Lezione 4 - 29/3/21 (System calls, Fork)

Architettura

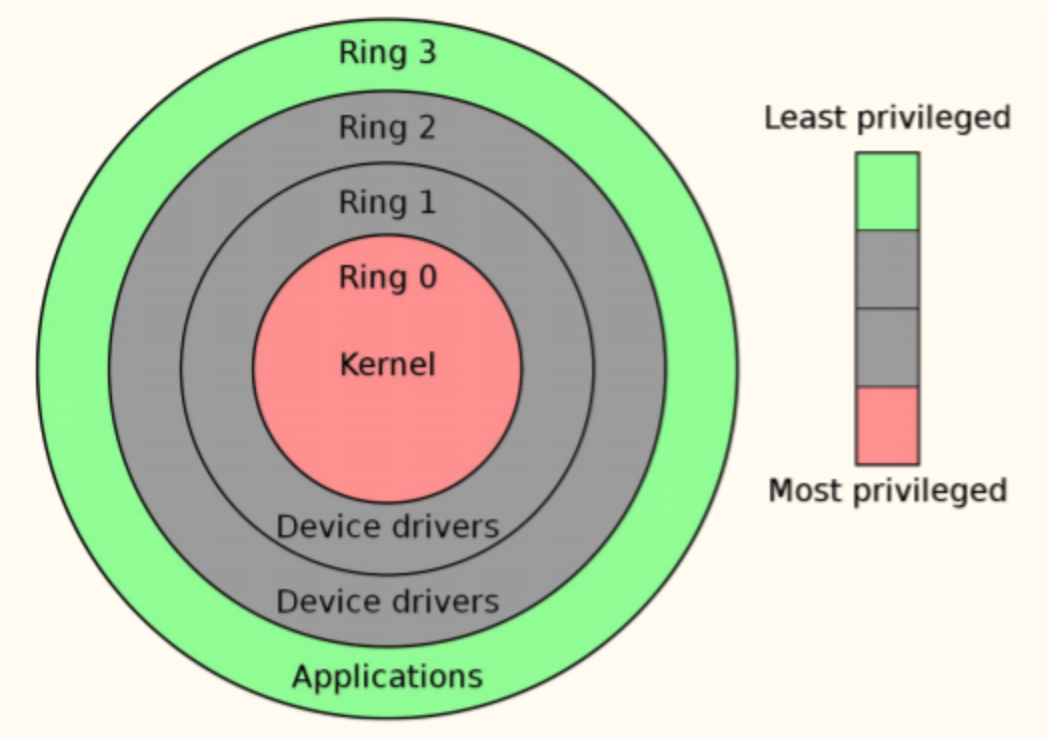
Kernel Linux

Il kernel è il cuore di un sistema Unix ed è incaricato di gestire le risorse essenziali (CPU, memoria, periferiche, ...).

Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

- **User Space:** ambiente in cui vengono eseguiti i programmi
- **Kernel Space:** ambiente in cui viene eseguito il kernel



Stile Windows non Linux

System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls.** Letteralmente "chiamate al sistema" che il kernel esegue nel **kernel space,** restituendo i risultati al programma chiamante nello user space.

Le chiamate restituiscono "-1" in caso di errore e settano la variabile globale **errno.** Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

ldd → Se seguito da un "file.out" mostra tutte le librerie incluse in particolare:

- **ld-linux.so:** è una libreria che contiene le istruzioni su come eseguire il programma main.out.
- **libc.so:** contiene tutte le librerie di C.

System calls utili:

TIME

- **time_t time(timt_t *second)** → Restituisce un intero (time_t) e accetta un puntatore a un "time_t"
- **open("nomeFile.txt", 0_CREAT|0_RDWR, S_IRUSR|S_IWUSR)open("nomeFile.txt", 0_CREAT|0_RDWR, S_IRUSR|S_IWUSR)char * ctime(const time_t *timeSeconds)** → Restituisce una stringa

```c
#include <time.h>                                          //time.c
#include <stdio.h>

void main(){
    time_t whatTime = time(NULL); //seconds since 1/1/1970
    //Print date in Www Mmm dd hh:mm:ss yyyy
    printf("Current time = %s", ctime(&whatTime));
}
```

CARTELLE

- **int chdir(const char *path)** → cambia la cartella corrente
- **char* getcwd(char *buf, size_t sizeBuf)** → prende la cartella corrente

```
#include <unistd.h>                                    //chdir.c
#include <stdio.h>

void main(){
    char s[100];
    chdir(".."); //Change working dir
    printf("%s\n", getcwd(s,100)); //Print current working dir
}
```

OPERAZIONI CON I FILE

- **int open(const char *pathname, int flags, mode_t mode)** → apre un file.
  - **flags:** O_RDONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC
- **int close(int fd)** → chiude un file.
- **ssize_t read(int fd, void *buf, size_t count)** → legge dei caratteri da un file.
- **ssize_t write(int fd, const void *buf, size_t count)** → scrive dei caratteri da un file.
- **off_t lseek(int fd, off_t offset, int whence)** → sposta il punto di lettura di un file (testina).

DUPLICAZIONE FILE DESCRIPTORS: dup() e dup2()

- **int dup(int oldfd)** →
- **int dup2(int oldfd, int newfd)** →

PERMESSI chmod() e chown()

- **int chown(const char *pathname, uid_t owner, gid_t group)** → Permette di modificare il proprietario di un file specificandone il pathname.
- **int fchown(int fd, uid_t owner, gid_t group)** → Permette di modificare il proprietario di un file specificandone il file descriptor.

- **int chmod(const char *pathnamme, mode_t mode)** → Permette di modificare i permessi di un file specificandone il pathname.
- **int fchmod(int fd, mode_t mode)** → Permette di modificare i permessi di un file specificandone il file descriptor.

```c
#include <fcntl.h>            //execute with sudo!    //chown.c
#include <unistd.h>
#include <sys/stat.h>
void main(){
    int fd = open("file",O_RDONLY);
    fchown(fd, 0, 0); // Change owner to root:root
    chmod("file",S_IRUSR|S_IRGRP|S_IROTH); // Permission to r/r/r
}
```

Permessi (visualizzati con ls -l): Utente|gruppo|other

Eseguire programmi: exec()

- int exec**v**(const char *path, char *const argv[])
- int exec**vp**(const char *file, char *const argv[])
- int exec**vpe**(const char *file, char *const argv[], char *const envp[])
- int exec**l**(const char *path, const char * arg0, ..., argn, NULL)
- int exec**lp**(const char *file, const char * arg0, ..., argn, NULL, char *const envp[])
- int **execve**(const char *filename. char *const argv[], char *cosnt envp[]) → Sostiuisci il processo attuale con quello descritto da "filename" passandogli i parametri "argv" e le variabili d'ambiente "envp"

- **v/l:** Accetta solo percorsi assoluti dei file.
- **vp/lp:** Accetta anche percorsi non assoluti quindi posso mettere solo il nome del file.
- **vpe/lpe:** Passa anche le variabili d'ambiente. (N.B. la lista deve essere terminata con un elemento nullo).

EXECV

```c
#include <unistd.h>                                          //execv1.out
#include <stdio.h>
void main(){
    char * argv[] = {"par1","par2",NULL};
    execv("./execv2.out",argv); //Replace current process
    printf("This is execv1\n");
}
```

Non verrà eseguita l'ultima riga (il printf) perché sostituisce il processo attuale con "execv2.out" appena prima se fallisce la sostituzione allora la esegue.

```c
#include <stdio.h>                                           //execv2.out
void main(int argc, char ** argv){
    printf("This is execv2 with %s and %s\n",argv[0],argv[1]);
}
```

EXECLE

```c
#include <unistd.h>                                         //execle1.out
#include <stdio.h>
void main(){
    char * env[] = {"CIAO=hello world",NULL};
    execle("./execle2.out","par1","par2",NULL,env); //Replace proc.
    printf("This is execle1\n");
}
```

```c
#include <stdio.h>                                          //execle2.out
#include <stdlib.h>
void main(int argc, char ** argv){
    printf("This is execv2 with par: %s and %s. CIAO =
            %s\n",argv[0],argv[1],getenv("CIAO"));
}
```

Chiamare la shell: system()

- int system(const char * string)

```c
#include <stdlib.h>                                          //system.c
#include <stdio.h>

void main(){
    // '/bin/sh -c string'
    int outcome = system("echo ciao"); // execute command in shell
    printf("%d\n",outcome);
    outcome = system("if [[ $PWD < \"ciao\" ]]; then echo min; fi");
    printf("%d\n",outcome);
}
```

N.B. attenzione system esegue i comandi su shell e non su bash quindi non riconosce comandi come "[[". Quindi o eseguo bash e do i comandi o uso shell ma ho meno comandi

FORK

- E' una system call
- Serve a creare un processo secondario

Identificativi dei processi

- PID - Process ID
- PPID - Parent Process ID
- SID - Session ID
- PGID - Progress Group ID
- UID/RUID - (Real) User ID
- EUID - Effective User ID

fork: getpid(), getppid()

- **pid_t getpid()** → restituisce il PID del processo attivo.
- **pid_t getppid()** → restituisce il PID del processo padre.

fork: wait(), waitpid()

- **pid_t wait(int *status)** → attende al conclusione di un figlio e ne restituisce il PID riportando lo status nel puntatore passato come argomento se non NULL.
- **pid_t waitpid(pid_t oud, int *status, int options)** → analoga a wait ma consente di passare delle opzioni e si può specificare come pid:
  - **-n** → <-1: attende tutti i figli il cui "gruppo" è |pid|
  - **-1** → attende un figlio qualunque
  - **0** → attende tutti i figli con lo stesso "gruppo" del padre
  - **n** → n>0 attende il figlio in cui pid è esattamente n

NOTE:

wait(st) corrisponde a **waitpid(-1,st,0).**

while(wait(NULL)>0); attende tutti i figli.

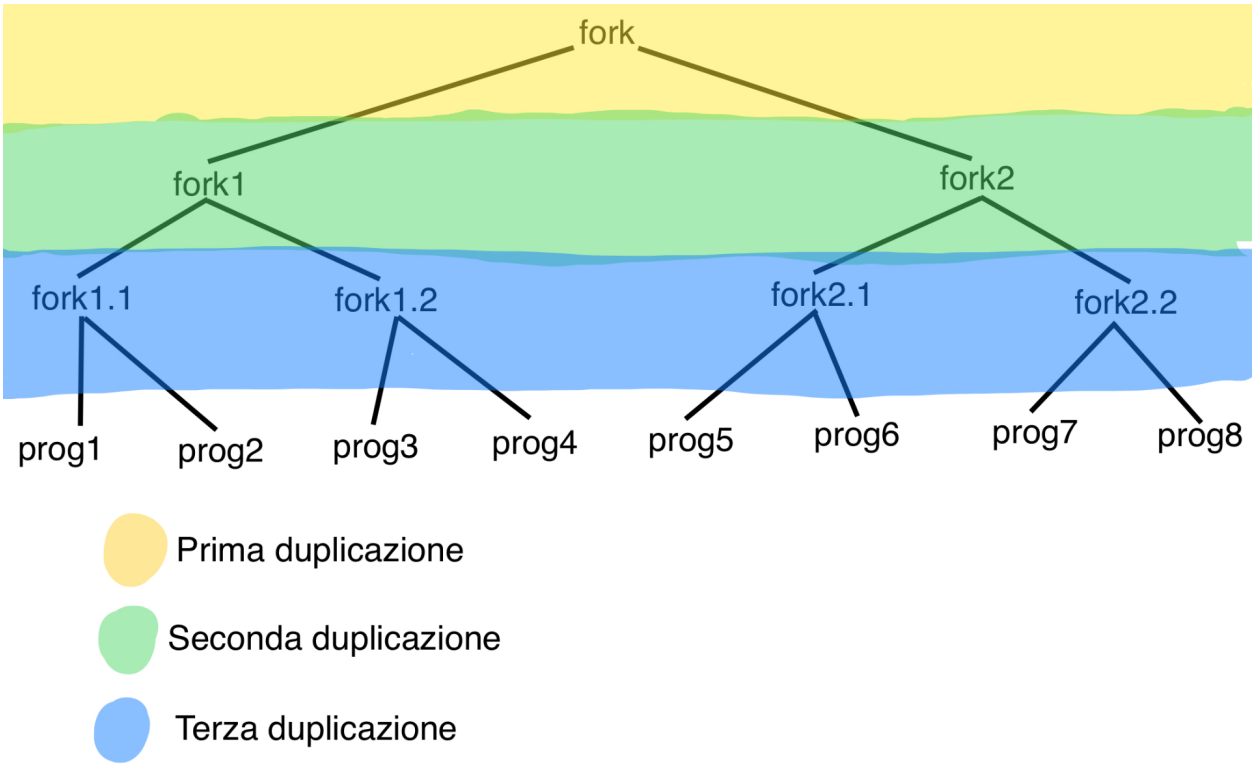
Wait: interpretazione stato

- **WEXITSTATUS(sts)** → restituisce lo stato vero e proprio (ad esempio il valore usato nella "exit")
- **WIFCONTINUED(sts)** → true se il figlio ha ricevuto segnale SIGCONT
- **WIFEXITED(sts)** → true se il figlio è terminato normalmente
- **WIFSIGNALED(sts)** → true se il figlio è terminato a causa di un segnale non gestito
- **WIFSTOPPED(sts)** → true se il figlio è attualmente in stato di "stop"
- **WSTOPSIG(sts)** → numero de segnale che ha causato lo "stop" del figlio
- **WTERMSIG(sts)** → numero del segnale che ha causato la terminazione del figlio

---

Esempio fork

```c
#include <stdio.h>                    //fork1.c
#include <unistd.h>
int main() {
  fork(); fork(); fork();
  printf("hello\n");
  return 0;
}
```



Esempio fork&wait

```c
#include <stdio.h> <stdlib.h> <unistd.h> <time.h> <sys/wait.h>          //fork2.c
int main() {
  int fid=fork(), wid, st, r; // Generate child
  srand(time(NULL)); // Initialise random
  r=rand()%256; // Get random
  if (fid==0) {  //If it is child
     printf("Child... (%d)", r); fflush(stdout);
     sleep(3);  // Pause execution for 3 seconds
     printf(" done!\n");
     exit(r); // Terminate with random signal
  } else { // If it is parent
     printf("Parent...\n");
     wid=wait(&st);  // wait for ONE child to terminate
     printf("...child's id: %d==%d (st=%d)\n", fid, wid, WEXITSTATUS(st));
  }
}
```