=open(nomefile, flags) (vedere `man 2 open`)
- Esempi: `fd=open(filename, O_RDONLY)`
- `fd=open(filename,O_CREAT|O_WRONLY|O_TRUNC,S_IRWXU|S_IRGRP|S_IROTH)`
- Chiusura di un file `close(fd)`
- Quando un processo termina vengono chiusi tutti i suoi fd. Se non rimangono altri fd sul file aperto viene chiuso anche il file.

Operazioni su files

- Lettura di un file: `n=read(int fd, char* buffer, int num_byte);`
- La funzione ritorna il numero di byte letti oppure -1 in caso di errore
- Scrittura di un file: `n=write(int fd, char* buffer, int num_byte);`
- La funzione ritorna il numero di byte scritti oppure -1 in caso di errore

Esercizi

- Approfondire: `http://www.linuxdidattica.org/docs/prg_C/cgiprg15.html`
- Provare i due esempi contenuti creando i files `scrivi.c` e `leggi.c`
- Tempo 20 minuti

Operazioni su files

- Modifica della posizione corrente di un file:
  `n=lseek(fd, offset, whence);`

- Valori possibili di `whence`: `SEEK_SET SEEK_CUR SEEK_END`

- La funzione ritorna la nuova posizione.

- Per conoscere la posizione corrente di un file: `lseek(fd,0,SEEK_CUR);`

- Per riportare all'inizio del file la posizione corrente:
  `lseek(fd,0, SEEK_SET);`

- Approfondire con `man 2 lseek`

Esercizio

- Esericzio: modificare il file `leggi.c`.
- Leggere e stampare a video gruppi di 2 caratteri saltando i successivi 3 (usare `lseek`)
- Esempio: file contiene `abcdefghiljkl`
- Output: `abfgjk`
- Tempo 20 minuti

Condivisione files

- Forzare lo scarico dei buffer del kernel: `sync()`
- Forzare lo scarico dei buffer di un singolo file: `fsync(fd)`
- Duplicazione di un FD su un nuovo FD: `newfd=dup(oldfd)`
- Duplicazione si un FD su un FD esistente: `dup2(oldfd,newfd)`
- Operazioni ausiliarie su un fd: `fcntl(fd,comando)`
- Operazioni su file speciali: `ioctl(fd,request)` (Esegue operazioni specifiche dell'hardware)
  - Cambiamento dei font di un terminale
  - Esecuzione di una traccia audio di un CDROM
  - Impostazione della velocità di trasmmissione di una linea seriale
  - Espulsione di un dispositivo removibile

Una shell minimale - esercizio (i)

- Scaricare il file `mysh-0.1.tar.gz` dal sito del corso
- Estrarre il contenuto in una nuova directory
- Osservare come è scritto il makefile
- Studiare il codice nelle sue parti
- Espandere la shell con il comando `which`, che restituisce il path completo di un comando immesso da prompt (se esiste).
- Prototipo funzione: `which(char *cmd, char *cmdfq)`
  `cmd` è il nome del comando
  `cmdfq` è il nome fully qualified, comprensivo del path
- Utilizzare la variabile `path` per ricavare le directory che potrebbero contenere il comando (`cmd[0]`)
- Utilizzare la funzione `access()` (`man 2 access`) per testare l'esistenza del percorso + nome comando

Una shell minimale - esercizio (ii)

- Commentare il codice per doxygen, produrre una nuova versione 0.2 della shell e la relativa documentazione.

- Estendere la shell e permettere l'esecuzione di comandi, utilizzando la funzione `which`, `fork`, `wait` ed `exec`.