ESD Electronic Systems Design
Synthesis Verification Testing Power Communication

Programmazione di sistema in Unix: Gestione di file e processi

N. Drago, G. Di Guglielmo, L. Di Guglielmo,
V. Guarnieri, M. Lora, G. Pravadelli, F. Stefanni

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 è 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*).

Alcune 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.
- È importante ottimizzare il numero di chiamate di sistema rispetto a quelle di libreria.
- Particolarmente evidente nel caso di system call per il file system.

Esempio:

```
1   /* PROG1 */
2   int main(void) {
3     int c;
4     while ((c = getchar()) != EOF) putchar(c);
5   }
6   /* PROG2 */
7   int main(void) {
8     char c;
9     while (read(0, &c, 1) > 0)
10    if(write(1, &c, 1)!=1){perror("write");exit(1)};
11  }
```

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

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

su `stderr` viene stampato:

```
1$ ./a.out
   str:  messaggio-di-errore \n
```

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

La libreria mylib

```
1  /*****************************************
2  MODULO: mylib.h
3  SCOPO: definizioni per la libreria mylib
4  *****************************************/
5
6  #ifndef MYLIB_H
7  #define MYLIB_H
8
9  void syserr (char *prog, char *msg);
10
11 #endif
```

La libreria mylib

```
1  /*************************************
2  MODULO: mylib.c
3  SCOPO: libreria di funzioni d'utilita'
4  *************************************/
5  #include <stdio.h>
6  #include <errno.h>
7  #include <stdlib.h>
8
9  #include "mylib.h"
10
11 void syserr (char *prog, char *msg)
12 {
13   fprintf (stderr, "%s - errore: %s\n",prog, msg);
14   perror ("system error");
15   exit (1);
16 }
```

Esempio di utilizzo di errno

```
1  #include <errno.h>
2  #include <unistd.h>
3  ...
4  /* Before calling the syscall, resetting errno */
5  errno = 0;
6  ssize_t bytes = write(1, "Hello!", 7);
7  if (bytes == -1)
8  {
9      if (errno == EBADF)
10     {
11         ...
12     }
13     ...
14 }
15 ...
```

Introduzione

- In UNIX esistono i seguenti tipi di file:
  1. File regolari
  2. Directory
  3. Link
  4. *pipe* o *fifo*
  5. *special file*
- Gli special file rappresentano un device (*block device* o *character device*)
- Non contengono dati, ma solo un puntatore al device driver:
  - **Major number**: indica il tipo del device (driver).
  - **Minor number**: indica il numero di unità del device.

I/O non bufferizzato

- Le funzioni in `stdio.h` (`fopen()`, `fread()`, `fwrite()`, `fclose()`, `printf()`) sono tutte bufferizzate. Per efficienza, si può lavorare direttamente sui buffer.
- Le funzioni POSIX `open()`, `read()`, `write()`, `close()` sono non bufferizzate.
  - In questo caso i file non sono più descritti da uno *stream* ma da un *descrittore* (*file descriptor* – un intero piccolo).
  - Alla partenza di un processo, i primi tre descrittori vengono aperti automaticamente dalla shell:

    0 ... stdin
    1 ... stdout
    2 ... stderr

    - Per distinguere, si parla di *canali* anziché di file.

Apertura di un canale

```
1    #include <fcntl.h>
2
3    int open (char *name, int access, mode_t mode);
```

Valori del parametro `access` (vanno messi in OR):

- Uno a scelta fra:
  `O_RDONLY O_WRONLY O_RDWR`
- Uno o più fra:
  `O_APPEND O_CREAT O_EXCL O_SYNC O_TRUNC`

Valori del parametro `mode`: uno o più fra i seguenti (in OR):
`S_IRUSR S_IWUSR S_IXUSR S_IRGRP S_IWGRP S_IXGRP`
`S_IROTH S_IWOTH S_IXOTH S_IRWXU S_IRWXG S_IRWXO`
Corrispondenti ai modi di un file UNIX (u=RWX,g=RWX,o=RWX), e rimpiazzabili con codici numerici *ottali* (`0000 ... 0777`). Comunque, per portabilità e mantenibilità del codice, è sempre meglio usare le costanti fornite dallo standard.

Apertura di un canale

- Modi speciali di `open()`:
  - `O_EXCL`: apertura in modo esclusivo (nessun altro processo può aprire/creare)
  - `O_SYNC`: apertura in modo sincronizzato (file tipo lock, prima terminano eventuali operazioni di I/O in atto)
  - `O_TRUNC`: apertura di file esistente implica cancellazione contenuto
- Esempi di utilizzo:
  - `int fd = open("file.dat",O_RDONLY|O_EXCL,S_IRUSR);`
  - `int fd = open("file.dat",O_CREAT, S_IRUSR|S_IWUSR);`
  - `int fd = open("file.dat",O_CREAT, 0700);`

Apertura di un canale

```
1    #include <fcntl.h>
2
3    int creat (char *name, int mode);
```

- `creat()` crea un file (più precisamente un inode) e lo apre in lettura.
  - Parametro `mode`: come `access`.
- Sebbene `open()` sia usabile per creare un file, tipicamente si utilizza `creat()` per creare un file, e la `open()` per aprire un file esistente da leggere/scrivere.

Manipolazione diretta di un file

```
1     #include <unistd.h>
2
3     ssize_t read  (int fildes, void *buf, size_t n);
4     ssize_t write (int fildes, void *buf, size_t n);
5     int close (int fildes);
6
7     #include <sys/types.h>
8     #include <unistd.h>
9
10    off_t  lseek (int fildes, off_t o, int whence);
```

- Tutte le funzioni restituiscono `-1` in caso di errore.
- `n`: numero di byte letti. Massima efficienza quando n = dimensione del blocco fisico (512 byte o 1K).
- `read()` e `write()` restituiscono il numero di byte letti o scritti, che può essere inferiore a quanto richiesto.
- `lseek()` riposiziona l'offset di un file aperto
  - Valori possibili di `whence`: `SEEK_SET` `SEEK_CUR` `SEEK_END`

Esempio

```
1  /*****************************************
2  MODULO: lower.c
3  SCOPO: esempio di I/O non bufferizzato
4  *****************************************/
5  #include <stdio.h>
6  #include <ctype.h>
7  #include "mylib.h"
8  #define BUFLEN 1024
9  #define STDIN 0
10 #define STDOUT 1
11
12 void lowerbuf (char *s, int l)
13 {
14   while (l-- > 0) {
15     if (isupper(*s)) *s = tolower(*s);
16     s++;
17   }
18 }
```

Esempio

```
1  int main (int argc, char *argv[])
2  {
3    char buffer[BUFLEN];
4    int x;
5
6    while ((x=read(STDIN,buffer,BUFLEN)) > 0)
7    {
8      lowerbuf (buffer, x);
9      x = write (STDOUT, buffer, x);
10     if (x == -1)
11       syserr (argv[0], "write() failure");
12   }
13   if (x != 0)
14     syserr (argv[0], "read() failure");
15   return 0;
16 }
```

Duplicazione di canali

```
1     int dup( int oldd );
```

- Duplica un file descriptor esistente e ne ritorna uno nuovo che ha in comune con il vecchio le seguenti proprietà:
  - si riferisce allo stesso file
  - ha lo stesso *puntatore* (per l'accesso casuale)
  - ha lo stesso modo di accesso.
- **Proprietà importante**: `dup()` ritorna il primo descrittore libero a partire da 0!

Creazione di una directory

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3
4    int mknod(char *path, mode_t mode, dev_t dev);
```

- Simile a `creat()`: crea un i-node per un file.
- Può essere usata per creare un file.
- Più tipicamente usata per creare directory e special file.
- Solo il super-user può usarla (eccetto che per special file).

Creazione di una directory

Valori di `mode`:

- Per indicare tipo di file:

`S_IFIFO`	0010000	FIFO special
`S_IFCHR`	0020000	Character special
`S_IFDIR`	0040000	Directory
`S_IFBLK`	0060000	Block special
`S_IFREG`	0100000	Ordinary file
	0000000	Ordinary file

- Per indicare il modo di esecuzione:

`S_ISUID`	0004000	Set user ID on execution
`S_ISGID`	0002000	Set group ID on execution
`S_ISVTX`	0001000	Set the sticky bit

Creazione di una directory

- Per indicare i permessi:

`S_IREAD`	0000400	Read by owner
`S_IWRITE`	0000200	Write by owner
`S_IEXEC`	0000100	Execute (search on directory) by owner
`s_IRWXG`	0000070	Read, write, execute (search) by group
`S_IRWXD`	0000007	Read, write, execute (search) by others

- Il parametro `dev` indica il major e minor number del device, mentre viene ignorato se non si tratta di uno special file.

Creazione di una directory

- La creazione con `creat()` di una directory **non** genera le entry "." e ".."
- Queste devono essere create "a mano" per rendere usabile la directory stessa.
- In alternativa (consigliato) si possono utilizzare le funzioni di libreria:

```
1       #include <sys/stat.h>
2       #include <sys/types.h>
3       #include <fcntl.h>
4       #include <unistd.h>
5
6       int mkdir (const char *path, mode_t mode);
7       int rmdir (const char *path);
```

Accesso alle directory

- Sebbene sia possibile aprire e manipolare una directory con `open()`, per motivi di portabilità è consigliato utilizzare le funzioni della libreria C (non system call).

```
1    #include <sys/types.h>
2    #include <dirent.h>
3
4    DIR *opendir (char *dirname);
5    struct dirent *readdir (DIR *dirp);
6    void rewinddir (DIR *dirp);
7    int closedir (DIR *dirp);
```

- `opendir()` apre la directory specificata (cfr. `fopen()`)
- `readdir()` ritorna un puntatore alla prossima entry della directory `dirp`
- `rewinddir()` resetta la posizione del puntatore all'inizio
- `closedir()` chiude la directory specificata

Accesso alle directory

- Struttura interna di una directory:

```
1 struct dirent  {
2   __ino_t d_ino;                    /* inode # */
3   __off_t d_off;                    /* offset in the
4                                    directory structure */
5   unsigned short int d_reclen; /* how large this
6                                   structure really is */
7   unsigned char d_type;             /* file type */
8   char d_name[256];                 /* file name */
9 };
```

Esempio

```
1  /*************************************
2  MODULO: dir.c
3  SCOPO: ricerca in un directory e
4         rinomina di un file
5  *************************************/
6  #include <string.h>
7  #include <sys/types.h>
8  #include <sys/dir.h>
9
10 int dirsearch( char*, char*, char*);
11
12 int main (int argc, char *argv[])
13 {
14     return dirsearch (argv[1],argv[2],".");
15 }
```

Esempio

```
1  int dirsearch (char *file1, char* file2,  char *dir)
2  {
3    DIR *dp;
4    struct dirent *dentry;
5    int status = 1;
6
7    if ((dp=opendir (dir)) == NULL) return -1;
8    for (dentry=readdir(dp); dentry!=NULL;
9         dentry=readdir(dp))
10   if ((strcmp(dentry->d_name,file1)==0)) {
11     printf("Replacing entry %s with %s",
12            dentry->d_name,file2);
13     strcpy(dentry->d_name,file2);
14     return 0;
15   }
16   closedir (dp);
17   return status;
18 }
```

Accesso alle directory

```
1     int chdir (char *dirname);
```

- Cambia la directory corrente e si sposta in `dirname`.
- È necessario che la directory abbia il permesso di esecuzione.

Gestione dei Link

```
1    #include <unistd.h>
2
3    int link (char *orig_name, char *new_name);
4    int unlink (char *file_name);
```

- `link()` crea un hard link a `orig_name`. E' possibile fare riferimento al file con entrambi i nomi.
- `unlink()`
  - Cancella un file cancellando l'*i-number* nella directory entry.
  - Sottrae uno al link count nell'i-node corrispondente.
  - Se questo diventa zero, libera lo spazio associato al file.
- `unlink()` è l'unica system call per cancellare file!

Esempio

```
1  /* Scopo: usare un file temporaneo senza che altri
2   * possano leggerlo.
3   * 1) aprire file in scrittura
4   * 2) fare unlik del file
5   * 3) usare il file, alla chiusura del processo
6   *      il file sara' rimosso
7   * NOTA: e' solo un esempio! Questo NON e' il modo
8   * corretto per creare file  temporanei. Per creare
9   * normalmente i file temporanei usare le funzioni
10  * tmpfile() o tmpfile64().
11  */
12  int   fd;
13  char  fname[32];
```

Esempio

```
1  ...
2  strcpy(fname,"myfile.xxx");
3  if ((fd = open(fname, O_WRONLY)) == -1)
4  {
5    perror(fname);
6    return 1;
7  } else if (unlink(fname) == -1) {
8    perror(fname);
9    return 2;
10 } else {
11   /* use temporary file */
12 }
13 ...
```

Privilegi e accessi

```
1    #include <unistd.h>
2    int access (char *file_name, int access_mode);
```

- `access()` verifica i permessi specificati in `access_mode` sul file `file_name`.
- I permessi sono una combinazione bitwise dei valori `R_OK`, `W_OK`, e `X_OK`.
- Specificando `F_OK` verifica se il file esiste
- Ritorna 0 se il file ha i permessi specificati

Privilegi e accessi

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    int chmod (char *file_name, int mode);
4    int fchmod (int fildes, int mode);
```

- Permessi possibili: bitwise OR di:

S_ISUID	04000	set user ID on execution
S_ISGID	02000	set group ID on execution
S_ISVTX	01000	sticky bit
S_IRUSR	00400	read by owner
S_IWUSR	00200	write by owner
S_IXUSR	00100	execute (search on directory) by owner
S_IRWXG	00070	read, write, execute (search) by group
S_IRWXO	00007	read, write, execute (search) by others

Privilegi e accessi

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    int chown (char *file_name, int owner, int group);
```

- `owner` = UID
- `group` = GID
- ottenibili con system call `getuid()` e `getgid()` (cfr. sezione sui processi)
- Solo super-user!

Stato di un file

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    #include <unistd.h>
4
5    int stat(char *file_name, struct stat *stat_buf);
6    int fstat(int fd, struct stat *stat_buf);
```

- Ritornano le informazioni contenute nell'i-node di un file
- L'informazione è ritornata dentro `stat_buf`.

Stato di un file

- Principali campi di `struct stat`:

```
1  dev_t           st_dev;      /* device */
2  ino_t           st_ino;      /* inode */
3  mode_t          st_mode;     /* protection */
4  nlink_t         st_nlink;    /* # of hard links */
5  uid_t           st_uid;      /* user ID of owner */
6  gid_t           st_gid;      /* group ID of owner */
7  dev_t           st_rdev;     /* device type
8                                * (if inode device) */
9  off_t           st_size;     /* total byte size */
10 unsigned long   st_blksize; /* blocksize for
11                               * filesystem I/O */
12 unsigned long   st_blocks;  /* number of blocks
13                               * allocated */
14 time_t          st_atime; /* time of last access */
15 time_t          st_mtime; /* time of last
16                             * modification */
17 time_t          st_ctime; /* time of last
18                            /* property change */
```

Esempio

```
1  #include <time.h>
2  ...
3  /* per stampare le informazioni con stat */
4  void display (char *fname, struct stat *sp)
5  {
6    printf ("FILE %s\n", fname);
7    printf ("Major number = %d\n", major(sp->st_dev));
8    printf ("Minor number = %d\n", minor(sp->st_dev));
9    printf ("File mode = %o\n", sp->mode);
10   printf ("i-node number = %d\n", sp->ino);
11   printf ("Links = %d\n", sp->nlink);
12   printf ("Owner ID = %d\n", sp->st_uid);
13   printf ("Group ID = %d\n", sp->st_gid);
14   printf ("Size = %d\n", sp->size);
15   printf ("Last access = %s\n", ctime(&sp->atime));
16 }
```

Stato di un file

- Alcuni dispositivi (terminali, dispositivi di comunicazione) forniscono un insieme di comandi *device-specific*
- Questi comandi vengono eseguiti dai device driver
- Per questi dispositivi, il mezzo con cui i comandi vengono passati ai device driver è la system call `ioctl()`.
- Tipicamente usata per determinare/cambiare lo stato di un terminale

```
1       #include <termio.h>
2       int ioctl(int fd, int request,
3                 ... /* argptr */ );
```

- `request` è il comando device-specific, `argptr` definisce una struttura usata dal device driver eseguendo `request`.

Le variabili di ambiente

```c
#include <stdlib.h>

char *getenv (char *env_var);
```

- Ritorna la definizione della variabile d'ambiente richiesta, oppure `NULL` se non è definita.
- Si può accedere anche alla seguente variabile globale:
  `extern char **environ;`
- È possibile esaminare in sequenza tutte le variabili d'ambiente usando il terzo argomento del `main()`. Questa signature non appartiene allo standard C, ma è diffusa nei sistemi POSIX. Pertanto è da evitare in programmi che vogliono essere portabili (usate uno degli altri due modi).

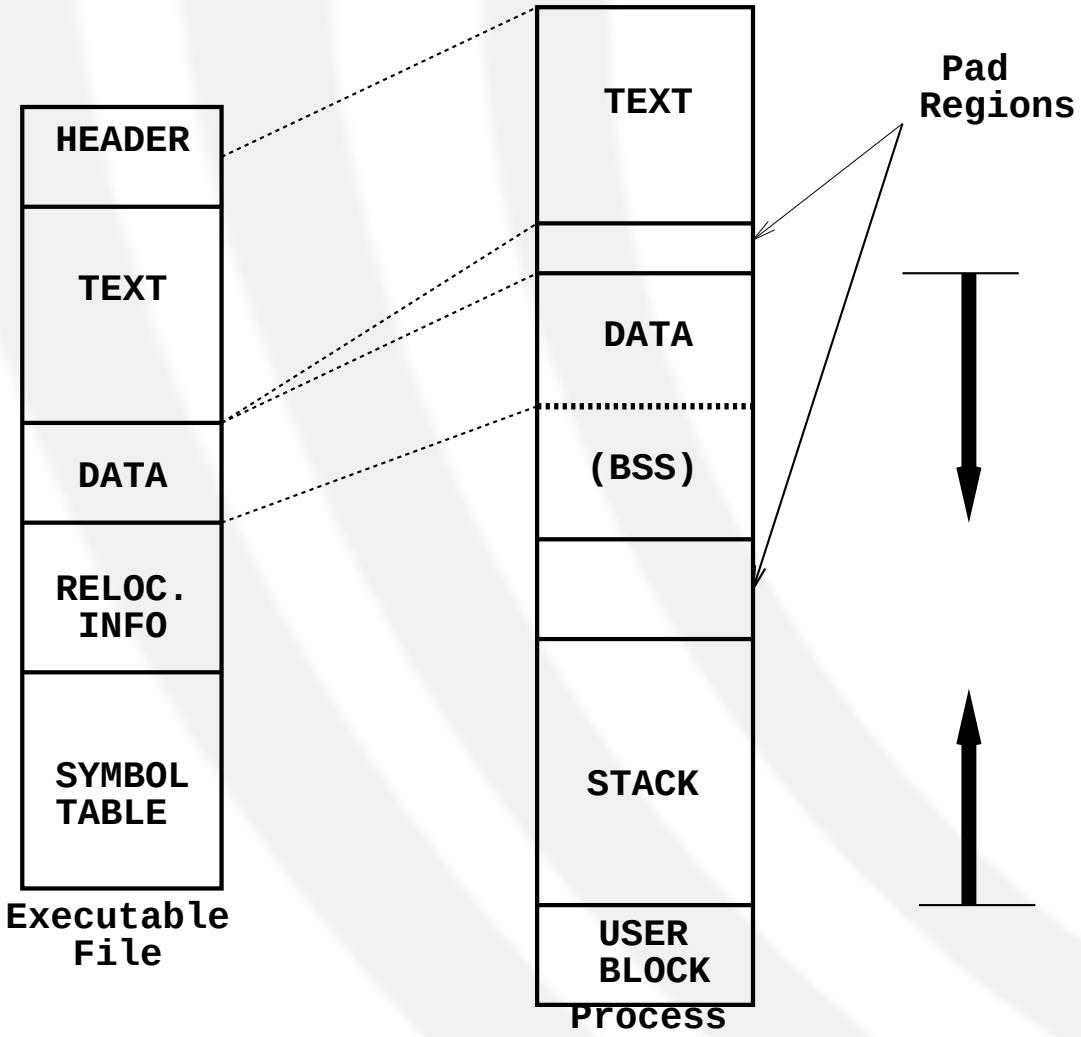
```c
int main (int argc, char *argv[], char *env[]);
```

Esempio

```
1  /*************************************
2  MODULO: env.c
3  SCOPO: elenco delle variabili d'ambiente
4  *************************************/
5  #include <stdio.h>
6
7  int main (int argc, char *argv[], char *env[])
8  {
9    puts ("Variabili d'ambiente:");
10   while (*env != NULL)
11     puts (*env++);
12   return 0;
13 }
```

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/linux/a.out.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 (statici, extern). 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: Visualizzabile con il comando `nm`. Può essere rimossa (`ld -s`) o con `strip` (programmi piú piccoli). 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

```
1 #include <unistd.h>
2
3 pid_t fork (void)
```

- Crea un nuovo processo, figlio di quello corrente, che eredita dal padre:
  - I file aperti (non i lock)
  - Le variabili di ambiente
  - Directory di lavoro
- Solo la thread corrente viene replicata nel processo figlio.
- 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!

Creazione di processi - Esempio

```
1  /*************************************
2  MODULO: fork.c
3  SCOPO: esempio di creazione di un processo
4  *************************************/
5  #include <stdio.h>
6  #include <sys/types.h>
7  #include "mylib.h"
8  int main (int argc, char *argv[]){
9     pid_t status;
10    if ((status=fork()) == -1)
11    syserr (argv[0], "fork() fallita");
12    if (status == 0) {
13       sleep(10);
14       puts ("Io sono il figlio!");
15    } else {
16       sleep(2);
17       printf ("Io sono il padre e");
18       printf (" mio figlio ha PID=%d)\n", status);
19    }
20 }
```

Esecuzione di un programma

```
1  #include <unistd.h>
2  int execl (char *file,char *arg0,char *arg1, ...,
3      (char *) NULL)
4  int execlp(char *file,char *arg0,char *arg1, ...,
5      (char *) NULL)
6  int execle(char *file,char *arg0,char *arg1, ...,
7      (char *) NULL, char *envp[])
8  int execv (char *file,char *argv[])
9  int execvp(char *file,char *argv[])
10 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 (i.e., "interpreter script") avente come prima riga `#! interpreter`
- In altri termini, `exec` trasforma un eseguibile in processo.
- NOTA: `exec` non ritorna se il processo viene creato con successo!!

La Famiglia di `exec`

- crea un processo come se fosse stato lanciato direttamente da linea di comando (non vengono ereditati file descriptors, semafori, memorie condivise, etc.),
- `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`.
- `(char *) NULL` nella famiglia `execl` serve come terminatore nullo (come nel vettore `argv` del `main()`).
- Nella famiglia `execv`, gli array `argv` e `envp` devono essere terminati da NULL.

Esempio di chiamate ad exec

Esempio di chiamata ad `execl`:

```
1    execl("/bin/ls", "/bin/ls", "/tmp", (char*) NULL);
```

Esempio di chiamata a `execv`, equivalente al codice precedente:

```
1    char * argv[3];
2    argv[0] = "/bin/ls";
3    argv[1] = "/tmp";
4    argv[2] = NULL;
5    execv("/bin/ls", argv);
```

Esempio di loop su array NULL-terminated

Esempio di loop su argv tramite `while`:

```
1    int main( int argc, char * argv[] ) {
2      char ** s = argv;
3      while( *s != NULL )
4      {
5        char * curr = *s;
6        ...
7        ++s;
8      }
9    }
```

Esempio di loop su argv tramite `for`:

```
1    int main( int argc, char * argv[] ) {
2      for(char ** s = argv; *s != NULL; ++s)
3      {
4        char * curr = *s;
5        ...
6      }
7    }
```

Esecuzione di un programma - Esempio

```
1  /*************************************
2  MODULO: exec.c
3  SCOPO: esempio d'uso di exec()
4  *************************************/
5  #include <stdio.h>
6  #include <unistd.h>
7  #include "mylib.h"
8
9  int main (int argc, char *argv[])
10 {
11   puts ("Elenco dei file in /tmp");
12   execl ("/bin/ls", "/bin/ls", "/tmp", (char *) NULL);
13   syserr (argv[0], "execl() fallita");
14 }
```

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:

```
1  int pid = fork ();
2  if (pid == -1) {
3    perror("");
4  } else if (pid == 0) {
5    char *args [2];
6    args [0] = "ls"; args [1] = NULL;
7    execvp (args [0], args);
8    exit (1);    /* vedi dopo */
9  } else {
10   printf ("Sono il padre");
11   printf (" e mio figlio e' %d.\n", pid);
12 }
```

Sincronizzazione tra padre e figli

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdlib.h> // richiesto per exit ed _exit
4
5 void  exit(int status)
6 void  _exit(int status)
7 pid_t wait (int *status)
```

- `exit()` è un wrapper all'effettiva system call `_exit()`
- `exit()` chiude gli eventuali stream aperti.
- `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()` (a meno che il padre non abbia impostato di ignorare la terminazione dei figli).

Sincronizzazione tra padre e figli

- Lo stato può essere testato con le seguenti macro:

```
1 WIFEXITED(status)
2 WEXITSTATUS(status) // se WIFEXITED ritorna true
3 WIFSIGNALED(status)
4 WTERMSIG(status)    // se WIFSIGNALED ritorna true
5 WIFSTOPPED(status)
6 WSTOPSIG(status)    // se WIFSTOPPED ritorna true
```

- 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`

```
1 #include <sys/time.h>
2 #include <sys/resource.h>
3
4 pid_t waitpid (pid_t pid, int *status, int options)
5 pid_t wait3 (int *status, int options,
6               struct rusage *rusage)
```

- `waitpid` attende la terminazione di un particolare processo
  - `pid = -1`: tutti i figli
    - `wait(&status);` è equivalente a `waitpid(-1, &status, 0);`
  - `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.

Uso di `wait()` - Esempio - pt. 1

```c
/*************************************
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);
  }
```

Uso di `wait()` - Esempio - pt. 2

```
1   if (child == -1)
2     syserr (argv[0], "fork() fallita");
3
4   if ((child=fork()) == 0) {
5     puts ("figlio 2 - sono in loop infinito,");
6     punts (" uccidimi con:");
7     printf (" kill -9 %d\n", getpid());
8
9     while (1) ;
10  }
11
12  if (child == -1)
13    syserr (argv[0], "fork() fallita");
```

Uso di `wait()` - Esempio - pt. 3

```
1   while ((child=wait(&status)) != -1) {
2     printf ("il figlio con PID %d e'", child);
3     if (WIFEXITED(status)) {
4       printf ("terminato (stato di uscita: %d)\n\n",
5         WEXITSTATUS(status));
6     } else if (WIFSIGNALED(status)) {
7       printf ("stato ucciso (segnale omicida: %d)\n\n",
8         WTERMSIG(status));
9     } else if (WIFSTOPPED(status)) {
10      puts ("stato bloccato");
11      printf ("(segnale bloccante: %d)\n\n",
12        WSTOPSIG(status));
13    } else if (WIFCONTINUED(status)) {
14      puts ("stato sbloccato");
15    } else
16      puts ("non c'e' piu' !?");
17  }
18  return 0;
19 }
```

Informazioni sui processi

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t uid = getpid()
5 pid_t gid = getppid()
```

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

Informazioni sui processi - Esempio

```
1  /*************************************************
2  MODULO: fork2.c
3  SCOPO: funzionamento di getpid() e getppid()
4  *************************************************/
5  #include <stdio.h>
6  #include <sys/types.h>
7  #include "mylib.h"
8  int main (int argc, char *argv[]) {
9    pid_t status;
10   if ((status=fork()) == -1) {
11     syserr (argv[0], "fork() fallita");
12   }
13   if (status == 0) {
14     puts ("Io sono il figlio:\n");
15     printf("PID = %d\tPPID = %d\n",getpid(),getppid());
16   }
17   else {
18     printf ("Io sono il padre:\n");
19     printf("PID = %d\tPPID = %d\n",getpid(),getppid());
20   }}
```

Informazioni sui processi – (cont.)

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 uid_t uid = getuid()
5 uid_t gid = getgid()
6 uid_t euid = geteuid()
7 uid_t egid = getegid()
```

- Ritornano la corrispondente informazione del processo corrente
- `geteuid` e `getegid` ritornano l'informazione sull'*effective* UID e GID.
- NOTA: la differenza tra *effective* e *real* sta nell'uso dei comandi `suid`/`sgid` che permettono di cambiare UID/GUID per permettere operazioni normalmente non concesse agli utenti. Il *real* UID/GID si riferisce sempre ai dati reali dell'utente che lancia il processo. L'*effective* UID/GID si riferisce ai dati ottenuti lanciando il comando `suid`/`sgid`.

Segnalazioni tra processi

- È possibile spedire asincronamente dei segnali ai processi:

```
1 #include <sys/types.h>
2 #include <signal.h>
3
4 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.

```
1 #include <signal.h>
2 typedef void (*sighandler_t)(int);
3 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
- Il valore ritornato è l'handler registrato precedentemente.
- A seconda dell'implementazione o dei flag di compilazione, il comportamento cambia. Nelle versioni attuali di Linux:
  - Handler `SIG_DFL` o `SIG_IGN`: all'arrivo di un segnale, l'handler impostato viene mantenuto.
  - Flag di compilazione `-std=` o `-ansi`: all'arrivo di un segnale l'handler e' resettato a `SIG_DFL` (semantica Unix e System V).
  - Senza tali flag di compilazione: rimane impostato l'handler corrente (semantica BSD).

Segnalazioni tra processi – (cont.)

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

POSIX.1-1990			
Signal	**Value**	**Def. Action**	**Description**
SIGHUP	1	Term	Hangup on contr. terminal or death of contr. process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Segnalazioni tra processi – (cont.)

POSIX.1-2001 and SUSv2			
Signal	**Value**	**Def. Action**	**Description**
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

Other signals			
Signal	**Value**	**Def. Action**	**Description**
SIGIOT	6	Core	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for SIGCHLD
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for SIGPWR
SIGLOST	-,-,-	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Core	Synonymous with SIGSYS

Segnalazioni tra processi – (cont.)

- Dove multipli valori sono forniti, il primo è valido per alpha e sparc, quello centrale per ix86, ia64, ppc, s390, arm e sh, e l'ultimo per mips.
- Un trattino indica che il segnale non è presente sulle architetture specificate.
- I segnali SIGKILL e SIGSTOP non possono essere intercettati.
- I processi figlio ereditano le impostazioni dal processo padre, ma `exec` resetta ai valori di default.
- NOTA: in processi multithread, la ricezione di un segnale risulta in un comportamento *non definito*!

Segnalazioni tra processi - Esempio 1 - pt. 1

```
1  #include <stdio.h>    /* standard I/O functions          */
2  #include <unistd.h>   /* standard unix functions,        *
3                         * like getpid()                   */
4  #include <signal.h>   /* signal name macros, and the     *
5                         * signal() prototype              */
6
7  /* first, here is the signal handler */
8  void catch_int(int sig_num)
9  {
10     /* re-set the signal handler again to catch_int,
11      * for next time */
12     signal(SIGINT, catch_int);
13     printf("Don't do that\n");
14     fflush(stdout);
15 }
```

Segnalazioni tra processi - Esempio 1 - pt. 2

```
1  int main(int argc, char* argv[])
2  {
3      /* set the INT (Ctrl-C) signal handler
4       * to 'catch_int' */
5      signal(SIGINT, catch_int);
6
7      /* now, lets get into an infinite loop of doing
8       * nothing. */
9      for ( ;; )
10         pause();
11 }
```

Segnalazioni tra processi - Esempio 2 - pt. 1

```c
/*************************************
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'");
  printf ("%ld\n",maxprim);
  printf ("Totale dei  numeri primi=%d\n",np);
}
```

Segnalazioni tra processi - Esempio 2 - pt. 2

```
1  void good_bye (int s) {
2    printf ("\nIl piu' grande primo trovato e'");
3    printf ("%ld\n",maxprim);
4    printf ("Totale dei  numeri primi=%d\n",np);
5    printf ("Ciao!\n");
6    exit (1);
7  }
8
9  int is_prime (long x) {
10   long fatt;
11   long maxfatt = (long)ceil(sqrt((double)x));
12   if (x < 4) return 1;
13   if (x % 2 == 0) return 0;
14
15   for (fatt=3; fatt<=maxfatt; fatt+=2)
16     return (x % fatt == 0 ? 0: 1);
17 }
```

Segnalazioni tra processi - Esempio 2 - pt. 3

```
1  int main (int argc, char *argv[]) {
2    long n;
3
4    signal (SIGUSR1, usr12_handler);
5    /* signal (SIGUSR2, usr12_handler); */
6    signal (SIGHUP, good_bye);
7
8    printf("Usa kill -SIGUSR1 %d per vedere il numero
9            primo corrente\n", getpid());
10   printf("Usa kill -SIGHUP %d per uscire", getpid());
11   fflush(stdout);
12
13   for (n=0; n<LONG_MAX; n++)
14   if (is_prime(n)) {
15     maxprim = n;
16     np++;
17   }
18 }
```

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

```
1 #include <unistd.h>
2 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);`

Timeout e sospensione - Esempio - pt. 1

```
1  #include <stdio.h>    /* standard I/O functions          */
2  #include <unistd.h>   /* standard unix functions,        *
3                         * like getpid()                   */
4  #include <signal.h>   /* signal name macros, and the     *
5                         * signal() prototype              */
6
7  /* buffer to read user name from the user */
8  char user[40];
9
10 /* define an alarm signal handler. */
11 void catch_alarm(int sig_num)
12 {
13     printf("Operation timed out. Exiting...\n\n");
14     exit(0);
15 }
```

Timeout e sospensione - Esempio - pt. 2

```
1  int main(int argc, char* argv[])
2  {
3      /* set a signal handler for ALRM signals */
4      signal(SIGALRM, catch_alarm);
5
6      /* prompt the user for input */
7      printf("Username: ");
8      fflush(stdout);
9      /* start a 10 seconds alarm */
10     alarm(10);
11     /* wait for user input */
12     scanf("%s",user);
13     /* remove the timer, now that we've got
14      * the user's input */
15     alarm(0);
16
17     printf("User name: '%s'\n", user);
18     return 0;
19 }
```

Timeout e Sospensione - (cont.)

```
1 #include <unistd.h>
2 int 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 `pause` e' già uscita a causa di un'altro segnale.