LabSO appunti

Ettore S. Enrico M. Alberto Z.

AA 2020-2021

README LAST EDIT: November 8, 2021
Il documento e' un WORK IN PROGRESS
Questi appunti sono arricchiti di link per approfondire i vari argomenti. E' fondamentale la conoscenza della lingua inglese.
Ho cercato di arricchire il documento il piu possibile per rendere lo studio completo. Se trovi errori sei pregato/a di contattarmi.
Sono stati inseriti molti link alle varie manpage, ne saranno aggiunti in futuro altri durante le riletture/-modifice del documento.
Alcune spiegazioni (principalmente per questioni di tempo) sono state tradotte da manpage e documentazioni varie ed arricchite di quello che ho ritenuto piu' utile.

UPDATE 27/02/2021 Pubblicazione documento. In futuro saranno modificati gli esempi con file .c linkati (cosi' sara' piu' facile comprendere, studiare e giocare con questi. Saranno aggiunti anche altri esempi.
UPDATE 27/07/2021 Gli esempi C nel documento sono i file .c presenti nel progetto GitHub. Corretti alcuni errori.

Contents

1 System calls e forking

Il kernel e' il nucleo del sistema operativo, gestisce le funzioni di controllo fondamentali del computer.

Ad ogni boot il kernel verifica lo stato delle periferiche, monta la prima partizione in read-only e lancia il primo programma (/sbin/init). Le altre operazioni vengono gestite da programmi eseguiti dal kernel. Permette ai programmi di accedere alle periferiche.

L'interazione tra programmi e il resto del sistema e' mascherata, ogni programma vede se stesso come unico possessore della CPU, quindi non puo' disturbare altri programmi.

1.1 Privilegi

- User space: ambiente in cui vengono eseguiti i programmi (Ring 3)
- Kernel space: ambiente in cui viene eseguito il kernel (Ring 0)
- Ring 1, 2 sono dedicati ai driver (non usati da Linux e Windows)

1.2 System calls

Eseguite nel Kernel Space, sono le interfacce con cui i programmi richiedono servizi dal Kernel, restituiscono i risultati nello user space.

Restituiscono '-1' in caso di errore e settano la var. glob. **errno** seguendo lo standard POSIX.

1.2.1 Librerie di sistema

Una libreria è un insieme di funzioni o strutture dati predefinite e predisposte per essere collegate ad un programma software attraverso un opportuno collegamento. Il collegamento può essere statico o dinamico, nel caso delle librerie di sistema il collegamento è dinamico (dynamic linking).

Il programma shell **ldd** *(acronimo di List Dynamic Dependencies)* esegue il programma che gli viene dato come argomento e non dovrebbe essere utilizzato con binari non fidati. Visualizza le librerie che il programma carica. ldd e' alias di `objdump -p /path/program | grep NEEDED`

- **ld-linux.so** trova e carica le librerie condivise richieste dal programma, prepara il programma all'esecuzione e lo esegue
- **libc.so** libreria nota come glibc, contiene le funzioni basilari più comuni

1.2.2 Get time

Libreria `time.h`, funzioni time() e ctime().

- `time()` restituisce il tempo dall'Epoch (00:00:00 UTC, January 1, 1970), misurato in secondi. WIKI

Esempio 1: time

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main()
{
    time_t seconds;
    seconds = time(NULL);
    printf("Ore dal 1 Gennaio 1970 = % ld \n", seconds / 3600);
    return (0);
}
```

- `ctime()` restituisce una stringa rappresentante l'ora locale basata sull'argomento timer. La stringa ritornata ha il formato: Www Mmm dd hh:mm:ss yyyy. WIKI

Esempio 2: ctime

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main()
{
    time_t curtime;     //variaible temporale
    time(&curtime);     //Return the current time and put it in *TIMER if TIMER is not NU
    printf("Tempo attuale = %s ", ctime(&curtime));
    return (0);
}
```

1.2.3 Working directory

Libreria `unistd.h`, libreria che consente l'accesso allo standard POSIX, funzioni chdir() e getcwd().

- `chdir()` change directory - cambia la directory su cui stiamo lavorando
- `getcwd()` get current working directory - stampa la directory in cui siamo

```c
#include <unistd.h>
#include <stdio.h>
void main()
{
    char s[100]; //array di caratteri su cui scrivere il path
    chdir(" .. "); // Change working dir
    printf(" %s \n ", getcwd(s, 100)); // Print current working dir
}
```

1.2.4 Operazioni con i file

```c
    int open(const char *pathname, int flags, mode_t mode);
    int close(int fd);
    ssize_t read(int fd, void *buf, size_t count);
    ssize_t write(int fd, const void *buf, size_t count);
    off_t lseek(int fd, off_t offset, int whence);

    FILE *fopen(const char *filename, const char *mode);
    int fclose(FILE *stream)
```

Si utilizzano le librerie `unistd.h` IMB IEEE e `unistd.h`

Duplicazione dei file descriptors: Un file descriptor è un **handle opaco** utilizzato nell'interfaccia tra lo spazio utente e il kernel per **identificare le risorse di file** / socket. Pertanto, quando si utilizza open() o socket() (chiamate di sistema per interfacciarsi al kernel), viene fornito un descrittore di file, che è un **numero intero** (sarebbe un indice nella struttura dei processi, ma non ci interessa). Pertanto, se vuoi interfacciarti direttamente con il kernel usando le chiamate di sistema a read(), write(), close() ecc. l'handle che usi è un file descriptor.

Ogni qualvolta un nuovo file viene aperto da un programma viene creata un'entry nella file table del kernel. Queste **entry sono indirizzabili da un processo tramite il file descriptor (un numero intero)**.
C'è uno strato di **astrazione** sovrapposto alle chiamate di sistema, che è l'interfaccia `stdio`. Ciò fornisce più funzionalità / caratteristiche rispetto alle chiamate di sistema di base. Per questa interfaccia, **l'handle opaco che ottieni è un FILE*,** che viene restituito dalla chiamata fopen(). Ci sono molte funzioni che usano l'interfaccia stdio fprintf(), fscanf(), fclose(), che sono lì per semplificarti la vita. **In C, stdin, stdout e stderr sono FILE*, che in UNIX corrispondono rispettivamente ai descrittori di file 0, 1 e 2.** Semplificando con un elenco:

- i File Descriptor (FD) sono interi non negativi (0, 1, 2, ...) associati ai file aperti
- 0, 1, 2 sono standard FD che corrispondono agli `STDIN_FILENO,STDOUT_FILENO e STDERR_FILENO` (definiti in unistd.h) aperti di default quando un programma shell viene eseguito
- gli FD di un processo particolare possono essere visti in /proc/$pid/fd (in un sistema Unix based)

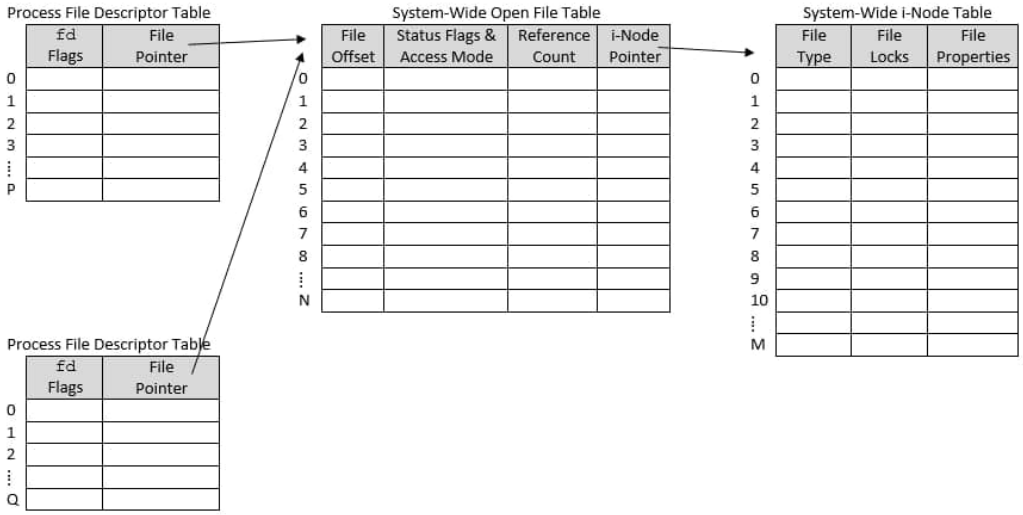

Figure 1: Wow such cool

Operazioni sugli FD: dup(), dup2() Libreria `unistd.h`

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

Le funzioni dup() e dup2() creano una copia del file descriptor oldfd. La funzione dup() attribuisce al nuovo file descriptor, il piu' piccolo intero non usato. La funzione dup2() crea newfd come copia di oldfd, chiudendo prima newfd se e' necessario.

Il vecchio e nuovo file descriptor possono essere utilizzati interscambiabilmente. Essi condividono locks, puntatiori di file position e flag, ad eccezione del flag close-on-exec. Per Esempio e' possibile effettuare una lseek() su un file descriptor e ritorvarsi la posizione modificata su entrami i file descriptor.

Esempio 4: dup e dup2

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
void main()
{
    char buf[50];
    int fd = open("file.txt", O_RDWR); // file exists
    read(fd, buf, 50);
    printf("Content: %s \n", buf);
    int cpy = dup(fd);                  // Create cpy of file descriptor
    dup2(cpy, 22);                      // Copy cpy to descriptor 22( close 22 if opened )
    lseek(cpy, 0, SEEK_SET);            // move I / O on all 3 file descriptors !
    write(22, " This is a file ", 15); // Write starting from 0 - pos }
}
```

Esempio: (non ha file C) Stai scrivendo un programma shell e vuoi che ridiriga stdin e stdout in un processo figlio. Dovrebbe assomigliare a:

```
fdin = open(infile, O_RDONLY);
fdout = open(outfile, O_WRONLY);
// Check for errors, send messages to stdout.
...
int pid = fork();
if(pid == 0) {
    close(STDIN_FILENO);
    dup2(fdin, STDIN_FILENO);

    close(STDOUT_FILENO);
    dup2(fdout, STDOUT_FILENO);

    execvp(program, argv);
}
// Parent process cleans up, maybe waits for child.
...
```

1.2.5 Permessi

Libreria `sys/stat.h e fcntl.h`
Sebbene ci siano già molte buone funzionalità di sicurezza integrate nei sistemi basati su Linux, può esistere una potenziale vulnerabilità quando viene concesso l'accesso locale, ossia problemi basati sui **permessi dei file dovuti** da un utente che non assegna i permessi corretti a file e directory.

Proprietà dei file Linux Ad ogni file e directory sul tuo sistema Unix / Linux vengono assegnati **3 tipi di proprietario**:

- **(u) User - Utente è il proprietario del file.** Per impostazione predefinita, la persona che ha creato un file diventa il suo **proprietario (owner)**.
- **(g) Group - Gruppo di utenti può contenere più utenti. Tutti gli utenti che appartengono a un gruppo avranno le stesse autorizzazioni del gruppo Linux per accedere al file.** Supponi di avere un progetto in cui un numero di persone richiede l'accesso a un file. Invece di assegnare manualmente le autorizzazioni a ciascun utente, è possibile aggiungere tutti gli utenti a un gruppo e assegnare l'autorizzazione del gruppo al file in modo che solo i membri del gruppo e nessun altro possano leggere o modificare i file.
- **(o) Other - Qualsiasi altro utente che abbia accesso a un file.** Questa persona non ha creato il file, né appartiene a un gruppo utenti che potrebbe possedere il file. In pratica, significa tutti gli altri. Pertanto, quando si imposta l'autorizzazione per gli altri, viene anche definita autorizzazioni impostate per il mondo.

Come fa Linux a distinguere tra questi tre tipi di utente in modo che un *utente "A" non possa influenzare un file che contiene informazioni / dati vitali di altri utenti "B"*? È come se non volessi che il tuo collega, che lavora sul tuo computer Linux, visualizzi le tue immagini. È qui che si inseriscono le autorizzazioni e definiscono il comportamento dell'utente.

Autorizzazioni Ogni file e directory nel tuo sistema UNIX / Linux ha i seguenti **3 permessi definiti per tutti e 3 i proprietari discussi sopra.**

- **(r) Read - Lettura**: questa autorizzazione ti dà l'**autorità di aprire e leggere un file**. Il permesso di lettura su una directory ti dà la possibilità di elencarne il contenuto.
- **(w) Write - Scrittura**: l'autorizzazione di scrittura consente di **modificare il contenuto di un file**. L'autorizzazione di scrittura su una directory ti dà l'autorità per aggiungere, rimuovere e rinominare i file archiviati nella directory. Si consideri uno scenario in cui è necessario l'autorizzazione di scrittura sul file ma non l'autorizzazione di scrittura sulla directory in cui è archiviato il file. Potrai modificare il contenuto del file. Ma non sarai in grado di rinominare, spostare o rimuovere il file dalla directory.

- **(x) Execute - Esegui**: in Windows, un programma eseguibile di solito ha un'estensione ".exe" e che puoi eseguire facilmente. In Unix / Linux, non è possibile eseguire un programma a meno che non sia impostata l'autorizzazione di esecuzione. Se l'**autorizzazione di esecuzione** non è impostata, potresti comunque essere in grado di vedere / modificare il codice del programma (a condizione che siano impostati i permessi di lettura e scrittura), ma non eseguirlo.

Metodo testuale Scomodo, leggi la wiki se interessato.

Metodo numerico L'utilizzo dei numeri è un altro metodo che consente di modificare le autorizzazioni per tutti e tre i proprietari, i gruppi e altri contemporaneamente, nonché i bit setuid, setgid e sticky. Questa struttura di base del codice è questa:

$ `chmod xxx nome file`

Dove xxx è un numero di 3 cifre dove ogni cifra può essere qualsiasi cifra da 0 a 7. La prima cifra si applica alle autorizzazioni per il proprietario, la seconda cifra si applica alle autorizzazioni per il gruppo e la terza cifra si applica alle autorizzazioni per tutti gli altri.

Esempio: $ `chmod 754 filename` — 7 -> Owner — 5 -> Group — 4 -> Others

Number	Permission Type	Symbol
0	No Permission	—
1	Execute	–x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r–
5	Read + Execute	r-x
6	Read + Write	rw-
7	Read + Write + Execute	rwx

Table 1: Numeri dei permessi

wiki — info permessi — altre info sui permessi

chmod(), chown()

- **chmod()** - change mode - modifica i permessi di file e directory
- **chown()** - change owner - modifica il proprietario e/o il gruppo assegnato di uno o più file e directory

```
1    int chown(const char *pathname, uid_t owner, gid_t group)
2    int fchown(int fd, uid_t owner, gid_t group)
3    int chmod(const char *pathname, mode_t mode)
4    int fchmod(int fd, mode_t mode)
```

La differenza tra chown e fchown consiste che il primo modifica i permessi del file dal suo percorso, il secondo dal suo Process Id (fd), allo stesso modo per chmod e fchmod.

Esempio 5: chown e chmod

```
1  #include <fcntl.h> // execute with sudo ! // chown . c
2  #include <unistd.h>
3  #include <sys/stat.h>
4  void main()
5  {
6      int fd = open("file.txt", O_RDONLY);
7      fchown(fd, 1000, 1000);                                  // Change owner to root : root, il pr
8      //chmod("file", S_IRUSR | S_IRGRP | S_IROTH); // Permission to r / r / r
9      chmod("file.txt", 0744); // Permission to rwx / rwx / rwx
10     //funziona anche con la notazione numerica (piu semplice).
11 }
```

La funziona chmod() funziona anche con i permessi numerici!

1.2.6 Eseguire programmi - correggere libreria e spiegare - dividere i programmi in 'esempi' e formattare meglio

Libreria `unistd.h` – info utili

```
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execvpe(const char *file, char *const argv[],
    char *const envp[])
int execl(const char *path, const char *arg0,...,argn,NULL)
int execlp(const char *file, const char *arg0,...,argn,NULL)
int execle(const char *file, const char *arg0,...,argn, NULL,
    char *const envp[])
int execve(const char *filename, char *const argv[],
    char *const envp[])
```

Esempio 6: execv()

```
#include <unistd.h> // execv1.out
#include <stdio.h>
void main()
{
    char *argv[] = {"par1", "par2", NULL};
    execv("./esempio6_execv2", argv); // sostituisce il processo attuale e passa argv -- NB i
    printf("This is execv1 \n"); //se non esegue execv2 allora stampa questo messaggio e cont
}
```

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

Esempio 7: execle()

```
#include <unistd.h> // execle1 . out
#include <stdio.h>
void main()
{
    char *env[] = {"CIAO = hello world", NULL};
    execle("./esempio7_execle2", "par1", "par2", NULL, env); // sostituisce il processo attua
    printf (" This is execle1 n");
}
```

```
#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
}
```

Esempio 8: dup2/exec

```
#include <stdio.h> // execvpDup.c
#include <fcntl.h>
#include <unistd.h>

void main()
{
    int outfile = open("/tmp/out.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
```

```
8       dup2(outfile, 1);                           // copy outfile to FD 1
9       char *argv[] = {"./time.out", NULL}; // time . out
10      execvp(argv[0], argv);                      // Replace current process
11  }
```

Esempio 9: chiamare la shell, system()
Se si accede alle cartelle con system bisogna stare attenti perche' si vedono da root.

```
1   // comando : int system ( const char * string )
2   #include <stdlib.h> //system.c
3   #include <stdio.h>
4   void main()
5   {
6       // / bin / sh -c string
7       int outcome = system("echo ciao"); // execute command in shell
8       printf("%d \n", outcome);
9       outcome = system("if [ [ $PWD < \"ciao\" ] ]; then echo min ; fi ");
10      printf("%d \n", outcome);
11  }
```

```
1   // comando : int system ( const char * string )
2   #include <stdlib.h> //system.c
3   #include <stdio.h>
4   void main()
5   {
6       // / bin / sh -c string
7       int outcome = system("cd /home/$USER/abadakadabra"); // execute command in shell
8       printf("%d \n", outcome);
9       outcome = system("mkdir abadakadabra");
10      printf("%d \n", outcome);
11      outcome = system("rm -rf abadakadabra");
12      printf("%d \n", outcome);
13      outcome = system("echo $USER");
14      printf("%d \n", outcome);
15  }
```

1.3 Fork

La chiamata di sistema **Fork viene utilizzata per creare un nuovo processo, chiamato processo figlio, che viene eseguito contemporaneamente al processo che effettua la chiamata fork()** (processo genitore). Dopo aver creato un nuovo processo figlio, entrambi i processi eseguiranno l'istruzione successiva che segue la chiamata di forking. Un processo figlio utilizza lo stesso PC (Program Counter), gli stessi registri della CPU e gli stessi file utilizzati nel processo genitore.

Non accetta parametri e restituisce un valore intero. I valori restituiti dal fork() sono:

- **Valore negativo**: la creazione di un processo figlio **non è riuscita**.
- **Zero**: **restituito al processo figlio appena creato**.
- **Valore positivo**: restituito **al genitore o al chiamante**. Il valore contiene l'**ID del processo figlio appena creato**.

Se la creazione del figlio ha **successo entrambi i processi ricevono un valore di ritorno**, ma questo è diverso nei due casi:

- Il processo padre riceve come valore il nuovo PID del processo figlio
- Il processo figlio riceve come valore 0

LINK ALTRE INFO

1.3.1 Identificativi dei processi

Ad ogni processo è associato un identificativo univoco per istante temporale, sono organizzati gerarchicamente (padre-figlio) e suddivisi in insiemi principali (sessioni) e secondari (gruppi). Anche gli utenti hanno un loro identificativo e ad ogni processo ne sono abbinati due: quello reale e quello effettivo (di esecuzione).

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

getpid(), getppid()

```
pid_t getpid() :  restituisce il PID del processo attivopid_t
getppid() :  restituisce il PID del processo padre
```

Esempio 10: getpid e getppid

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    pid_t process_id;         // variable to store calling function 's process id
    pid_t p_process_id;       // variable to store parent function 's process id
    process_id = getpid();    // getpid () - will return process id of calling function
    p_process_id = getppid(); // getppid () - will return process id of parent function
    // printing the process ids
    printf("The process id :%d \n", process_id);
    printf("The process id of parent function :%d\n", p_process_id);
    return 0;
}
/*
OUTPUT :
The process id : 31120
The process id of parent function : 31119
*/
```

1.3.2 Relazione tra i processi

I processi padre-figlio:

- **Conoscono reciprocamente il loro PID** (ciascuno conosce il proprio tramite getpid(), il figlio conosce quello del padre con **getppid()**, il padre conosce quello del figlio come valore di ritorno di fork())
- Si possono usare altre syscall per semplici interazioni come `wait e waitpid`
- Eventuali variabili definite prima del fork sono valorizzate allo stesso modo in entrambi: se riferiscono risorse (ad Esempio un "file descriptor" per un file su disco) fanno riferimento esattamente alla stessa risorsa.

1.3.3 Processi zombie e orfani

wait(), waitpid() Librerie usate: `sys/types.h> e <sys/wait.h>`

La funzione `wait()` **sospende il processo corrente finche' un figlio (child) termina** o finche' il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione. Quando un **child termina il processo, senza che il parent abbia atteso la sua terminazione** attraverso la funzione di wait(), allora il child assume lo **stato di "zombie"** ossia di processo "defunto". Se il processo corrente **esegue la funzione di wait(), in presenza di child in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata.**

La funzione `waitpid()` `sospende il processo corrente finche' il figlio (child) corrispobndente al pid passato in argomento termina` o finche' il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione. Se il processo corrente **esegue la funzione di waitpid() e il child identificato dal pid e' in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata** (altre info sui processi zombie).

Il valore del pid puo' essere uno dei seguenti:

- -n (<-1: attende un qualunque figlio il cui "gruppo" è |-n|)
- -1 (attende un figlio qualunque)
- 0 (attende un figlio con lo stesso "gruppo" del padre)
- n (n>0: attende il figlio il cui pid è esattamente n)

Se **status non e' NULL, le funzioni wait() e waitpid() memorizzano l'informazione dello stato nell'area di memoria puntata da questo argomento**.

Comandi utili:

```
1    wait(st) corrisponde a waitpid(-1, st, 0)
2    while(wait(NULL)>0); # attende tutti i figli
```

Sono disponibili alcune **macro** per la valutazione dello stato status:

- **WEXITSTATUS(sts)**: restituisce lo stato vero e proprio (ad Esempio il valore usato nella "exit")
- **WIFCONTINUED(sts)**: true se il figlio ha ricevuto un segnale
- **SIGCONTWIFEXITED(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 del segnale che ha causato lo "stop" del figlio
- **WTERMSIG(sts)**: numero del segnale che ha causato la terminazione del figlio

Esempio 11: forking multiplo

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

Esempio 12: fork&wait

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

2 Segnali

I **segnali** sono una forma limitata di **comunicazione inter-processo (IPC).** Un segnale è una **notifica asincrona inviata a un processo** o a un thread specifico all'interno dello stesso processo **per notificarlo di un evento**.

Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per Esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico.

Quando viene inviato un segnale, **il sistema operativo interrompe il normale flusso di esecuzione del processo di destinazione per fornire il segnale**. Se il processo ha precedentemente registrato un gestore di segnali, quella routine viene eseguita. In caso contrario, viene eseguito il gestore del segnale predefinito.
Per ogni processo, all'interno della process table, vengono mantenute due liste:

- **Pending signals**: **segnali emessi che il processo dovrà gestire** - sono segnali nello stato "D" (**uninterruptible sleep**). Solitamente perche' il processo e' in attesa (tipo di I/O). Questo sonno non puo' essere interrotto, nemmeno sigkill (da terminale `kill -9`) puo'. Il kernel attende che il processo si "risvegli" e il segnale viene consegnato.
- **Blocked signals**: **segnali non comunicati al processo**. Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata.

SIGXXX	**description**	**default**
SIGALRM	*alarm clock*	quit
SIGCHLD	*child terminated*	ignore
SIGCONT	*continue, if stopped*	ignore
SIGINT	*terminal interrupt, CTRL + C*	quit
SIGKILL	*kill process*	quit
SIGSYS	*bad argument to syscall*	quit with dump
SIGTERM	*software termination*	quit
SIGUSR1/2	*user signal 1/2*	quit
SIGSTOP	*stopped*	quit
SIGTSTP	*terminal stop, CTRL + Z*	quit

Table 2: Alcuni Segnali

2.1 Gestione dei segnali

I segnali sono **simili agli interrupt**, con la differenza che gli interrupt sono mediati dal processore e gestiti dal kernel mentre **i segnali sono mediati dal kernel e gestiti dai processi**. Come per gli interrupts, **il programma può decidere come gestire l'arrivo di un segnale** (presente nella **lista pending**):

- **Eseguendo l'azione default**
- **Ignorandolo** (non sempre possibile) → programma prosegue normalmente
- **Eseguendo un handler personalizzato** → programma si interrompe

2.2 Default handler

Ogni segnale ha un suo handler di default che tipicamente può:

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era in stop)

- Stoppare il processo

Ogni processo può sostituire il gestore di default con una funzione "custom" (a parte per `SIGKILL e SIGSTOP`) e comportarsi di conseguenza. La sostituzione avviene tramite la **system call** `signal()`

Libreria `\signal.h"` | Info Libreria. || LINUXDIE

signal() system call

```
1      sighandler_t signal(int signum, sighandler_t handler);
2      typedef void (*sighandler_t)(int);
```

Esempio 13: signal()

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  void myHandler(int sigNum)
5  {
6      printf(" CTRL + Z \n");
7  }
8
9  void main()
10 {
11     signal(SIGINT, SIG_IGN);     // Ignore signal
12     signal(SIGCHLD, SIG_DFL);    // Use default handler
13     signal(SIGTSTP, myHandler); // Use myHandler
14 }
```

Esempio 14: (premo CTRL+C durante l'esecuzione)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <signal.h>
5
6  void sighandler(int signum)
7  {
8      printf(" Caught signal %d , coming out ...\n ", signum);
9      exit(1);
10 }
11
12 int main()
13 {
14     signal(SIGINT, sighandler);
15     while (1)
16     {
17         printf(" Going to sleep for a second ...\n ");
18         sleep(1);
19     }
20     return (0);
21 }
```

Esempio 15:

```
1  #include <signal.h> // sigCST . c
2  #include <stdio.h>
```

```c

void myHandler(int sigNum)
{
    printf("CTRL + Z\n");
    exit(2);
}

int main()
{
    signal(SIGTSTP, myHandler);
    while (1);
}
```

```c
#include <signal.h> // sigDFL . c

int main()
{
    signal(SIGTSTP, SIG_DFL);
    while (1);
}

/*
SIG_DFL:
It does exactly what one would expect: informs the kernel
that there is no user signal handler for the given signal,
and that the kernel should take default action for it
(the action itself may be to ignore the signal,
to terminate the program (with or without core dump),
etc. depending on the signal).
*/
```

```c
#include <signal.h> // sigIGN . c

int main()
{
    signal(SIGTSTP, SIG_IGN);
    while (1);
}
//SIG_IGN       signal is ignored
```

2.3 Custom Handler

Un handler personalizzato deve essere una funzione di tipo void che accetta come argomento un intero, il quale rappresenta il segnale catturato. Questo consente l'utilizzo di uno stessa handler per segnali differenti.

```c
    #include <signal.h> <stdio.h> //param.c
    void myHandler(int sigNum){
        if(sigNum == SIGINT) printf("CTRL+C\n");
        else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
    }
    signal(SIGINT,myHandler);
    signal(SIGTSTP,myHandler);
```

signal() return Signal() restituisce un riferimento all'handler che era precedentemente assegnato al segnale:

- NULL: handler precedente era l'handler di default
- 1: l'handler precedente era SIG_IGN
- address: l'handler precedente era *(address)

Esempio 6:

```c
#include <signal.h>
#include <stdio.h> // return . c

void myHandler(int sigNum) {}

int main()
{
    printf("DFL : %p \n", signal(SIGINT, SIG_IGN));
    printf("IGN : %p \n", signal(SIGINT, myHandler));
    printf("Custom : %p == %p \n ", signal(SIGINT, SIG_DFL), myHandler);
}
```

Esempio 17: oggi, riguardare i process ID di padre e figlio se metto printf("%d",child);

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

void myHandler(int sigNum)
{
    printf("Child terminated !\n ");
}
// child . c
int main()
{
    signal(SIGCHLD, myHandler);
    int child = fork();

    if (!&child)
    {
        return 0; // terminate child
    }
    while (wait(NULL) > 0);
}
```

2.4 Inviare i segnali: kill()

kill() `int kill(pid_t pid, int sig);`
Invia un segnale ad uno o più processi a secondo dell'argomento pid:

- pid > 0: segnale al processo con PID=pid
- pid = 0: segnale ad ogni processo dello stesso gruppo
- pid = -1: segnale ad ogni processo possibile (stesso UID/RUID)
- pid < -1: segnale ad ogni processo del gruppo —pid—Restituisce 0 se il segnale viene inviato, -1 in caso di errore.

Ogni tipo di segnale può essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto!

Esempio 8:

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
```

```c
#include <unistd.h> // kill . c

void myHandler(int sigNum)
{
    printf("[%d] ALARM !\n", getpid());
}

int main()
{
    signal(SIGALRM, myHandler);
    int child = fork();
    if (!child)
    {
        while (1); // this is the child
    }

    printf("[%d] sending alarm to %d in 1s \n", getpid(), child);
    sleep(1);
    kill(child, SIGALRM); // send ALARM , child 's handler reacts
            //remember that kill() is used to send alarms.
    printf("[%d] sending SIGTERM to %d in 1s \n", getpid(), child);
    sleep(1);
    kill(child, SIGTERM); // send TERM : default is to terminate

    while (wait(NULL) > 0);
}
```

altri esempi
come uccidere un figlio dal processo genitore

Kill da bash kill è un programma bash che accetta come primo argomento il tipo di segnale e come secondo argomento il PID del processo. Per maggiori info (da terminale) `man kill` per info su kill, `man signal` per informazioni sui segnali.

Esempio 19:

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // bash . c

void myHandler(int sigNum)
{
    printf("[%d] ALARM!\n", getpid());
    exit(0);
}
int main()
{
    signal(SIGALRM, myHandler);
    printf("I am %d \n", getpid());
    while (1);
}

/*
$ gcc bash . c -o bash . out
$ ./ bash . out
# On new window / terminal
$ kill -14 <PID >

in pratica si esegue il programma che stampa il proprio PID e
continua ad eseguire; lo si uccide col comando kill -14 <PID> da
```

```
terminale

DA MAN: significato di kill -14
SIGALRM P1990 Term Timer signal from alarm (2)
*/
```

2.5 Programmare un alarm: alarm()

`unsigned int alarm(unsigned int seconds);`
La funzione alarm () viene utilizzata per generare un segnale SIGALRM dopo che è trascorso un periodo di tempo specificato.

La funzione richiede come argomento i secondi. Dopo che sono trascorsi tot secondi dalla richiesta della funzione alarm(), viene generato il segnale SIGALRM. Il comportamento predefinito al ricevimento di SIGALRM è di terminare il processo. Ma possiamo catturare e gestire il segnale come preferiamo.

La funzione alarm() restituirà un valore diverso da zero, se un altro allarme è stato precedentemente impostato e il valore è il numero di secondi rimanenti per il precedente avviso programmato dovuto alla consegna. Altrimenti alarm() restituirà zero.
altri esempi e un po' di WIKI ‖ MANPAGE

Esempio 20:

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //alarm.c
short cnt = 0;
void myHandler(int sigNum)
{
    printf("ALARM !\n");
    cnt++;
}
int main()
{
    signal(SIGALRM, myHandler);
    alarm(5); // Set alarm in 5 seconds
    // Set new alarm ( cancelling previous one )
    printf("Seconds remaining to previous alarm %d \n", alarm(2));
    while (cnt < 1); //questo while fa aspettare 2 secondi
}
```

Esempio 21:

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
/* number of times the handle will run : */
volatile int breakflag = 3;

void handle(int sig)
{
    printf("Hello \n");
    --breakflag;
    alarm(1);
}

int main()
{
    signal(SIGALRM, handle);
    alarm(1);
    while (breakflag)
```

```
19      {
20          sleep(1);
21      }
22      printf("done\n");
23      return 0;
24  }
```

2.6 Mettere in pausa: pause()

`int pause();` Sospende l'esecuzione del thread chiamante. Il thread non riprende l'esecuzione finché non viene consegnato un segnale e viene eseguito un gestore di segnale o terminato il thread.

Se un segnale non bloccato in arrivo termina il thread, pause() non torna mai al chiamante. Se un segnale in arrivo è gestito da un gestore di segnali, pause() ritorna dopo che il gestore di segnali è tornato.

Se pause () ritorna, restituisce sempre -1 e imposta errno su EINTR, indicando che un segnale è stato ricevuto e gestito con successo.

da IBM ‖ MANPAGE

Esempio 22:

```
1  #include <signal.h>
2  #include <unistd.h>
3  #include <stdio.h> // pause . c
4
5  void myHandler(int sigNum)
6  {
7      printf("Continue!\n");
8  }
9
10 int main()
11 {
12     printf("%d\n", getpid());
13     signal(SIGCONT, myHandler); //Alarm clock
14     signal(SIGUSR1, myHandler); //Continue!
15     pause();
16 }
17
18 /*
19 $ gcc pause . c -o pause . out
20 $ ./ pause . out
21 # On new window / terminal
22 $ kill -18/ -10 <PID >
23 */
```

2.7 Bloccare i segnali

Si considera la lista dei "blocked signals", i segnali ricevuti dal processo ma volutamente non gestiti.

Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo temporaneamente non gestiti. Un segnale bloccato rimane nello stato *pending* fino a quando esso non **viene gestito** oppure il suo handler tramutato in ignore. L'insieme dei segnali bloccati è detto **"signal mask", una maschera dei segnali che è modificabile attraverso** la system call `sigprocmask()`

altre info — MANPAGE

sigset_t Una signal mask può essere gestita con un `sigset_t`, una lista di segnali modificabile con alcune funzioni (non modificano maschera dei segnali del processo).

sigprocmask() `int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oldset);`

A seconda del valore di how e di set, la maschera dei segnali del processo viene cambiata. Nello specifico:

- **how = SIG_BLOCK**: i segnali in set sono aggiunti alla maschera;

int sigemptyset(sigset_t *set);	Svuota
int sigfillset(sigset_t *set);	Riempie
int sigaddset(sigset_t *set, int signo);	Aggiunge singolo
int sigdelset(sigset_t *set, int signo);	Rimuove singolo
int sigismember(const sigset_t *set, int signo);	Interpella

Table 3: Alcune funzioni `sigset_t`

- **how = SIG_UNBLOCK**: i segnali in set sono rimossi dalla maschera;
- **how = SIG_SETMASK**: set diventa la maschera.

Se oldset non è nullo, in esso verrà salvata la vecchia maschera. oldset viene riempito anche se set è nullo.
da IBM, leggere assolutamente

Esempio 23:

```c
#include <signal.h> //sigSet.c
int main()
{
    sigset_t mod, old;
    sigfillset(&mod);           // Add all signals to the blocked list
    sigemptyset(&mod);          // Remove all signals from blocked list
    sigaddset(&mod, SIGALRM);   // Add SIGALRM to blocked list
    sigismember(&mod, SIGALRM); // is SIGALRM in blocked list ?
    sigdelset(&mod, SIGALRM);   // Remove SIGALRM from blocked list
    // Update the current mask with the signals in ' mod '
    sigprocmask(SIG_BLOCK, &mod, &old);
}

/*
$ gcc sigprocmask.c -o sigprocmask.out
$ ./sigprocmask.out
# On new window/terminal
$ kill -10 <PID> # ok
$ kill -10 <PID> # blocked
*/
```

Esempio 24:

```c
// sigpending . c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
sigset_t mod, pen;

void handler(int signo)
{
    printf("SIGUSR1 received \n");
    sigpending(&pen);
    if (!sigismember(&pen, SIGUSR1))
        printf("SIGUSR1 not pending \n");
    exit(0);
}

int main()
{
    signal(SIGUSR1, handler);
```

```
20      sigemptyset(&mod);
21      sigaddset(&mod, SIGUSR1);
22      sigprocmask(SIG_BLOCK, &mod, NULL);
23      kill(getpid(), SIGUSR1); // sent but it 's blocked ...
24      sigpending(&pen);
25      if (sigismember(&pen, SIGUSR1))
26          printf("SIGUSR1 pending \n");
27      sigprocmask(SIG_UNBLOCK, &mod, NULL);
28      while (1);
29  }
```

2.8 Verificare pending signals: sigpending()

`sigpending()` restituisce l'insieme di segnali che sono in attesa di consegna al thread chiamante (cioè, i segnali che sono stati ricevuti mentre erano bloccati).

Esempio 25:

```
1   #include <signal.h>
2   #include <unistd.h>
3   #include <stdio.h> //sigprocmask.c
4   sigset_t mod, old;
5   int i = 0;
6   void myHandler(int signo)
7   {
8       printf("signal received \n");
9       i++;
10  }
11  int main()
12  {
13      printf("my id = %d \n", getpid());
14      signal(10, myHandler);
15      sigemptyset(&mod);
16      sigaddset(&mod, SIGUSR1);
17      while (1)
18          if (i == 1)
19              sigprocmask(SIG_BLOCK, &mod, &old);
20  }
21
22  /*
23  $ gcc sigprocmask . c -o sigprocmask . out
24  $ ./ sigprocmask . out
25  # On new window / terminal
26  $ kill -10 <PID > # ok
27  $ kill -10 <PID > # blocked
28  */
```

sigaction() - sintetizzare Esamina e modifica l'azione associata a un segnale specifico.

int sig è il numero di un segnale riconosciuto. sigaction () esamina e imposta l'azione da associare a questo segnale. Vedere la Tabella 1 per i valori di sig, nonché i segnali supportati dai servizi z / OS® UNIX. L'argomento sig deve essere una delle macro definite nel file di intestazione signal.h.

const struct sigaction * new può essere un puntatore NULL. In tal caso, sigaction () determina semplicemente l'azione attualmente definita per gestire sig. Non cambia questa azione. Se new non è NULL, dovrebbe puntare a una struttura sigaction. L'azione specificata in questa struttura diventa la nuova azione associata a sig.

struct sigaction * old punta a una posizione di memoria in cui sigaction () può memorizzare una struttura sigaction. sigaction () utilizza questa posizione di memoria per memorizzare una struttura sigaction che descrive l'azione attualmente associata a sig. old può anche essere un puntatore NULL, nel qual caso sigaction () non memorizza queste informazioni.

Questa funzione è supportata solo in un programma POSIX. Comportamento speciale per C ++:

Il comportamento quando si combina la gestione del segnale con la gestione delle eccezioni C ++ non è definito. Inoltre, l'uso della gestione del segnale con costruttori e distruttori non è definito. Le convenzioni di collegamento del linguaggio C ++ e C non sono compatibili e pertanto sigaction () non può ricevere puntatori a funzione C ++. Se si tenta di passare un puntatore a funzione C ++ a sigaction (), il compilatore lo contrassegna come errore. Pertanto, per utilizzare la funzione sigaction () nel linguaggio C ++, è necessario assicurarsi che le routine di gestione dei segnali stabilite abbiano un collegamento C, dichiarandole come "C" esterno.

da IBM|linux.die.net

```
1     int sigaction(int signum, const struct sigaction *restrict act,
2     struct sigaction *restrict oldact);
3
4     struct sigaction {
5         void (*sa_handler)(int);
6         void (*sa_sigaction)(int, siginfo_t *, void *);
7         sigset_t sa_mask; //Signals blocked during handlerint sa_flags;
8         //modify behaviour of signal
9         void (*sa_restorer)(void); //Deprecated
```

Esempio 26:

```
1  #include <signal.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h> // sigaction . c
5  void handler(int signo)
6  {
7      printf("signal received \n");
8  }
9  int main()
10 {
11     struct sigaction sa;      // Define sigaction struct
12     sa.sa_handler = handler;  // Assign handler to struct field
13     sigemptyset(&sa.sa_mask); // Define an empty mask
14     sigaction(SIGUSR1, &sa, NULL);
15     kill(getpid(), SIGUSR1);
16 }
```

Esempio 27: blocking signal

```
1  #include <signal.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h> // sigaction2 . c
5  void handler(int signo)
6  {
7      printf("signal %d received \n ", getpid());
8      sleep(2);
9      printf("Signal done \n");
10 }
11 int main()
12 {
13     printf("Process id : %d \n ", getpid());
14     struct sigaction sa;
15     sa.sa_handler = handler;
16     sigemptyset(&sa.sa_mask);
17     sigaction(SIGUSR1, &sa, NULL);
18     while (1)
19         ;
```

```
}

/*
$ ./ sigaction2 . out
# On new window / terminal
$ kill -10 <PID > ; sleep 1
&& kill -12 <PID >
*/
```

Esempio 28: blocking signal

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h> // sigaction3 . c
void handler(int signo)
{
    printf("signal %d received \n", getpid());
    sleep(2);
    printf("Signal done \n");
}
int main()
{
    printf("Process id: %d \n", getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGUSR2); // Block SIGUSR2 in handler
    sigaction(SIGUSR1, &sa, NULL);
    while (1);
}
/*
$ ./ sigaction3 . out
# On new window / terminal
$ kill -10 <PID > ; sleep 1
&& kill -12 <PID >
*/
```

Esempio 29: sa_sigaction

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h> // sigaction4 . c

void handler(int signo, siginfo_t *info, void *empty)
{
    printf("Signal received from %d \n", *(info));
}
int main()
{
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags |= SA_SIGINFO;   // Use sa_sigaction
    sa.sa_flags |= SA_RESETHAND; // Restore default handler after exiting custom one sigactio
    while (1);
}
/*
$ ./ sigaction4 . out
# On new window / terminal
```

```
22 $ echo $$ ; kill -10 <PID > # custom
23 $ kill -10 <PID > # default
24 */
```

easter egg Trova le applicazioni per signal.h (UBUNTU MAN PAGE)

3 Gruppi di processi

3.1 Gestione processi in Unix

All'interno di Unix i processi vengono raggruppati secondi vari criteri, dando vita a sessioni, gruppi e threads. I process groups consentono una migliore gestione dei segnali e della comunicazione tra i processi. Un processo, per l'appunto, può:

- Aspettare che tutti i processi figli appartenenti ad un determinato gruppo terminino;
- Mandare un segnale a tutti i processi appartenenti ad un determinato gruppo.

Esempio di utilizzo dei gruppi:

```
1     waitpid(-33,NULL,0); // Wait for a children in group 33
2     kill(-33,SIGTERM); // Send SIGTERM to all children in group 33
```

Mentre, generalmente, una sessione è collegata ad un terminale, i processi vengono raggruppati nel seguente modo:

- In bash, processi concatenati tramite pipes appartengono allo stesso gruppo:

  ```
  cat /tmp/ciao.txt | wc -l | grep '2'
  ```

- Alla loro creazione, i figli di un processo ereditano il gruppo del padre
- Inizialmente, tutti i processi appartengono al gruppo di 'init', ed ogni processo può cambiare il suo gruppo in qualunque momento.

Il processo il cui PID è uguale al proprio GID è detto process group leader.

3.2 Group System Calls

`int setpgid(pid_t pid, pid_t pgid); //set GID of proc.  (0=self)` MANPAGE
La funzione setpgid() deve unirsi a un processo esistente o creare un nuovo gruppo di processi all'interno della sessione di processo di chiamata.

L'ID del gruppo di processi di un leader di sessione non deve cambiare.

In caso di completamento con esito positivo, l'ID del gruppo di processi che corrisponde a pid deve essere impostato su pgid.

Dopo il completamento con successo, setpgid() restituirà 0; altrimenti, -1 e errno deve essere impostato per indicare l'errore.

`pid_t getpgid(pid_t pid); // get GID of process (0=self)` MANPAGE
La funzione getpgid() restituirà l'ID del gruppo di processi del file processo il cui ID processo è uguale a pid. Se pid è uguale a 0, getpgid() restituirà l'ID del gruppo di processi della chiamata processi.

Dopo il completamento con successo, getpgid() restituirà il process ID del gruppo. Altrimenti, restituirà (pid_t) -1 e imposterà errno per indicare l'errore.

Esempio 30:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h> // setpgid . c
4  void main()
5  {
6              //printf("%d FatherID %d\n", getpid(), getpgid(0));
7      int isChild = !fork(); // new child
8              //printf("%d isChild %d\n", isChild, getpid());
9      printf("PID %d GID %d\n", getpid(), getpgid(0));
10     if (isChild)
11     {
```

```
12          isChild = !fork(); // new child
13              //printf("%d isChildNew %d\n", isChild, getpid());
14          if (!isChild)
15              setpgid(0, getpid()); // Become group leader
16          fork();                   // new child
17          printf("PID %d GID %d\n", getpid(), getpgid(0));
18      }
19              //printf("waiting\n");
20      while (wait(NULL) > 0);
21  }
```

Esempio 31, Mandare segnali ai gruppi:

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <sys/wait.h>
4   #include <signal.h>
5   #include <stdlib.h> //gsignal.c
6   void handler(int signo)
7   {
8       printf("[%d,%d] sig %d received\n", getpid(), getpgid(0), signo);
9       sleep(1);
10      exit(0);
11  }
12
13  void main()
14  {
15      signal(SIGUSR1, handler);
16      signal(SIGUSR2, handler);
17      int ancestor = getpid();
18      int group1 = fork();
19      int group2;
20      if (getpid() != ancestor)
21      {                           // First child
22          setpgid(0, getpid()); // Become group leader
23          fork();
24          fork(); // Generated 4 children in new group
25      }
26      else
27      {
28          group2 = fork();
29          if (getpid() != ancestor)
30          {                           // Second child
31              setpgid(0, getpid()); // Become group leader
32              fork();
33              fork();
34          }
35      } // Generated 4 children in new group
36
37      if (getpid() == ancestor)
38      {
39          printf("[%d ] Ancestor and I'll send signals \n", getpid());
40          sleep(1);
41          kill(-group2, SIGUSR1); // Send signals to group2
42          kill(-group1, SIGUSR2); // Send signals to group1
43      }
44      else
45      {
46          printf("[%d,%d] chld waiting signal \n", getpid(), getpgid(0));
47          while (1);
48      }
49      while (wait(NULL) > 0);
```

```
50      printf("All children terminated\n");
51  }
```

1. Processo 'ancestor' crea un figlioa.Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)b.I 4 processi aspettano fino all'arrivo di un segnale
2. Processo 'ancestor' crea un secondo figlioa.Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)b.I 4 processi aspettano fino all'arrivo di un segnale
3. Processo 'ancestor' manda due segnali diversi ai due gruppi

Esempio 32, Wait figli in un gruppo:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h> // waitgroup . c
4  void main()
5  {
6      int group1 = fork();
7      int group2;
8      if (group1 == 0)
9      {                           // First child
10         setpgid(0, getpid()); // Become group leader
11         fork();
12         fork(); // Generated 4 children in new group
13         sleep(2);
14         return; // Wait 2 sec and exit
15     }
16     else
17     {
18         group2 = fork();
19         if (group2 == 0)
20         {
21             setpgid(0, getpid()); // Become group leader
22             fork();
23             fork(); // Generated 4 children
24             sleep(4);
25             return; // Wait 4 sec and exit
26         }
27         sleep(1); // make sure the children changed their group
28         while (waitpid(-group1, NULL, 0) > 0);
29         printf("Children in % d terminated \n", group1);
30         while (waitpid(-group2, NULL, 0) > 0);
31         printf("Children in % d terminated \n", group2);
32     }
33 }
```

more info

4 Pipe e FIFO

4.1 Error in C

Durante l'esecuzione di un programma ci possono essere diversi tipi di errori:

- system calls che falliscono
- divisioni per zero
- problemi di memoria
- ...

Alcuni di questi errori non fatali, come una system call che fallisce, possono essere indagati attraverso la variabile `errno`. Questa variabile globale contiene l'ultimo codice di errore generato dal sistema.
Libreria utilizzata: `errno.h` - DOC1
Usiamo le funzioni:

- `char *strerror(int errnum)` converte il codice di errore in una stringa comprensibile
- `void perror(const char *str)` stampa su stderr la stringa passatagli come argomento e vi prepende l'output di strerror

La libreria presenta anche delle variabili globali che rappresentano il tipo d'errore:

- **EDOM** - rappresenta un errore di dominio, che si verifica se un argomento di input è esterno al dominio (esempio 1)
- **ERANGE** - rappresenta un errore di intervallo, che si verifica se un argomento di input è al di fuori dell'intervallo (esempio 2)
- altre macro (IBM)

Esempio 33 - EDOM

```c
#include <stdio.h>
#include <errno.h>
#include <math.h>
int main()
{
    double val;
    errno = 0;
    val = sqrt(-10);
    if (errno == EDOM)
    {
        printf("Invalid value \n");
    }
    else
    {
        printf("Valid value \n");
    }
    errno = 0;
    val = sqrt(10);
    if (errno == EDOM)
    {
        printf("Invalid value \n");
    }
    else
    {
        printf("Valid value \n");
    }
    return (0);
}
/*
```

```
OUTPUT :
Invalid value
Valid value
*/
```

Esempio 34 - ERANGE

```
#include <stdio.h>
#include <errno.h>
#include <math.h>
int main()
{
    double x;
    double value;
    x = 2.000000;
    value = log(x);
    if (errno == ERANGE)
    {
        printf("Log (%f) is out of range \n", x);
    }
    else
    {
        printf("Log (%f) = % f \n", x, value);
    }
    x = 1.000000;
    value = log(x);
    if (errno == ERANGE)
    {
        printf("Log (%f) is out of range \n", x);
    }
    else
    {
        printf(" Log (%f) = %f \n", x, value);
    }
    x = 0.000000;
    value = log(x);
    if (errno == ERANGE)
    {
        printf(" Log (% f ) is out of range \n", x);
    }
    else
    {
        printf(" Log (%f) = %f \n", x, value);
    }
    return 0;
}
/*
OUTPUT :
Log (2.000000) = 0.693147
Log (1.000000) = 0.000000
Log (0.000000) is out of range
*/
```

Esempio 35: errore apertura file

```
#include <stdio.h>
#include <errno.h>
#include <string.h> // errFile . c
extern int errno;                               // declare external global variable
void main()
{
```

```
7       FILE *pf;
8       pf = fopen("nonExistingFile.boh", "rb"); // Try to open file
9       if (pf == NULL)
10      {
11          fprintf(stderr, "errno = %d \n", errno);
12          perror("Error printed by perror ");
13          fprintf(stderr, "Strerror : %s \n", strerror(errno));
14      }
15      else
16      {
17          fclose(pf);
18      }
19  }
```

```
1   #include <stdio.h>
2   #include <errno.h>
3   #include <string.h>
4   #include <signal.h> // errSig . c
5   extern int errno; // declare external global variable
6   void main()
7   {
8       int sys = kill(3443, SIGUSR1); // Send signal to non existing proc
9       if (sys == -1)
10      {
11          fprintf(stderr, "errno = %d \n", errno);
12          perror("Error printed by perror ");
13          fprintf(stderr, "Strerror : %s \n", strerror(errno));
14      }
15      else
16      {
17          printf("Signal sent \n");
18      }
19  }
```

4.2 Piping (def)

Una pipeline o data pipeline, è un insieme di elementi di elaborazione dati collegati in serie, dove l'output di un elemento è l'input di quello successivo. Gli elementi di una pipeline vengono spesso eseguiti in parallelo o in modo suddiviso nel tempo.

Si possono utilizzare le Pipe concatenando programmi tramite || in fase di chiamata.

Esempio 37: utilizzo di una pipe

```
1   // output . out
2   #include <stdio.h>
3   #include <unistd.h>
4   void main()
5   {
6       for (int i = 0; i < 3; i++)
7       {
8           sleep(2);
9           fprintf(stdout, "Written in buffer");
10          fflush(stdout);
11      };
12  };
```

```
1   // input . out
2   #include <stdio.h>
3   #include <unistd.h>
4   void main()
```

```
{
    char msg[50];
    int n = 3;
    while ((n--) > 0)
    {
        int c = read(0, msg, 50);
        if (c > 0)
        {
            msg[c] = 0;
            fprintf(stdout, "Read : '%s' (%d)\n", msg, c);
        };
    };
};
// $ ./output.out | ./input.out
```

4.3 Pipe anonime

Una pipe anonima è un canale di **comunicazione FIFO** che può essere utilizzato per la comunicazione interprocesso (IPC) **unidirezionale**. Un'implementazione è spesso integrata nel sottosistema di I / O file del sistema operativo.

In genere un programma padre apre pipe anonime e crea un nuovo processo che eredita le altre estremità delle pipe o crea diversi nuovi processi e li dispone in una pipeline. Il collegamento avviene utilizzando **file descriptors** (motivo per cui serve l'antenato comune).

Libreria utilizzata: `unistd.h`

4.3.1 Creazione e scrittura di pipe - pipe()

Funzione: `int pipe (int pipefd[2])`

La funzione pipe crea una pipe e inserisce i descrittori di file per le estremità di lettura e scrittura della pipe (rispettivamente) in pipefd[0] e pipefd[1] (corrispondono a stdin e stdout).

In caso di esito positivo, la pipe restituisce 0; in caso di errore, restituisce -1. I codici di errore `errno` sono:

- **EMFILE** Il processo ha troppi file aperti.
- **ENFILE** Sono presenti troppi file aperti nell'intero sistema.

Esempio 38: Creazione pipe

```
#include <stdio.h>// pipe . c
#include <unistd.h>
void main()
{
    int fd[2];
    int esito = pipe(fd); // Create unnamed pipe
    if (esito == 0)
    {
        write(fd[1], "writing", 8); // Write to pipe using fd [1]
        char buf[50];
        int c = read(fd[0], &buf, 50); // Read from pipe using fd [0]
        printf("Read '%s' (%d)\n", buf, c);
    }
}
```

4.3.2 Lettura di pipe - read()

`int read(int fd[0],char * data, int num)`

Tenta di leggere fino al conteggio dei byte dal descrittore di file fd nel buffer a partire da buf. Sui file che supportano la ricerca, l'operazione di lettura inizia dall'offset del file.

La lettura della pipe tramite il comando read restituisce valori differenti a seconda della situazione:

- In caso di successo restituisce il numero di bytes effettivamente letti
- Se il lato di scrittura è stato chiuso (da ogni processo) ed il buffer è vuoto restituisce 0
- Se il buffer è vuoto ma il lato di scrittura è ancora aperto (in qualche processo) il processo si sospende fino alla disponibilità dei dati o alla chiusura
- Se si provano a leggere più bytes (num) di quelli disponibili, vengono recuperati solo quelli presenti
- In caso di errore, viene restituito -1 e viene impostato `errno`.

MANPAGE

Esempio 39: lettura pipe

```
1  #include <stdio.h> // readPipe . c
2  #include <unistd.h>
3  void main()
4  {
5      int fd[2];
6      char buf[50];
7      int esito = pipe(fd); // Create unnamed pipe
8      if (esito == 0)
9      {
10         write(fd[1], "writing", 8);  // Writes to pipe
11         int r = read(fd[0], &buf, 50); // Read from pipe
12         printf("Last read %d. Received : '%s'\n", r, buf);
13         // close ( fd [1]); // hangs when commented
14         r = read(fd[0], &buf, 50); // Read from pipe
15         printf("Last read %d. Received : '%s'\n", r, buf);
16     }
17 }
```

4.3.3 Lettura pipe - write()

`int write(int fd[0],char * data, int num)`

La funzione `write()` scrive i dati da un buffer dichiarato dall'utente su un determinato dispositivo, come un file. E' il modo principale per produrre dati da un programma utilizzando direttamente una chiamata di sistema. La destinazione è identificata da un codice numerico. I dati da scrivere, ad esempio un pezzo di testo, sono definiti da un puntatore e da una dimensione, espressa in numero di byte. WIKI

Argument	Description
fd	It is the file descriptor which has been obtained from the call to open. It is an integer value. The values 0, 1, 2 can also be given, for standard input, standard output & standard error, respectively .
data	It points to a character array, with content to be written to the file pointed to by fd.
num	It specifies the number of bytes to be written from the character array into the file pointed to by fd.

Table 4: Argomenti `write()`

4.4 Gestione dei segnali

`int fcntl(int fd, cmd, arg;` Letteralmente **File Control**.

Apre il descrittore di file fd. Può accettare un terzo argomento opzionale. MANPAGE

F_DUPFD	duplica il descrittore di file fd usando il numero più basso descrittore di file disponibile maggiore o uguale a arg. Questo è diverso da dup2 (2), che utilizza esattamente l'estensione descrittore di file specificato. In caso di successo, viene restituito il nuovo descrittore di file.
F_GETFD	Restituisce (come risultato della funzione) i flag del descrittore di file; arg viene ignorato.
F_SETFD	Imposta i flag del descrittore di file sul valore specificato da arg.

Table 5: Alcuni cmd

Esempio 40: comunicazione bidirezionale
Un tipico esempio di comunicazione unidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

1. P1 crea una pipe()
2. P1 esegue un fork() e crea P2
3. P1 chiude il lato lettura: close(fd[0])
4. P2 chiude il lato scrittura: close(fd[1])
5. P1 e P2 chiudono l'altro fd appena finiscono.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h> // uni . c
void main()
{
    int fd[2];
    char buf[50];
    pipe(fd); // Create unnamed pipe
    int p2 = !fork();
    if (p2)
    {
        close(fd[1]);
        int r = read(fd[0], &buf, 50); // Read from pipe
        close(fd[0]);
        printf("Buf : '%s'\n", buf);
    }
    else
    {
        close(fd[0]);
        write(fd[1], "writing", 8); // Writes to pipe
        close(fd[1]);
    }
    while (wait(NULL) > 0);
}
```

Esempio 41: bidirezionale
Un tipico esempio di comunicazione bidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

1. P1 crea duepipe(), pipe1 e pipe2
2. P1 esegue un fork() e crea P2
3. P1 chiude il lato lettura di pipe1 ed il lato scrittura di pipe2
4. P2 chiude il lato scrittura di pipe1 ed il lato lettura di pipe2
5. P1 e P2 chiudono gli altri fd appena finiscono di comunicare.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define READ 0
#define WRITE 1

void main()
{
    int pipe1[2], pipe2[2];
    char buf[50];
    pipe(pipe1);
    pipe(pipe2); // Create two unnamed pipe
    int p2 = !fork();
    if (p2)
    {
        close(pipe1[WRITE]);
        close(pipe2[READ]);
        int r = read(pipe1[READ], &buf, 50); // Read from pipe
        close(pipe1[READ]);
        printf("P2 received : '%s'\n", buf);
        write(pipe2[WRITE], "Msg from p2", 12); // Writes to pipeclose ( pipe2 [ WRITE ]);
    }
    else
    {
        close(pipe1[READ]);
        close(pipe2[1]);
        write(pipe1[WRITE], "Msg from p1", 12); // Writes to pipe
        close(pipe1[WRITE]);
        int r = read(pipe2[READ], &buf, 50); // Read from pipe
        close(pipe2[READ]);
        printf("P1 received : '%s'\n", buf);
    }
    while (wait(NULL) > 0);
}
```

4.4.1 Gestire la comunicazione

Per gestire comunicazioni complesse c'è bisogno di definire un "protocollo", ad esempio:

- Messaggi di lunghezza fissa (magari inviata prima del messaggio)
- Marcatore di fine messaggio (per esempio con carattere NULL o newline)Più in generale occorre definire la sequenza di messaggi attesi

Esempio 42:-MAYBE ERR- redirige lo stdout di cmd1 sullo stdin di cmd2

```c
#include <stdio.h> //segmentation fault (core dumped)
#include <unistd.h>
#define READ 0
#define WRITE 1 // redirect . c
int main(int argc, char *argv[])
{
    int fd[2];
    pipe(fd); // Create an unnamed pipe
    if (fork() != 0)
    {                                   // Parent , writer
        close(fd[READ]);                // Close unused end
        dup2(fd[WRITE], 1);             // Duplicate used end to stdout
        close(fd[WRITE]);               // Close original used end
        execlp(argv[1], argv[1], NULL); // Execute writer program
        perror("connect");             // Should never execute
    }
    else
```

```
18      {
19          // Child , reader
20          close(fd[WRITE]);                // Close unused end
21          dup2(fd[READ], 0);               // Duplicate used end to stdin
22          close(fd[READ]);                 // Close original usedend
23          execlp(argv[2], argv[2], NULL); // Execute reader program
24          perror("connect");              // Should never execute
25      }
26  }
```

4.5 Pipe con nome (FIFO)

Le pipe con nome, o FIFO, corrispondono a dei file speciali nel filesystem grazie ai quali i processi, senza vincoli di gerarchia, possono comunicare. Un processo può accedere ad una di queste pipe se ha i permessi sul file corrispondente ed è vincolato, ovviamente, all'esistenza del file stesso.Essendo oggetti nel file system, si possono usare le funzioni di scrittura/lettura dei file viste nelle scorse lezioni. Una volta creata una pipe con nome, il file associato è persistente!NB: al contrario di un normale file, una FIFO deve essere aperta da entrambi i lati per potervi interagire in modo ragionevole.

4.5.1 Esempio 43: Creazione Pipe

```
int mkfifo(const char *pathname, mode_t mode);
```

```
1   #include <sys/stat.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4   #include <fcntl.h>
5   #include <stdio.h> //fifo.c
6   void main()
7   {
8       char *fifoName = "/tmp/fifo1";
9       mkfifo(fifoName, S_IRUSR | S_IWUSR);
10      // Create pipe if doesn 't exist
11      perror("Created?");
12      if (fork() == 0)
13      {
14          open(fifoName, O_RDONLY); // Open pipe in read only ... stuck !
15          printf("Open read \n");
16      }
17      else
18      {
19          sleep(1);
20          open(fifoName, O_WRONLY); // Open pipe in write only
21          printf("Open write \n");
22      }
23  }
```

Esempio 44: writer

```
1   #include <sys/stat.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4   #include <fcntl.h>
5   #include <stdio.h>
6   #include <string.h> // fifoWriter . c
7       void main(int argc, char *argv[])
8   {
9       int fd;
10      char *fifoName = "/tmp/fifo1";
11      char str1[80], *str2 = "I'm a writer";
12      mkfifo(fifoName, S_IRUSR | S_IWUSR); // Create pipe if doesn 't exist
```

```
13      fd = open(fifoName, O_WRONLY);        // Open FIFO for write only
14      write(fd, str2, strlen(str2) + 1);    // write and close
15      close(fd);
16      fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
17      read(fd, str1, sizeof(str1));  // Read from FIFO
18      printf("Reader is writing : %s \n", str1);
19      close(fd);
20  }
```

Esempio 45: reader

```
1   #include <sys/stat.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4   #include <fcntl.h>
5   #include <stdio.h>
6   #include <string.h> // fifoWriter . c
7       void main(int argc, char *argv[])
8   {
9       int fd;
10      char *fifoName = "/tmp/fifo1";
11      char str1[80], *str2 = "I'm a writer";
12      mkfifo(fifoName, S_IRUSR | S_IWUSR); // Create pipe if doesn 't exist
13      fd = open(fifoName, O_WRONLY);        // Open FIFO for write only
14      write(fd, str2, strlen(str2) + 1);    // write and close
15      close(fd);
16      fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
17      read(fd, str1, sizeof(str1));  // Read from FIFO
18      printf("Reader is writing : %s \n", str1);
19      close(fd);
20  }
```

5 Message Queue and Threads

Una coda di messaggi, message queue, è una lista concatenata memorizzata all'interno del kernel ed identificata con una chiave (un intero positivo univoco), chiamata queue identifier. Questa chiave viene condivisa tra i processi interessati, i quali generano degli ulteriori identificativi da usare durante l'interazione con la coda.Una coda deve essere innanzitutto generata in maniera analoga ad una FIFO, impostando dei permessi. Ad una coda esistente si possono aggiungere o recuperare messaggi tipicamente in modalità "autosincrona": la lettura attende la presenza di un messaggio, la scrittura attende che via sia spazio disponibile. Questi comportamenti possono però essere configurati.

5.1 Queues

5.1.1 Struttura del messaggio

Ogni messaggio inserito nella coda ha tre campi:

- Un tipo (intero "long")
- Una lunghezza non negativa
- Un insieme di dati (bytes) di lunghezza corretta

Al contrario delle FIFO, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine "assoluto" di arrivo. Così come i files, le code sono delle strutture persistenti che continuano ad esistere, assieme ai messaggi in esse salvati, anche alla terminazione del processo che le ha create. L'eliminazione deve essere esplicita.

```
1 struct msg_buffer{
2     long mtype;
3     char mtext[100];
4 } message;
```

IBM - MANPAGE

5.1.2 Creazione coda

Funzione `int msgget(key_t key, int msgflg)`

MANPAGE Restituisce l'identificativo di una coda basandosi sulla chiave "key" e sui flags:

- **IPC_CREAT**: crea una coda se non esiste già, altrimenti restituisce l'identificativo di quella già esistente
- **IPC_EXCL**: (da usare assieme al precedente) fallisce se coda già esistente
- **0xxx**: numero ottale di permessi, analogo a quello che si può usare nel file system

In caso di successo, il valore restituito sarà l'identificatore della coda dei messaggi (un numero intero non negativo), altrimenti -1 con errno che indica l'errore.

```
1 #include <sys/types.h> <sys/ipc.h> <sys/msg.h> /msgget.c
2 key_t queueKey = 56; //Unique key
3 int queueId = msgget(queueKey, 0777 | IPC_CREAT | IPC_EXCL);
```

5.1.3 Ottenere una chiave univoca

Funzione `key_t ftok(const char *pathname, int proj_id)` .

MANPAGE Restituisce una chiave basandosi sul path (una cartella o un file), esistente ed accessibile nel file-system, e sull'id numerico. La chiave dovrebbe essere univoca e sempre la stessa per ogni coppia $< path, id >$ in ogni istante sullo stesso sistema.Un metodo d'uso, per evitare possibili conflitti, potrebbe essere generare un path (es. un file) temporaneo univoco, usarlo, eventualmente rimuoverlo, ed usare l'id per per rappresentare diverse "categorie" di code, a mo' di indice.

`key_t ftok(const char *path, int id)`

```c
#include <sys/ipc.h> //ftok.c
key_t queue1Key = ftok("/tmp/unique", 1);
key_t queue2Key = ftok("/tmp/unique", 2); ...
```

Esempio 46: creazione

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h> // ipcCreation . c
void main()
{
    remove("/tmp/unique");                          // Remove file
    key_t queue1Key = ftok("/tmp/unique", 1);       // Get unique key -> fail
    creat("/tmp/unique", 0777);                     // Create file
    queue1Key = ftok("/tmp/unique", 1);             // Get unique key -> ok
    int queueId = msgget(queue1Key, 0777 | IPC_CREAT); // Create queue -> ok
    queueId = msgget(queue1Key, 0777);                 // Get queue -> ok
    msgctl(queue1Key, IPC_RMID, NULL);                 // Remove non existing queue -> fail
    msgctl(queueId, IPC_RMID, NULL);                   // Remove queue -> ok
    queueId = msgget(queue1Key, 0777);                 // Get non existing queue -> fail
    queueId = msgget(queue1Key, 0777 | IPC_CREAT);     // Create queue -> ok
    queueId = msgget(queue1Key, 0777 | IPC_CREAT);     // Get queue -> ok
    queueId = msgget(queue1Key, 0777 | IPC_CREAT | IPC_EXCL);
    /* Create already existing queue*/
}
```

5.1.4 Persistenza delle code

Se eseguiamo questo programma dopo aver eseguito il precedente *ipcCreation.c* verrà generato un errore dato che la coda esiste già ed abbiamo usato il flag **IPC_EXCL**!

Esempio 47: le queue sono persistenti

```c
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h> // persistent . c
void main()
{
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key, 0777 | IPC_CREAT | IPC_EXCL);
    perror("Error :");
}
```

5.1.5 Inviare messaggi

```c
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

LINUX.DIE Aggiunge un messaggio di dimensione msgsz alla coda identificata da msquid, puntata dal buffer msgp. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se msgflg è **IPC_NOWAIT** allora la chiamata fallisce in assenza di spazio.

5.1.6 Ricevere messaggi

```c
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

LINUX.DIE Rimuove un messaggio dalla coda msqid e lo salva nel buffer msgp. msgsz specifica la lunghezza massima del testo del messaggio (mtext della struttura msgp). Se il messaggio ha una

lunghezza maggiore e *msgflg* è **MSG_NOERROR** allora il messaggio viene troncato (viene persa la parte in eccesso), se **MSG_NOERROR** non è specificato allora il messaggio non viene eliminato e la chiamata fallisce. A seconda di *msgtyp* viene recuperato il messaggio:

- *msgtyp* = 0: primo messaggio della coda
- *msgtyp* > 0: primo messaggio di tipo *msgtyp*, o primo messaggio di tipo diverso da *msgtyp* se **MSG_EXCEPT** è impostato come flag
- *msgtyp* < 0: primo messaggio il cui tipo T è `min(T<=|msgtyp|)`. In fine, il flag **IPC_NOWAIT** fa fallire la syscall se non sono presenti messaggi (altrimenti hang)

Esempio 48: comunicazione

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h> // ipc . c
struct msg_buffer
{
    long mtype;
    char mtext[100];
} msgp, msgp2;
void main()
{
    msgp.mtype = 20;
    strcpy(msgp.mtext, "This is a message");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key, 0777 | IPC_CREAT | IPC_EXCL);
    int esito = msgsnd(queueId, &msgp, sizeof(msgp.mtext), 0);
    esito = msgrcv(queueId, &msgp2, sizeof(msgp2.mtext), 20, 0);
}
```

5.1.7 Modificare la coda

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

MANPAGE Modifica la coda identificata da *msqid* secondo i comandi cmd, riempiendo buf con informazioni sulla coda (ad esempio tempo di ultima scrittura, di ultima lettura, numero messaggi nella coda, etc...). I valori per cmd sono:

- **IPC_STAT**: recupera informazioni da kernel
- **IPC_SET**: imposta alcuni parametri a seconda di buf
- **IPC_RMID**: rimuove immediatamente la coda
- **IPC_INFO**: recupera informazioni generali sui limiti delle code nel sistema
- **MSG_INFO**: come **IPC_INFO** ma con informazioni differenti
- **MSG_STAT**: come **IPC_STAT** ma con informazioni differenti

msqid_ds structure

```c
struct msqid_ds {
  struct ipc_perm msg_perm; /* Ownership and permissions */
  time_t msg_stime; /* Time of last msgsnd(2) */
  time_t msg_rtime; /* Time of last msgrcv(2) */
  time_t msg_ctime; //Time of creation or last modification by msgctl
  unsigned long msg_cbytes; /* # of bytes in queue */
  msgqnum_t msg_qnum; /* # of messages in queue */
```

```
  msglen_t msg_qbytes; /* Maximum # of bytes in queue */
  pid_t msg_lspid; /* PID of last msgsnd(2) */
  pid_t msg_lrpid; /* PID of last msgrcv(2) */
};
```

ipc_perm structure

```
 struct ipc_perm {
   key_t __key; /* Key supplied to msgget(2) */
   uid_t uid; /* Effective UID of owner */
   gid_t gid; /* Effective GID of owner */
   uid_t cuid; /* Effective UID of creator */
   gid_t cgid; /* Effective GID of creator */
   unsigned shortmode; /* Permissions */
   unsigned short __seq; /* Sequence number */
 };
```

Esempio 49: modifica

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h> // modifica . c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgpSND, msgpRCV;

void main(){
    struct msqid_ds mod;
    msgpSND.mtype = 1;
    strcpy(msgpSND.mtext, "This is a message from sender");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key, 0777 | IPC_CREAT);
    msgctl(queueId, IPC_RMID, NULL);                    // Remove queue if exists
    queueId = msgget(queue1Key, 0777 | IPC_CREAT);      // Create queue
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext), 0); // Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext), 0); // Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext), 0); // Send msg
    msgctl(queueId, IPC_STAT, &mod);                    // Modify queue
printf ( "Msg in queue : %ld \n Current max bytes in queue : %ld\n\n" , mod . msg_qnum , mod
mod . msg_qbytes = 200; // Change buf to modify queue bytes
msgctl ( queueId , IPC_SET , & mod ); // Apply modification
printf ( "Msg in queue : %ld --> same number \nCurrent max bytes in queue : %ld\n\n" , mod .
if ( fork () != 0) {
        // Parent keep on writing on the queue
        printf("[ SND ] Sending third message with a full queue ...\n");
        msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext), 0); // Send msg
        printf("[ SND ] msg sent \n");
        printf("[ SND ] Sending fourth message again with IPC_NOWAIT \n");
        if (msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext), IPC_NOWAIT) == -1)
        {
            // Send
            msgperror("Queue is full --> Error");
        }
} else {
        // Child keeps reading the queue every 3 seconds
        sleep(3);
```

```
43          msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext), 1, 0);
44          printf("[ Reader ] Received msg 1 with msg '%s'\n", msgpRCV.mtext);
45          sleep(3);
46          msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext), 1, 0);
47          printf("[ Reader ] Received msg 2 with msg '%s'\n", msgpRCV.mtext);
48          sleep(3);
49          msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext), 1, 0);
50          printf("[ Reader ] Received msg 3 with msg '%s'\n", msgpRCV.mtext);
51          sleep(3);
52          msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext), 1, 0);
53          printf("[ Reader ] Received msg 4 with msg '%s'\n", msgpRCV.mtext);
54          printf("[ Reader ] Received msg 5 with msg '%s'\n", msgpRCV.mtext);
55          sleep(3);
56          if (msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext), 1, IPC_NOWAIT) == -1)
57          {
58              perror("Queue is empty --> Error");
59          }
60          else
61          {
62              printf("[ Reader ] Received msg 6 with msg '%s'\n", msgpRCV.mtext);
63          }
64  }
65  while ( wait ( NULL ) > 0);
66  }
```

5.2 Threads

I thread sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. Non sono indipendenti tra loro: condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo hanno alcuni elementi indipendenti, come lo stack, il PC ed i registri del sistema. La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra threads è rapido, così come la loro creazione, terminazione e comunicazione.

Per la compilazione è necessario aggiungere il flag `-pthread`
ad esempio:textttgcc -o program main.c -pthread

5.2.1 Creazione

In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è identificato da un ID e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione.

```
1  int pthread_create(
2      pthread_t *restrict thread, /* Thread ID */
3      const pthread_attr_t *restrict attr, /* Attributes */
4      void *(*start_routine)(void *), /* Function to be executed */
5      void *restrict arg /* Parameters to above function */
6  );
```

Esempio 50: Creazione

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h> // threadCreate . c
4  void *my_fun(void *param)
5  {
6      printf("This is a thread that received %d\n", *(int *)param);
7      return (void *)3;
8  }
9  void main()
10 {
```

```
11      pthread_t t_id;
12      int arg = 10;
13      pthread_create(&t_id, NULL, my_fun, (void *)&arg);
14      printf("Executed thread with id %ld\n", t_id);
15      sleep(3);
16  }
```

5.2.2 Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione noreturn void pthread_exit(void *retval); specificando un puntatore di ritorno.
- Ritorna dalla funziona associata al thread specificando un valore di ritorno.
- Viene cancellato
- Qualche thread chiama exit(), o il thread che esegue main() ritorna dallo stesso, terminando così tutti i threads.

5.2.3 Cancellazione Thread

`int pthread_cancel(pthread_t thread);`

Invia una richiesta di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda di due suoi attributi: state e type. State può essere enabled(default) o disabled: se disabled la richiesta rimarrà in attesa fino a che state diventa enabled, se enabled la cancellazione avverrà a seconda di type. Type può essere deferred (default) o asynchronous: attende la chiamata di un cancellation point o termina in qualsiasi momento, rispettivamente. Cancellation points sono funzioni definite nella libreria pthread.h (lista). State e typepossono essere modificati:

`int pthread_setcancelstate(int state, int *oldstate);`
Con state = PTHREAD_CANCEL_DISABLE o PTHREAD_CANCEL_ENABLE

`int pthread_setcanceltype(int type, int *oldtype);`
Con type = PTHREAD_CANCEL_DEFERRED o PTHREAD_CANCEL_ASYNCHRONOUS

Esempio 51: Creazione e cancellazione

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h> // thCancel . c
4  int i = 1;
5  void * my_fun ( void * param ){
6  if (i - -) pthread_ set cancel state ( PTHREAD_CANCEL_DISABLE , NULL ); // Change m o d e p t
7  printf ( "Thread %ld started \n" ,*( pthread_t *) param );
8  sleep (3);
9  printf ( "Thread %ld finished\n" ,*( pthread_t *) param );
10 }
11 void main (){
12 pthread_t t_id1 , t_id2 ;
13 pthread_create (& t_id1 , NULL , my_fun , ( void *)& t_id1 );
14 sleep (1); // Create
15 pthread_cancel ( t_id1 ); // Cancel
16 printf ( "Sent cancellation request for thread %ld \n" , t_id1 );
17 pthread_create (& t_id2 , NULL , my_fun , ( void *)& t_id2 );
18 sleep (1); // Create
19 pthread_cancel ( t_id2 ); // Cancel
20 printf ( "Sent cancellation request for thread % ld \n" , t_id2 );
21 sleep (5);
```

```
22 printf ( "Terminating program \n" );
23 }
```

5.2.4 Aspettare un Thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:
`int pthread_join(pthread_t thread, void **retval);`
Che ritorna quando il thread identificato da thread termina, o subito se il thread è già terminato. Se il valore di ritorno del thread non è nullo (parametro di pthread_exit o di return), esso viene salvato nella variabile puntata da retval. Se il thread era stato cancellato, retval è riempito con PTHREAD_CANCELED.Solo se il thread è joinable può essere aspettato! Un thread può essere aspettato da al massimo un thread!

Esempio 52: Join 1

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h> // thJoin . c
4  void *my_fun(void *param){
5      printf("Thread % ld started \n", *(pthread_t *)param);
6      sleep(3);
7      char *str = "Returned string";
8      pthread_exit((void *)str); // or ' return ( void *) str ; '
9  }
10 void main()
11 {
12     pthread_t t_id;
13     void *retFromThread;                                  // This must be a pointer to void !
14     pthread_create(&t_id, NULL, my_fun, (void *)&t_id); // Create
15     pthread_join(t_id, &retFromThread);                   // wait thread
16     // We must cast the returned value !
17     printf("Thread %ld returned '%s'\n", t_id, (char *)retFromThread);
18 }
```

Esempio 53: Join 2

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h> // threadJoin . c ( v . esempio threadCreate . c )
4  void *my_fun(void *param){
5      printf("This is a thread that received %d \n", *(int *)param);
6      return (void *)3;
7  }
8
9  void main()
10 {
11     pthread_t t_id;
12     int arg = 10, retval;
13     pthread_create(&t_id, NULL, my_fun, (void *)&arg);
14     printf("Executed thread with id %ld\n", t_id);
15     sleep(3);
16     pthread_join(t_id, (void **)&retval);
17     printf("retval =%d\n", retval);
18 }
```

Esempio 54: Join 3

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h> // thJoin . c
```

```
4  void *my_fun(void *param){
5      sleep(2);
6  }
7
8  void *my_fun2(void *param){
9      if (pthread_join(*(pthread_t *)param, NULL) != 0)
10         printf("Error\n");
11 }
12
13 void main(){
14     pthread_t t_id, t_id2;
15     pthread_create(&t_id, NULL, my_fun, NULL);           // Create
16     pthread_create(&t_id2, NULL, my_fun, (void *)&t_id); // Create
17     pthread_join(t_id, NULL);                            // wait thread
18     sleep(1);
19     perror();
20 }
```

Ogni thread viene creato con degli attributi specificati nella struttura pthread_attr_t. Questa struttura, analogamente alla struttura usata per gestire le maschere dei segnali, è un oggetto usato solo alla creazione di un thread, ed è poi indipendente dallo stesso (se cambia, gli attributi del thread non cambiano). La struttura va inizializzata con int pthread_attr_init(pthread_attr_t *attr); che imposta tutti gli attributi al loro valore di default. Una volta usata e non più necessaria, la struttura va distrutta con int pthread_attr_destroy(pthread_attr_t *attr);I vari attributi della struct possono, e devono, essere modificati singolarmente con le seguenti funzioni:int pthread_attr_setxxxx(pthread_attr_t *attr, params);int pthread_attr_getxxxx(const pthread_attr_t *attr, params);

- ...detachstate(pthread_attr_t *attr, int detachstate)
  - PTHREAD_CREATE_DETACHED -¿ non può essere aspettato
  - PTHREAD_CREATE_JOINABLE -¿ default, può essere aspettato
  - Può essere cambiato durante l'esecuzione con int pthread_detach(pthread_t thread);
- ...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);
- ...affinity_np(...)
- ...setguardsize(...)
- ...inheritsched(...)
- ...schedparam(...)
- ...schedpolicy(...)
- altri

5.2.5 Detached e joinable threads

I threads vengono creati di default nello stato joinable, il che consente ad un altro thread di attendere la loro terminazione attraverso il comando pthread_join. I thread joinable rilasciano le proprie risorse non alla terminazione ma quando un thread fa il join con loro (salvando lo stato di uscita) (così come i sottoprocessi), oppure alla terminazione del processo. Contrariamente, i thread in stato detached liberano le loro risorse immediatamente una volta terminati, ma non consentono ad altri processi di fare il "join".NB: un thread detached non può diventare joinable durante la sua esecuzione, mentre il contrario è possibile.

Esempio 55: Attributi

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #include <errno.h> // threadAttr . c
```

```
5  void *my_fun(void *param){
6      printf("This is a thread that received %d \n", *(int *)param);
7      sleep(3);
8      return (void *)3;
9  }
10
11 void main(){
12     pthread_t t_id;
13     pthread_attr_t attr;
14     int arg = 10, detachSTate;
15     pthread_attr_init(&attr);
16     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // Set detached
17     pthread_attr_getdetachstate(&attr, &detachSTate);                          // Get detach
18     if (detachSTate == PTHREAD_CREATE_DETACHED)
19         printf("Detached\n");
20     pthread_create(&t_id, &attr, my_fun, (void *)&arg);
21     printf("Executed thread with id %ld \n", t_id);
22     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); // Inneffective
23     sleep(3);
24     int esito = pthread_join(t_id, (void **)&detachSTate);
25     printf("Esito '%d' is different from 0\n", esito); // EINVAL
26     pthread_attr_destroy(&attr);
27 }
```

6 Esercizi

6.1 Cap1

Scrivere dei programmi in C che:

- Avendo come argomenti dei "binari", si eseguono con exec ciascuno in un sottoprocesso (*)
- idem punto 1 ma in più salvando i flussi di stdout e stderr in un unico file (*)
- Dati due eseguibili come argomenti del tipo ls e wc si eseguono in due processi distinti: il primo deve generare uno stdout redirezionato su un file temporaneo, mentre il secondo deve essere lanciato solo quando il primo ha finito leggendo lo stesso file come stdin.Ad esempio ./main ls wc deve avere lo stesso effetto di ls | wc.

Suggerimenti: anziché due figli usare padre-figlio- usare dup2 per far puntare il file descriptor del file temporaneo su stdout in un processo e stdin nell'altro sfruttare wait per attendere la conclusione del processo che genera l'output
(*) generando DUE figli.

7 Link utili

- libreria unistd.h
- signal.h da DOC Ubuntu
- esempi di signal in C -- da implementare nel doc
- signal.h in C
- codici segnali