Programmazione di Sistema in UNIX

Nicola Bombieri - Nicola Drago - Graziano Pravadelli

Dipartimento di Informatica
Università di Verona

Sommario

- Interfaccia tramite system call
- System call: gestione di processi
- System call: gestione del file system
- Inter-process communication (IPC)

Interfaccia tramite system call

- L'accesso al kernel è permesso soltanto tramite le system call, che permettono di passare all'esecuzione in modo kernel.
- Dal punto di vista dell'utente, l'interfaccia tramite system call funziona come una normale chiamata C.
- In realtà più complicato:
  - Esiste una *system call library* contenente funzioni con lo stesso nome della system call
  - Le funzioni di libreria cambiano il modo user in modo kernel e fanno sì che il kernel esegua il vero e proprio codice delle system call
  - La funzione di libreria passa un identificatore, unico, al kernel, che identifica una precisa system call.
  - Simile a una routine di interrupt (detta *operating system trap*)

System Call

Classe	System Call	
File	`creat()`	`open()`
	`close()`	`read()`
	`write()`	`creat()`
	`lseek()`	`dup()`
	`link()`	`unlink()`
	`stat()`	`fstat()`
	`chmod()`	`chown()`
	`umask()`	`ioctl()`
Processi	`fork()`	`exec()`
	`wait()`	`exit()`
	`signal()`	`kill()`
	`getpid()`	`getppid()`
	`alarm()`	`chdir()`
Comunicazione tra processi	`pipe()`	`msgget()`
	`msgctl()`	`msgrcv()`
	`msgsnd()`	`semop()`
	`semget()`	`shmget()`
	`shmat()`	`shmdt()`

Efficienza delle system call

- L'utilizzo di system call è in genere meno efficiente delle (eventuali) corrispondenti chiamate di libreria C
- Particolarmente evidente nel caso di system call per il file system
  - Esempio:

```
/* PROG1 */
int main(void) {
    int c;
    while ((c = getchar()) != EOF) putchar(c);
}
/* PROG2 */
int main(void) {
    char c;
    while (read(0, &c, 1) > 0)
        if (write(1, &c, 1) != 1) perror("write"), exit(1);
}
```

    PROG1 è circa 5 volte più veloce!

Errori nelle chiamate di sistema

- In caso di errore, le system call ritornano tipicamente un valore -1, ed assegnano lo specifico codice di errore nella variabile `errno`, definita in `<errno.h>`
- Per mappare il codice di errore al tipo di errore, si utilizza la funzione

```
#include <stdio.h>
void perror (char *str)
```

  su `stderr` viene stampato:

$$str\ :\ messaggio\text{-}di\text{-}errore\ \texttt{\textbackslash n}$$

- Solitamente str è il nome del programma o della funzione.
- Per comodità definiamo una funzione di errore alternativa `syserr`, definita in un file `mylib.c`
  - Tutti i programmi descritti di seguito devono includere `mylib.h` e llinkare `mylib.o`

```
/*****************************************
MODULO: mylib.h
SCOPO: definizioni per la libreria mylib
*****************************************/

void syserr (char *prog, char *msg);
```

```
/*****************************************
MODULO: mylib.c
SCOPO: libreria di funzioni d'utilita'
*****************************************/
#include <stdio.h>
#include <errno.h>

#include "mylib.h"

void syserr (char *prog, char *msg)
{
  fprintf (stderr, "%s - errore: %s\n",prog, msg);
  perror ("system error");
  exit (1);
}
```

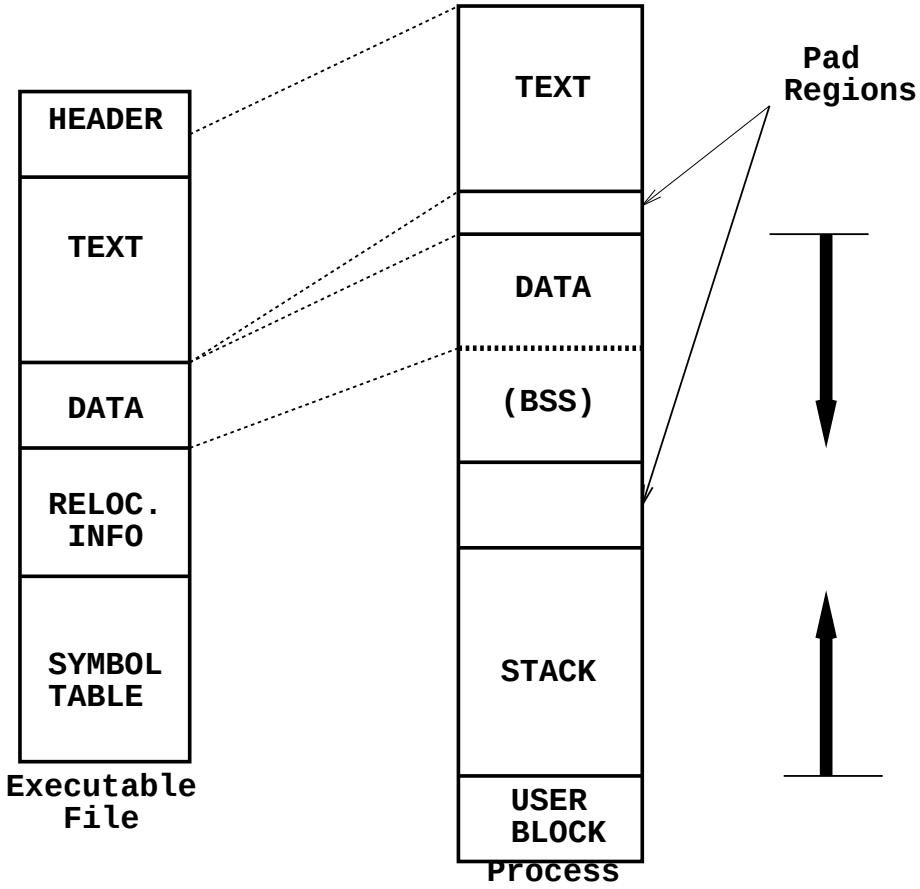
```
/*****************************************
MODULO: env.c
SCOPO: elenco delle variabili d'ambiente
*****************************************/
#include <stdio.h>

int main (int argc, char *argv[], char *env[])
{
  puts ("Variabili d'ambiente:");
  while (*env != NULL)
    puts (*env++);
  return 0;
}
```

System Call per la Gestione dei Processi

Gestione dei processi

- Come trasforma UNIX un programma eseguibile in processo (con il comando `ld`)?



Gestione dei processi – Programma eseguibile

- HEADER: definita in `/usr/include/filehdr.h`.
  - definisce la dimensione delle altre parti
  - definisce l'entry point dell'esecuzione
  - contiene il $magic\ number$, numero speciale per la trasformazione in processo (system-dependent)
- TEXT: le istruzioni del programma
- DATA: I dati inizializzati (statici, extern)
- BSS (Block Started by Symbol): I dati non inizializzati (automatici). Nella trasformazione in processo, vengono messi tutti a zero in una sezione separata.
- RELOCATION: come il loader carica il programma. Rimosso dopo il caricamento
- SYMBOL TABLE: Può essere rimossa (`ld -s`) o con `strip` (toglie anche la relocation info). Contiene informazioni quali la locazione, il tipo e lo scope di variabili, funzioni, tipi.

Gestione dei processi – Processo

- TEXT: copia di quello del programma eseguibile. Non cambia durante l'esecuzione
- DATA: possono crescere verso il basso ($heap$)
- BSS: occupa la parte bassa della sezione dati
- STACK: creato nella costruzione del processo. Contiene:
  - le variabili automatiche
  - i parametri delle procedure
  - gli argomenti del programma e le variabili d'ambiente
  - riallocato automaticamente dal sistema
  - cresce verso l'alto
- USER BLOCK: sottoinsieme delle informazioni mantenute dal sistema sul processo

Creazione di processi

```
#include <unistd.h>

pid_t fork (void)
```

- Crea un nuovo processo, figlio di quello corrente, che eredita dal padre:
  - I file aperti
  - Le variabili di ambiente
  - Tutti i settaggi dei segnali (v.dopo)
  - Directory di lavoro
- Al figlio viene ritornato `0`.
- Al padre viene ritornato il PID del figlio (o `-1` in caso di errore).
- NOTA: un processo solo chiama `fork`, ma è come se due processi ritornassero!

```
/*****************************************
MODULO: fork.c
SCOPO: esempio di creazione di un processo
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include "mylib.h"
int main (int argc, char *argv[]){
   pid_t status;
   if ((status=fork()) == -1)
   syserr (argv[0], "fork() fallita");
   if (status == 0) {
      sleep(10);
      puts ("Io sono il figlio!");
   } else {
      sleep(2);
      printf ("Io sono il padre e mio figlio ha PID=%d)\n",status);
   }
}
```

`fork` e debugging

- gdb non supporta automaticamente il debugging di programmi con `fork` $\Longrightarrow$ debugging sempre del padre
- Per debuggare il figlio:
  - Eseguire un gdb dello stesso programma da un'altra finestra
  - Usare il comando di gdb

    `attach` pid

    dove pid è il pid del figlio, determinato con `ps`
- Per garantire un minimo di sincronizzazione tra padre e figlio, è consigliato inserire una pausa condizionale all'ingresso del figlio

Esecuzione di un programma

```
#include <unistd.h>
int execl (char *file,char *arg0,char *arg1, ...,0)
int execlp(char *file,char *arg0,char *arg1, ...,0)
int execle(char *file,char *arg0,char *arg1, ...,0,char *envp[])
int execv (char *file,char *argv[])
int execvp(char *file,char *argv[])
int execve(char *file,char *argv[],char *envp[])
```

- Sostituiscono all'immagine attualmente in esecuzione quella specificata da `file`, che può essere:
  - un programma binario
  - un file di comandi
- In altri termini, `exec` trasforma un eseguibile in processo.
- NOTA: `exec` non ritorna!!

La Famiglia di `exec`

- `execl` utile quando so in anticipo il numero e gli argomenti, `execv` utile altrimenti.
- `execle` e `execve` ricevono anche come parametro la lista delle variabili d'ambiente.
- `execlp` e `execvp` utilizzano la variabile `PATH` per cercare il comando `file`.

```
/*****************************************
MODULO: exec.c
SCOPO: esempio d'uso di exec()
*****************************************/
#include <stdio.h>
#include <unistd.h>
#include "mylib.h"

int main (int argc, char *argv[])
{
  puts ("Elenco dei file in /tmp");
  execl ("/bin/ls", "ls", "/tmp", NULL);
  syserr (argv[0], "execl() fallita");
}
```

fork e exec

- Tipicamente `fork` viene usata con `exec`.
- Il processo figlio generato con `fork` viene usato per fare la `exec` di un certo programma.
- Esempio:

```
int pid = fork ();
if (pid == -1) {
  perror("");
} else if (pid == 0) {
  char *args [2];
  args [0] = "ls"; args [1] = NULL;
  execvp (args [0], args);
  exit (1);   /* vedi dopo */
} else {
  printf ("Sono il padre, e mio figlio e' %d.\n", pid);
}
```

Sincronizzazione tra padre e figli

```
#include <sys/types.h>
#include <sys/wait.h>

void  exit(status)
void  _exit(status)
pid_t wait (int *status)
```

- `exit` è un wrapper all'effettiva system call `_exit()`
- `wait` sospende l'esecuzione di un processo fino a che uno dei figli termina.
  - Ne restituisce il PID ed il suo stato di terminazione, tipicamente ritornato come argomento dalla `exit`.
  - Restituisce `-1` se il processo non ha figli.
- Un figlio resta *zombie* da quando termina a quando il padre ne legge lo stato (con `wait()`).

Sincronizzazione tra padre e figli

- Lo stato può essere testato con le seguenti macro:

```
WIFEXITED(status)    WEXITSTATUS(status)    WIFSIGNALED(status)
WTERMSIG(status)     WIFSTOPPED(status)     WSTOPSIG(status)
```

- Informazione ritornata da `wait`
  - Se il figlio è terminato con `exit`
    - Byte 0: tutti zero
    - Byte 1: l'argomento della `exit`
  - Se il figlio è terminato con un segnale
    - Byte 0: il valore del segnale
    - Byte 1: tutti zero
- Comportamento di `wait` modificabile tramite segnali (v.dopo)

La Famiglia di `wait`

```
#include <sys/time.h>
#include <sys/resource.h>

pid_t waitpid (pid_t pid, int *status, int options)
pid_t wait3 (int *status, int options, struct rusage *rusage)
```

- `waitpid` attende la terminazione di un particolare processo
  - `pid = -1`: tutti i figli
  - `pid = 0`: tutti i figli con stesso GID del processo chiamante
  - `pid` $<$ `-1` : tutti i figli con GID $= |$`pid`$|$
  - `pid > 0`: il processo `pid`
- `wait3` e' simile a `waitpid`, ma ritorna informazioni aggiuntive sull'uso delle risorse all'interno della struttura `rusage`. Vedere `man getrusage` per ulteriori informazioni.

```
/*****************************************
MODULO: wait.c
SCOPO: esempio d'uso di wait()
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "mylib.h"

int main (int argc, char *argv[]){
  pid_t child;
  int status;

  if ((child=fork()) == 0) {
    sleep(5);
    puts ("figlio 1 - termino con stato 3");
```

```
  exit (3);
}

if (child == -1)
  syserr (argv[0], "fork() fallita");

if ((child=fork()) == 0) {
  puts ("figlio 2 - sono in loop infinito, uccidimi con:");
  printf ("  kill -9 %d\n", getpid());

  while (1) ;
}

if (child == -1)
  syserr (argv[0], "fork() fallita");

/*while ((child=wait(&status)) != -1) {*/
while ((child=waitpid(-1, &status, WUNTRACED|WCONTINUED)) != -1) {
```

```
    printf ("il figlio con PID %d e'", child);
    if (WIFEXITED(status)) {
      printf ("terminato (stato di uscita: %d)\n\n",
        WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
      printf ("stato ucciso (segnale omicida: %d)\n\n",
        WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
      puts ("stato bloccato");
      printf ("(segnale bloccante: %d)\n\n",
        WSTOPSIG(status));
    } else if (WIFCONTINUED(status)) {
      puts ("stato sbloccato");
    } else
      puts ("non c'e` piu` !?");
  }
  return 0;
}
```

Informazioni sui processi

```
#include <sys/types.h>
#include <unistd.h>

pid_t uid = getpid()
pid_t gid = getppid()
```

- `getpid` ritorna il PID del processo corrente
- `getppid` ritorna il PID del padre del processo corrente

```
/*****************************************
MODULO: fork2.c
SCOPO: funzionamento di getpid() e getppid()
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include "mylib.h"

int main (int argc, char *argv[])
{
  pid_t status;
  if ((status=fork()) == -1) {
    syserr (argv[0], "fork() fallita");
  }
  if (status == 0) {
    puts ("Io sono il figlio:\n");
    printf("PID = %d\tPPID = %d\n",getpid(),getppid());
  }
```

```
  else {
    printf ("Io sono il padre:\n");
    printf("PID = %d\tPPID = %d\n",getpid(),getppid());
  }
}
```

Informazioni sui processi – (cont.)

```
#include <sys/types.h>
#include <unistd.h>

uid_t uid = getuid()
uid_t gid = getgid()
uid_t euid = geteuid()
uid_t egid = getegid()
```

- Ritornano la corrispondente informazione del processo corrente
- `geteuid` e `getegid` ritornano l'informazione sull'*effective* UID e GID, eventualmente settato con `chmod` (bit `s,S,t`).

Segnalazioni tra processi

- E' possibile spedire asincronamente dei segnali ai processi

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig)
```

- Valori possibili di `pid`:

  (`pid` > 0) segnale inviato al processo con PID=`pid`

  (`pid` $= 0$) segnale inviato a tutti i processi con gruppo uguale a quello del processo chiamante

  (`pid` -1) segnale inviato a tutti i processi (tranne quelli di sistema)

  (`pid` < -1) segnale inviato a tutti i processi nel gruppo -`pid`

- *Gruppo di processi*: insieme dei processi aventi un antenato in comune.

Segnalazioni tra processi – (cont.)

- Il processo che riceve un segnale asincrono può specificare una routine da attivarsi alla sua ricezione.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int signum, sighandler_t func);
```

- `func` è la funzione da attivare, anche detta *signal handler*. Può essere una funzione definita dall'utente oppure:

  `SIG_DFL` per specificare il comportamento di default

  `SIG_IGN` per specificare di ignorare il segnale

- All'arrivo di un segnale l'handler e' resettato a `SIG_DFL`.

Segnalazioni tra processi – (cont.)

- Segnali disponibilii (Linux): con il comando `kill -l` o su `man 7 signal`

SIGHUP	1+	Hangup	SIGINT	2+	Interrupt
SIGQUIT	3*	Quit	SIGILL	4*	Illegal instr.
SIGTRAP	5*	Trace trap	SIGABRT	6*	Abort signal.
SIGBUS	7*	Bus error	SIGFPE	8*	FP exception
SIGKILL	9+@	Kill	SIGUSR1	10+	User defined 1
SIGSEGV	11*	Segm. viol.	SIGUSR2	12+	User defined 2
SIGPIPE	13+	write on pipe	SIGALRM	14	Alarm clock
SIGTERM	15+	Software termination signal	-	16	
SIGCHLD	17#	Child stop/termination	SIGCONT	18	Continue after stop
SIGSTOP	19$@	Stop process	SIGTSTP	20$	Stop typed at tty
SIGTTIN	21$	Background read from tty	SIGTTOU	22$	Background write to tty
SIGURG	23#	Urgent condition on socket	SIGXCPU	24*	Cpu time limit
SIGXFSZ	25*	File size limit	SIGVTALRM	26+	Virtual time alarm
SIGPROF	27+	Profiling timer alarm	SIGWINCH	28#	Window size change
SIGIO	29+	I/O now possible	SIGPWR	30+	Power failure
SIGSYS	31+	Bad args to system call			

Segnalazioni tra processi – (cont.)

- Segnali con '+': azione di default = terminazione
- Segnali con '*': azione di default = terminazione e scrittura di un *core file*
- Segnali con '#': azione di default = ignorare il segnale
- Segnali con '$': azione di default = stoppare il processo
- Segnali con '@': non possono essere nè ignorati nè intercettati.
- I segnali 10 e 12 sono a disposizione dell'utente per gestire dei meccanismi di interrupt ad hoc.

  Sono tipicamente utilizzati insieme al *comando* `kill` per attivare la funzione desiderata in modo asincrono
- Esempio:
  Se un programma include l'istruzione `signal(SIGUSR1,int_proc);`, la funzione `int_proc` verrà eseguita tutte le volte che eseguo il comando

  `kill -10` *<PID del processo che esegue la* `signal`*>* .

```
#include <stdio.h>      /* standard I/O functions                    */
#include <unistd.h>     /* standard unix functions, like getpid()*/
#include <signal.h>     /* signal name macros, and the signal()
                           prototype */

/* first, here is the signal handler */
void catch_int(int sig_num)
{
    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[])
{
    /* set the INT (Ctrl-C) signal handler to 'catch_int' */
    signal(SIGINT, catch_int);
```

```
    /* now, lets get into an infinite loop of doing nothing. */
    for ( ;; )
        pause();
}
```

```
/*****************************************
MODULO: signal.c
SCOPO: esempio di ricezione di segnali
*****************************************/
#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <signal.h>
#include <stdlib.h>

long maxprim = 0;
long np=0;

void usr12_handler (int s) {
  printf ("\nRicevuto segnale n.%d\n",s);
  printf ("Il piu‘ grande primo trovato e‘ %ld\n",maxprim);
  printf ("Totale dei  numeri primi=%d\n",np);
}
```

```
void good_bye (int s) {
  printf ("\nIl piu‘ grande primo trovato e‘ %ld\n",maxprim);
  printf ("Totale dei  numeri primi=%d\n",np);
  printf ("Ciao!\n");
  exit (1);
}

int is_prime (long x) {
  long fatt;
  long maxfatt = (long)ceil(sqrt((double)x));
  if (x < 4) return 1;
  if (x % 2 == 0) return 0;

  for (fatt=3; fatt<=maxfatt; fatt+=2)
    return (x % fatt == 0 ? 0: 1);
}
```

```
int main (int argc, char *argv[]) {
  long n;

  signal (SIGUSR1, usr12_handler);
  /* signal (SIGUSR2, usr12_handler); */
  signal (SIGHUP, good_bye);

  printf("Usa kill -SIGUSR1 %d per vedere il numero
          primo corrente\n", getpid());
  printf("Usa kill -SIGHUP %d per uscire", getpid());
  fflush(stdout);

  for (n=0; n<LONG_MAX; n++)
  if (is_prime(n)) {
    maxprim = n;
    np++;
  }
}
```

Segnali e terminazione di processi

- Il segnale `SIGCLD` viene inviato da un processo figlio che termina al padre
- L'azione di default è quella di ignorare il segnale (che causa lo sblocco della `wait()`)
- Può essere intercettato per modificare l'azione corrispondente

Timeout e Sospensione

```
#include <unistd.h>
unsigned int alarm (unsigned seconds)
```

- `alarm` invia un segnale (SIGALRM) al processo chiamante dopo `seconds` secondi. Se `seconds` vale 0, l'allarme è annullato.
- La chiamata resetta ogni precedente allarme
- Utile per implementare dei *timeout*, fondamentali per risorse utilizzate da più processi.
- Valore di ritorno:
  - 0 nel caso normale
  - Nel caso esistano delle `alarm()` con tempo residuo, il numero di secondi che mancavano all'allarme.
- Per cancellare eventuali allarmi sospesi: `alarm(0);`

```
#include <stdio.h>     /* standard I/O functions */
#include <unistd.h>    /* standard unix functions, like alarm() */
#include <signal.h>    /* signal name macros,
                          and the signal() prototype */
#include <stdlib.h>

char user[40];    /* buffer to read user name from the user */

/* define an alarm signal handler. */
void catch_alarm(int sig_num)
{
    printf("Operation timed out. Exiting...\n\n");
    exit(0);
}

int main(int argc, char* argv[])
{
    /* set a signal handler for ALRM signals */
```

```
    signal(SIGALRM, catch_alarm);

    /* prompt the user for input */
    printf("Username: ");
    fflush(stdout);
    /* start a 10 seconds alarm */
    alarm(10);
    /* wait for user input */
    scanf("%s",user);
    /* remove the timer, now that we've got the user's input */
    alarm(0);

    printf("User name: '%s'\n", user);

    return 0;
}
```

Timeout e Sospensione

```
#include <unistd.h>

void pause ()
```

- Sospende un processo fino alla ricezione di un qualunque segnale.
- Ritorna sempre -1
- N.B.: se si usa la `alarm` per uscire da una `pause` bisogna inserire l'istruzione `alarm(0)` dopo la `pause` per disabilitare l'allarme. Questo serve per evitare che l'allarme scatti dopo anche se la `pause` e' già uscita a causa di un'altro segnale.