Capitolo 1

Gestione dei processi

1.1 Creazione di un processo

I principali eventi che possono dare luogo ad un nuovo processo sono:

- inizializzazione del sistema;
- esecuzione di una primitiva di sistema per la creazione di un nuovo processo;
- richiesta dell'avvio di un nuovo processo da parte dell'utente, eseguita tramite l'interprete dei comandi;
- avvio di un nuovo lavoro batch.

In ogni caso, è un processo a generare un nuovo processo. A seconda del sistema operativo, ed eventualmente delle scelte fatte dal programmatore, si hanno diverse possibilità relativamente ai seguenti problemi:

- condivisione delle risorse:
  - padre e figlio condividono tutte le risorse;
  - padre e figlio condividono alcune risorse;
  - padre e figlio non condividono alcuna risorsa;
- esecuzione/sincronizzazione:
  - padre e figlio procedono in modo concorrente;
  - il padre attende il completamento dell'esecuzione del figlio;
- spazio di indirizzamento/memoria:
  - quello del figlio è una copia di quello del padre. In realtà, qualche differenza esiste sempre: se includiamo nella memoria del processo anche il suo PCB, almeno il PID dovrà differire da quello del padre;

- il figlio carica in memoria un nuovo programma. In Java si ha solo questa seconda possibilità, mentre in Unix/Linux si possono avere entrambe.

1.2 Processi in Unix/Linux

Un processo Unix è caratterizzato dalla sua *immagine*. Questa consiste di:

- immagine di memoria (*virtual address space*);
- valore dei registri (*CPU status*);
- tabella dei file (*file descriptor table*);
- directory di lavoro (*working directory*).

Alla creazione di un processo ad esso è assegnato uno spazio virtuale degli indirizzi (*virtual address space*). È lo spazio usato dal processo durante la sua esecuzione.

Le informazioni di sistema relative ad un processo sono invece mantenute in due aree: la *user area* e la *process table*.

La *user area* di un processo è localizzata al termine della parte superiore dell'address space del processo ed è accessibile solo quando il sistema esegue in modo monitor. Contiene:

- puntatore all'ingresso relativo al processo nella process table;
- setting dei segnali[1];
- tabella dei file;
- directory di lavoro corrente;
- umask settings[2].

La user area può essere sottoposta a swap su disco.

La process table contiene le principali informazioni mantenute dal sistema operativo relativamente ad ogni processo. È localizzata nella memoria del kernel del sistema operativo e non può essere sottoposta a swap. Ogni ingresso della tabella contiene:

- PID del processo
- PID del processo padre (PPID)
- ID utente (user ID)

1

2

- stato
- descrittore di evento (per un processo in stato di sleeping, indica l'evento per cui il processo è in attesa);
- puntatori alle tabelle delle pagine;
- dimensione delle tabelle delle pagine;
- posizione della user area;
- priorità;
- segnali pendenti.

Il comando UNIX `ps` mostra una combinazione delle informazioni contenute nella process table e nella user area.

1.2.1 Struttura della memoria dei processi

Lo spazio di indirizzi virtuale di un processo è diviso in segmenti, cioè un insieme contiguo di indirizzi virtuali ai quali il processo può accedere. Solitamente un programma in linguaggio *C* viene suddiviso nei seguenti segmenti:

1. Il segmento di testo (*text segment*). Contiene il codice del programma, il codice delle funzioni di libreria da esso utilizzate e le costanti. Normalmente è condiviso fra tutti i processi che eseguono lo stesso programma (e anche da processi che eseguono altri programmi nel caso delle librerie). Viene marcato in sola lettura per evitare sovrascritture accidentali (o maliziose) che ne modifichino le istruzioni. Viene allocato da `exec()` all'avvio del programma e resta invariato per tutto il tempo di esecuzione del processo;
2. Il segmento dei dati (*data segment*). Contiene le variabili globali (cioè quelle definite al di fuori di tutte le funzioni che compongono il programma) e le variabili statiche (cioè quelle dichiarate con l'attributo `static`). Di norma è diviso in due parti:
    - la prima parte è il segmento dei dati inizializzati, che contiene le variabili il cui valore è stato assegnato esplicitamente. Ad esempio, se si definisce: `double pi = 3.1415;` questo valore sarà immagazzinato in questo segmento. La memoria di questo segmento viene preallocata all'avvio del programma e inizializzata ai valori specificati;

- la seconda parte è il segmento dei dati non inizializzati, che contiene le variabili il cui valore non è stato assegnato esplicitamente. Ad esempio, se si definisce: `int vect [100];` questo vettore sarà immagazzinato in questo segmento. Anch'esso viene allocato all'avvio, e tutte le variabili vengono inizializzate a zero (ed i puntatori a NULL). Storicamente questo segmento viene chiamato BBS (*block started by symbol*). La sua dimensione è fissa, nel senso che non varia durante l'esecuzione del processo. Notare che il BSS non è salvato sul file che contiene l'eseguibile, dato che viene sempre inizializzato a zero al caricamento del programma;

3. Lo *heap*. Lo si può considerare come l'estensione del segmento dati, a cui di solito è posto di seguito. È qui che avviene l'allocazione dinamica della memoria; può essere ridimensionato allocando e deallocando la memoria dinamica con le apposite funzioni (`malloc, calloc, free, ...`), ma il suo limite inferiore, quello adiacente al segmento dati, ha una posizione fissa;

4. Il segmento di *stack*, che contiene lo stack del programma. Tutte le volte che si effettua una chiamata ad una funzione è qui che viene salvato l'indirizzo di ritorno e le informazioni dello stato del chiamante (ad esempio, il contenuto di alcuni registri della CPU). La funzione chiamata alloca qui lo spazio per le sue variabili locali: in questo modo le funzioni possono essere chiamate ricorsivamente. Al ritorno dalla funzione, lo spazio è automaticamente rilasciato e ripulito. La pulizia in C e C++ viene fatta dal chiamante. La dimensione di questo segmento aumenta seguendo la crescita dello stack del programma, ma non viene ridotta quando quest'ultimo si restringe.

La disposizione tipica di questi segmenti è riportata in Fig.1.1. Usando il comando `size` su un programma è possibile stampare le dimensioni dei segmenti di testo e di dati (inizializzati e BSS).

1.2.2 System call per la gestione dei processi

In Unix/Linux un numero limitato di system call sono sufficienti per la gestione dei processi. In particolare:

```
int fork();
int getpid();
int getppid();
int exit(int);
int wait(int * status);
int execve(char * path-file-name, char ** argv, char ** envp);
```

Il risultato della chiamata delle system call precedenti può essere schematicamente descritto come segue:

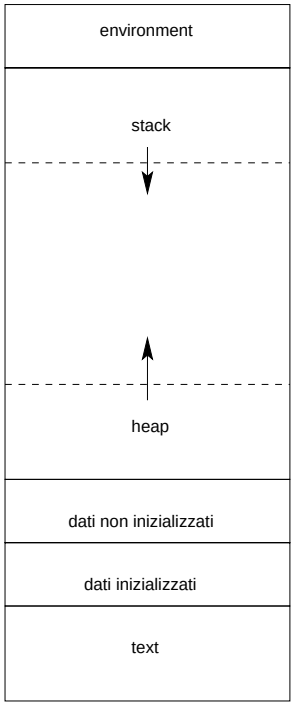

Figura 1.1: Disposizione tipica dei segmenti di memoria di un processo Linux.

- `fork()`: crea un nuovo processo. Il nuovo processo è una copia (a parte alcune piccole differenze) del processo chiamante. Il processo che ha invocato la `fork()` è detto padre (*parent*), il nuovo processo creato dalla `fork()` è detto figlio (*child*);
- `getpid()`: restituisce il PID del processo chiamante;
- `getppid()`: restituisce al processo chiamante il PID del processo padre;
- `exit()`: termina il processo chiamante. Non ha un valore di ritorno in quanto non può fallire. Il singolo parametro in ingresso alla `exit` è usato come codice di uscita da restituire al processo padre. Un codice di uscita uguale a 0 è normalmente usato per indicare una terminazione normale;
- `wait()`: pone il processo chiamante in attesa del termine di uno dei suoi processi figli. Il PID del processo figlio terminato è il valore di ritorno. Utilizzando questa chiamata è possibile stabilire una semplice forma di sincronizzazione tra processo padre e figlio.

Nel seguito l'uso delle precedenti system call è discusso attraverso esempi.

1.2.3 Creazione dei processi

Una delle caratteristiche dei Sistemi Operativi Unix/Linux è che qualunque processo può generare altri processi, detti *figli* (child process). Ogni processo nel sistema è identificato da un numero univoco, il cosiddetto *process identifier* (PID), assegnato in forma progressiva quando il processo viene creato. In Unix/Linux ogni processo è sempre generato da un altro processo, detto *padre* (parent process). L'unica eccezione è rappresentata dal primo processo (`/sbin/init`), creato dal kernel al completamento della fase di bootstrap del sistema, il cui PID è 1 e non ha un processo padre.

In ogni momento un processo può accedere al proprio PID e a quello del padre tramite le system call `getpid()` e `getppid()`. Come conseguenza, i processi hanno una organizzazione di tipo gerarchico. Il comando `pstree` produce una rappresentazione ad albero di questa gerarchia.

In Linux i processi vengono creati con la funzione fork(), che si appoggia alla system call `_clone()`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)
```

In caso di successo, la `fork()` crea un nuovo processo (processo figlio) e restituisce il PID del figlio al padre e zero al figlio; ritorna -1 al padre (senza creare il figlio) in caso di errore. Possibili cause di errore sono la mancanza di risorse sufficienti per creare un altro processo (per allocare la tabella delle pagine e le strutture del task), l'esaurimento dei degli identificati da assegnare ai processi, oppure l'indisponibilità di memoria per le strutture necessarie al kernel per creare il nuovo processo. Se la `fork()` è eseguita con successo, sia il processo padre che il processo figlio continuano la loro esecuzione in modo indipendente a partire dall'istruzione successiva alla fork. Tuttavia, il processo figlio è una copia del padre, ha cioè una copia dei segmenti di testo, stack e dati del padre al momento della invocazione della `fork()`. Come conseguenza, il codice eseguito dal figlio è esattamente lo stesso del padre. In Fig.1.2 è schematizzato l'effetto della chiamata a `fork()`.

Nei sistemi Unix/Linux, l'esecuzione della `fork()` si svolge, almeno a grandi linee, come indicato di seguito:

1. creazione di un nuovo ingresso nella tabella dei processi, ovvero creazione di un nuovo Process Control Block (PCB). Questo comporta l'assegnazione di un identificatore al nuovo processo, la registrazione dell'identificatore del padre, la definizione dello stato iniziale del nuovo processo indicato con `NEW` oppure `IDL` (*Image Definition and Loading*, ovvero definizione dell'immagine di memoria e caricamento), nonché l'assegnazione delle informazioni necessarie per il calcolo della priorità. Per il resto si provvede a copiare i valori del padre;

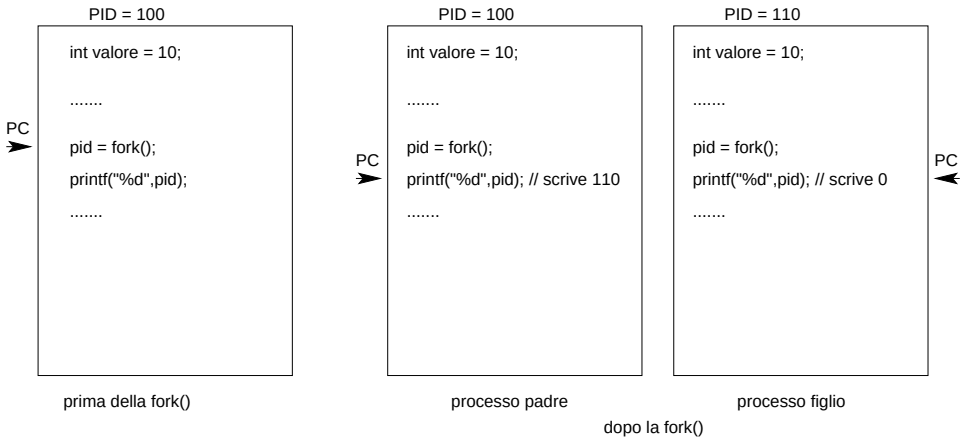

Figura 1.2: Duplicazione dell'immagine di memoria di un processo dopo la `fork()`.

2. se il codice è spartibile (ovvero, più processi possono fare riferimento allo stesso codice operativo) si procede all'aggiornamento della tabella dei testi (una tabella che tiene conto dei processi che fanno riferimento ad uno stesso segmento di codice in memoria), altrimenti si duplicherà anche il codice. Nel caso di Linux il segmento di testo (cioè il codice), è identico per i due processi, e quindi può essere condiviso (in sola lettura, per evitare problemi);

3. duplicazione dei segmenti dati e stack. Per motivi di efficienza, Linux utilizza la tecnica del copy on write, per cui una pagina di memoria viene effettivamente copiata nello spazio indirizzi del nuovo processo solo quando alla pagina è fatto accesso in scrittura (e si ha quindi una reale differenza tra padre e figlio);

4. duplicazione dei dati di sistema, con qualche eccezione (gli indirizzi dei segmenti dati e stack saranno nuovi perchè le aree di memoria dei segmenti del figlio sono diverse da quelle del padre). Il program counter ed i registri sono invece copiati;

5. determinazione del valore di ritorno della fork (il PID del figlio per il padre, 0 per il figlio). Si noti come la funzione fork ritorni due volte: una nel padre e una nel figlio;

6. lo stato del figlio viene posto a `READY`;

7. si riprende con l'esecuzione di un processo (quale esattamente, dipende dai criteri di scheduling implementati). Non è quindi possibile stabilire a priori quale dei due processi, padre o figlio, verrà eseguito al rientro dalla `fork()`.

I processi figli ereditano dal padre una serie di proprietà, tra cui:

- i file aperti e gli eventuali flag di close-on-exec impostati;
- gli identificatori per il controllo di accesso;
- la directory di lavoro e la directory radice;
- la maschera dei permessi di creazione;
- la maschera dei segnali bloccati e le azioni installate;
- i segmenti di memoria condivisa agganciati al processo;
- i limiti sulle risorse;
- le variabili di ambiente.

Le differenze tra padre e figlio dopo la fork sono:

- il valore di ritorno della fork();
- il pid (process id) e il ppid (parent process id), dove il ppid del figlio viene impostato al pid del padre;
- i valori dei tempi di esecuzione, che nel figlio sono posti a zero;
- i lock sui file, che non vengono ereditati dal figlio;
- gli allarmi ed i segnali pendenti, che per il figlio vengono cancellati.

In Fig.1.3 è riportato un programma di esempio che illustra l'uso della funzione `fork()`.
In Unix/Linux, il caricamento di un nuovo programma è cosa diversa dalla creazione di un nuovo processo. Un qualunque processo in esecuzione può sostituire la propria immagine di memoria tramite la system call `exec()`, che in pratica rimpiazza stack, heap dati e testo del processo, mentre il PID del processo non cambia.

```
#include <unistd.h>
int execve(char * filename, char * argv[], char * envp[])
```

Su questa system call di base sono costruite più funzioni di libreria che differiscono per numero e tipo degli argomenti passati:

- `int execv(char * filename, char * argv[])`. In questa forma, i puntatori alle variabili di ambiente non sono specificati ed `envp` è copiato dall'ambiente del programma chiamante;
- `int execle(char * filename, char * arg0, char * arg1, char * arg2, 0, char * envp[])`. Gli argomenti sono listati esplicitamente nella lista dei parametri;

```
/* necessari per fork() */
#include <sys/types.h>
#include <unistd.h>

/* necessario per atoi() */
#include <stdlib.h>

/* necessario per printf() */
#include <stdio.h>

int main(int argc, char ** argv)
{
    int valore = 0;

    if ( 1 < argc )
        valore = atoi( argv[1] );
    int pid = fork();
    if ( 0 == pid ) {
        /* codice eseguito dal figlio */
        printf( "figlio [pid: %d]> valore iniziale= %d\n", getpid(), valore );
        valore += 15;
        printf( "figlio [pid: %d]> valore finale= %d\n", getpid(), valore );
    }
    else if ( 0 > pid ) {
        /* codice eseguito dal padre in caso di errore */
        printf("padre [pid: %d]> errore creazione figlio.\n", getpid());
        return 1;
    }
    else {
        /* codice eseguito dal padre */
        printf("padre [pid: %d]> generato figlio con pid %d\n", getpid(), pid);
        printf( "padre [pid: %d]> valore = %d\n", getpid(), valore );
        valore += 10;
        printf( "padre [pid: %d]> valore finale= %d\n", getpid(), valore );
    }
    return 0;
}
```

Figura 1.3: Creazione di un nuovo processo con la fork().

- `int execl(char * filename, char * arg0, char * arg1, char * arg2, 0)`. Gli argomenti sono listati sequenzialmente come in `execle()`, mentre le variabili di ambiente sono derivate come in `execv()`;

- `execlp()` e `execvp()` hanno una sintassi equivalente, rispettivamente, ad `execl` ed `execv`, con la differenza che il nome del file indicato come primo argomento non necessita dell'intero PATH. Se solo un nome file è specificato, senza `PATH`, viene automaticamente ricercata la lista delle directory data nel `PATH` per localizzare il file.

In dettaglio, i passi eseguiti in corrispondenza alla chiamata della `exec()` sono:

- controllo sui diritti di esecuzione del programma specificato nella chiamata;

- rilascio della memoria attualmente allocata al processo (memoria centrale, ma anche la memoria su disco eventualmente utilizzata per lo swapping) e allocazione della memoria necessaria alla nuova immagine del programma;

- se il codice è spartibile, si verifica se è già presente in memoria ed eventualmente si provvede all'aggiornamento della tabella dei testi, altrimenti lo si carica; se non è spartibile, si procede comunque al caricamento;

- caricamento dell'area dati;

- inizializzazione dei registri;

- una volta terminata l'inizializzazione, il sistema esegue la funzione `main()` del programma.

Si osservi che la chiamata alla `exec()` ritorna solo in caso di errore. In particolare, la chiamata può fallire se non è possibile trovare il file eseguibile, se l'utente che ha lanciato il processo che esegue la `exec()` non ha sufficienti privilegi per eseguire il comando richiesto, se il file indicato esiste ma non è eseguibile.

Il codice in Fig.1.4 mostra l'uso della system call `exec()` per sostituire un diverso programma al processo generato dalla `fork()`.
Il processo mandato in esecuzione dalla `exec()` mantiene alcune proprietà del processo che ha sostituito, tra cui:

- il PID ed il PPID;

- il terminale di controllo;

- il tempo restante ad un allarme;

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

int main()
{
    int pid;

    pid = fork();

    if (pid<0) {
        /* la fork non ha avuto successo */
        fprintf(stderr,"errore durante esecuzione della fork\n");
        return (1);
    }
    else if (pid==0) {
        /* codice eseguito dal processo figlio */
        printf("esegue il processo figlio %d\n", getpid());
        fflush(stdout);

        /* listing dei file nella directory corrente */
        execlp("ls","ls",NULL);
    }
    else {
        /* codice eseguito dal processo padre */
        printf("esegue il processo padre %d\n", getpid());
        fflush(stdout);
        pid = wait(NULL);
        printf("Il figlio %d ha completato\n", pid);
        fflush(stdout);
    }
    return (0);
}
```

Figura 1.4: Esempio di utilizzo di fork() e exec().

- la directory radice e la directory di lavoro corrente;
- la maschera di creazione dei file ed i lock sui file ;
- i segnali sospesi e la maschera dei segnali;
- i limiti sulle risorse;
- i valori delle variabili che contabilizzano il tempo di esecuzione.

Per quanto riguarda i file aperti, il comportamento dipende dal valore del flag `close-on-exec` per ciascun file descriptor. I file per cui è impostato vengono chiusi, mentre gli altri restano aperti. Quindi i file restano aperti, a meno che non sia stata utilizzata la funzione fcntl() per impostare il flag. L'esempio di Fig.1.5 mostra l'uso di uno stesso file nel processo padre e figlio. La Fig.1.6 mostra i valori scritti sul file di uscita eseguendo l'esempio di Fig.1.5 su due diversi sistemi operativi (Linux e Unix).

1.2.4 Terminazione dei processi

Si possono avere diverse forme di terminazione di un processo:

- normale (volontaria);
- a seguito di errore (volontaria);
- a seguito di errore grave (involontaria);
- ad opera di un altro processo (involontaria)

In caso di terminazione normale il processo esegue l'ultima istruzione, lasciando al sistema l'ultima decisione. Quando il processo termina:

- i dati di uscita (exit status) passano dal figlio al padre;
- le risorse allocate al processo sono restituite al sistema.

Un processo può terminare forzatamente l'esecuzione di un suo figlio per diversi motivi:

- il processo figlio ha allocato più risorse di quanto non gli fosse consentito;
- il compito assegnato al processo figlio non è più necessario;
- il processo padre sta terminando. In particolare, quest'ultima evenienza si può verificare nel caso in cui il sistema operativo non consenta al figlio di sopravvivere al padre;
- si voglia realizzare una terminazione a cascata.

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

int main()
{
    int valore=4,
        * vettore,
        pid;
    FILE * fp;

    vettore = (int *) malloc (sizeof(int));
    vettore[0] = 2;
    fp = fopen("fork_file.txt","wt");
    pid = fork();
    if (pid<0) {
        /* la fork non ha avuto successo */
        fclose(fp);
        return(1);
    }
    else if (pid>0) {
        /* codice eseguito dal processo padre */
        fprintf(fp,"Step-1-padre: valore=%d  vettore=%p  vettore[0]=%d \n",
                valore,vettore,vettore[0]);
        valore = vettore[0];
        vettore[0] = 10;
        fprintf(fp,"Step-2-padre: valore=%d  vettore=%p  vettore[0]=%d \n",
                valore,vettore,vettore[0]);
        fclose(fp);
    }
    else {
        /* codice eseguito dal processo figlio */
        fprintf(fp,"Step-1-figlio: valore=%d  vettore=%p  vettore[0]=%d \n",
                valore,vettore,vettore[0]);
        valore = vettore[0];
        vettore[0] = 15;
        fprintf(fp,"Step-2-figlio: valore=%d  vettore=%p  vettore[0]=%d \n",
                valore,vettore,vettore[0]);
        fclose(fp);
    }
    return(0);
}
```

Figura 1.5: Uso della `fork()` e comportamento rispetto ai file aperti.

```
su LINUX:
Step-1-padre:  valore=4  vettore=0x8049898  vettore[0]=2
Step-2-padre:  valore=2  vettore=0x8049898  vettore[0]=10
Step-1-figlio: valore=4  vettore=0x8049898  vettore[0]=2
Step-2-figlio: valore=2  vettore=0x8049898  vettore[0]=15

su IRIX:
Step-1-figlio: valore=4  vettore=10015010  vettore[0]=2
Step-2-figlio: valore=2  vettore=10015010  vettore[0]=15
Step-1-padre:  valore=4  vettore=10015010  vettore[0]=2
Step-2-padre:  valore=2  vettore=10015010  vettore[0]=10
```

Figura 1.6: Contenuto del file "fork_file.txt" nel caso di esecuzione su sistema operativo Linux e Unix.

Terminazione dei processi in Unix/Linux

In ambiente Linux si hanno due possibilità per terminare normalmente un programma:

- la funzione di libreria `exit()`. Questa è anche la funzione che viene chiamata automaticamente quando la funzione `main()` ritorna;
- la system call `_exit()`.

```
#include <stdlib.h>
void exit(int status)

#include <unistd.h>
void _exit(int status)
```

Entrambe le chiamate terminano il programma e quindi non ritornano alcun valore. Le due funzioni accettano un argomento in ingresso che rappresenta lo stato di uscita (*exit status*) del processo. Tale stato può essere letto dal processo padre che si è eventualmente sospeso in attesa del termine del figlio. La funzione `exit()` esegue i seguenti passi:

- esegue tutte le funzioni che sono state registrate con `atexit()` e `on_exit()`;
- chiude tutti gli stream e salva i dati sospesi invocando la funzione `fclose()`;
- passa il controllo al kernel invocando la funzione di sistema `_exit()`.

La system call `_exit()`, invece, opera come segue:

- chiude tutti i descrittori di file (senza salvare i dati rimasti in sospeso);

- assegna eventuali figli del processo che si sta per terminare al processo *init*. In questo modo è garantita la condizione per cui ogni processo ha un processo padre;

- invia un segnale `SIGCHLD` al processo padre, ad indicare che il processo invocante sta per terminare;

- memorizza lo stato di uscita, che potrà essere recuperato dal padre invocando la system call `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * status)
```

Poiché non necessariamente un processo può ricevere immediatamente la notifica circa la terminazione di un figlio, si conserva il descrittore nella tabella dei processi finché non viene eseguita la funzione `wait()`. In questo periodo il processo terminato si trova nello stato `ZOMBIE`. La funzione `wait()`, infatti, scandisce la tabella dei processi per verificare se esistono processi figli nello stato `ZOMBIE`: se un tale processo esiste, allora viene acquisito lo stato di uscita e viene definitivamente cancellato dalla tabella dei processi; altrimenti, il processo chiamante si autosospende in attesa della terminazione dell'esecuzione di un suo processo figlio. In particolare:

- se il processo che invoca la `wait()` non ha figli, la chiamata ritorna immediatamente con un codice di errore;

- se il chiamante ha processi figli, ma nessuno ha terminato, il chiamante si sospende fino al termine di uno dei processi figli. La funzione restituisce il pid del processo figlio che ha terminato ed in `status` lo stato con cui ha terminato;

- se un processo figlio per cui nessuno si era già messo in attesa termina, il figlio è rimosso dalla tabella dei processi e la `wait` ritorna con la stato del figlio.

Se nella chiamata a `wait`, il puntatore ad intero passato come parametro (`int * status`), è diverso dal puntatore nullo (`NULL`), 16 bit di informazione sono memorizzati nei 16 bit meno significativi della locazione puntata da `status`. In particolare, i valori ritornati, rispettivamente nel caso di terminazione corretta e anomala (es. segnale `CTRL-C`), sono:

Si osservi che un processo è tenuto ad eseguire la funzione `wait()`, altrimenti si rischia di far crescere a dismisura la tabella dei processi e di esaurire i PID disponibili.

Per quanto riguarda la terminazione anomala, un programma Linux può invocare la funzione `abort()`, oppure può essere terminato da un segnale (in realtà anche l'invocazione di `abort()` ricade in quest'ultimo caso, visto che genera un segnale `SIGABRT`).

Nel caso di terminazione anormale di un processo, il sistema simula una chiamata alla funzione `exit()`, quindi liberando le risorse allocate ad un processo ed aggiornando le strutture dati necessarie al funzionamento del sistema operativo (tabella dei processi, tabella dei testi, etc.).

1.3 Creazione e terminazione dei processi Java

Le principali classi e metodi per la gestione dei processi in Java sono:

```
Runtime r = Runtime.getRuntime();
Process p = r.exec( String <programma> );
```

Il metodo `exec(String <programma>)` della classe `Runtime` in Java è diverso dalla funzione omonima in C/Linux. In pratica, la `exec()` di Java combina insieme le primitive `fork()` ed `exec()` di Linux. Come conseguenza, il controllo offerto da Java per la creazione di nuovi processi è meno fine. Questo approccio è seguito anche da alcuni sistemi operativi, tra cui Windows con la sua `CreateProcess()`. L'operazione che combina insieme `fork()` e `exec()` è spesso indicata con il termine *spawn*.

I principali metodi della classe `Process` sono:

```
public abstract int waitFor() throws InterruptedException
public abstract int exitValue() throws IllegalThreadStateException
public abstract void destroy()
```

Un processo può terminare volontariamente tramite l'invocazione del seguente metodo statico:

```
public static void exit( int <stato> )
```

definito nella classe `System`.

La Fig.1.7, mostra un esempio in cui un processo Java genera un processo figlio e si mette in attesa del suo completamento.

```
import java.io.IOException;

public class Padre
{
    /** Crea una nuova istanza della classe Padre */
    public Padre() {
    }

    public void esegui( String[] comando )
            throws IOException {
        System.out.println( "Sto per generare il processo figlio." );
        Process figlio = Runtime.getRuntime().exec( comando );
        System.out.println("Processo figlio generato; attesa del completamento");
        boolean attendi = true;
        while( attendi ) {
            try {
                figlio.waitFor();
                System.out.println( "Il figlio ha completato l'esecuzione" );
                attendi = false;
            }
            catch ( InterruptedException ie ) {
                System.out.println( "Sono stato interrotto nell'attesa" );
            }
        }
    }

    /**
     * @param args  argomenti sulla linea di comando
     */
    public static void main(String[] args)
    {
        if ( args.length < 2 ) {
            System.out.println( "java Padre <figlio> {<argomenti>}" );
            System.exit( 1 );
        }
        Padre p = new Padre();
        try {
            p.esegui( args );
        }
        catch ( IOException ioe ) {
            System.err.println("ERRORE: comando non trovato");
        }
    }
}
```

Figura 1.7: Creazione di processi in Java. Il processo padre genera il figlio e si mette in attesa del suo completamento.

Bibliografia

[1] S. Piccardi. http://gapil.firenze.linux.it.