Gestione dei processi in Unix/Linux

La creazione dei processi

Una delle caratteristiche di Unix/Linux è che qualunque processo può a sua volta generarne altri, detti **processi figli** (*child process*). Ogni processo è identificato da un numero univoco, il cosiddetto *process identifier* o PID, assegnato in forma progressiva quando il processo viene creato.

Quindi, ogni processo è sempre generato da un altro, che viene chiamato **processo padre** (*parent process*). L'unica eccezione è rappresentata dal primo processo, `/sbin/init`, che viene creato dal kernel al completamento del bootstrap; il suo pid è sempre 1, e non ha un padre.

In ogni momento un processo può accedere al proprio PID e a quello del padre tramite le funzioni `getpid()` e `getppid()`.

Da quanto detto sopra segue che i processi possono sono organizzati gerarchicamente. Il comando `pstree` è in grado di produrre una rappresentazione ad albero di questa gerarchia.

```
init-+-aacraid
   |-atd
   |-2*[automount]
   |-httpd---20*[httpd]
   |-java---java---28*[java]
   |-klogd
   |-kswapd
   |-lockd---rpciod
   |-lpd
   |-6*[mingetty]
   |-8*[nfsd]
   |-nmbd
   |-portmap
   |-postmaster---postmaster---postmaster
   |-rpc.mountd
   |-rpc.rquotad
   |-rpc.statd
   |-rpc.yppasswdd
   |-safe_mysqld---mysqld---mysqld---mysqld
   |-smbd
   |-sshd---sshd---tcsh---pstree
   |-syslogd
   |-xinetd
   |-ypbind---ypbind---2*[ypbind]
   `-ypserv
```

Figura 1: Estratto dell'albero dei processi di un sistema Linux.

Per completezza riportiamo anche il descrittore di processo per il S.O. Linux:

```
struct task_struct {
        volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
        unsigned long flags;  /* per process flags, defined below */
        int sigpending;
        mm_segment_t addr_limit;      /* thread address space:
                                                0-0xBFFFFFFF for user-thead
                                                0-0xFFFFFFFF for kernel-thread */
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;
        int lock_depth;               /* Lock depth */
        long counter;
        long nice;
        unsigned long policy;
        struct mm_struct *mm;
        int processor;
        /* cpus_runnable is ~0 if the process is not running on any
         * CPU. It's (1 << cpu) if it's running on a CPU. This mask
         * is updated under the runqueue lock.
         *
         * To determine whether a process might run on a CPU, this
         * mask is AND-ed with cpus_allowed. */
        unsigned long cpus_runnable, cpus_allowed;
        /* (only the 'next' pointer fits into the cacheline, but
         * that's just fine.) */
        struct list_head run_list;
        unsigned long sleep_time;
        struct task_struct *next_task, *prev_task;
        struct mm_struct *active_mm;
/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        /* ??? */
        unsigned long personality;
        int did_exec:1;
        unsigned task_dumpable:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;
        /* boolean value for session group leader */
        int leader;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->p_pptr->pid)
         */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        struct list_head thread_group;
        /* PID hash table linkage. */
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;
        wait_queue_head_t wait_chldexit;     /* for wait4() */
        struct completion *vfork_done;              /* for vfork() */
        unsigned long rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        struct tms times;
        unsigned long start_time;
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t  groups[NGROUPS];
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;
/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
```

```
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count, total_link_count;
        struct tty_struct *tty; /* NULL if no tty */
        unsigned int locks; /* How many file locks are being held */
/* ipc stuff */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespace */
        struct namespace *namespace;
/* signal handlers */
        spinlock_t sigmask_lock;      /* Protects signal and blocked */
        struct signal_struct *sig;
        sigset_t blocked;
        struct sigpending pending;
        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
        /* TUX state */
        void *tux_info;
        void (*tux_exit)(void);
/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
        spinlock_t alloc_lock;

/* journalling filesystem info */
        void *journal_info;
};
```

Nel descrittore si individuano facilmente i campi per la memorizzazione di PID e PPID, priorità, stato, ed altro ancora.

Per conoscere lo stato, così come altre informazioni relative ai processi in esecuzione sul sistema si può eseguire il comando `ps`:

```
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S     0     1     0  0  69   0    -   344 do_sel ?        00:00:22 init
040 S     0     2     1  0  69   0    -     0 contex ?        00:00:04 keventd
040 S     0    12     1  0  69   0    -     0 kupdat ?        00:00:14 kupdated
040 S     0    13     1  0  59 -20    -     0 md_thr ?        00:00:00 mdrecoveryd
040 S     0    19     1  0  69   0    -     0 end    ?        00:00:00 aacraid
140 S     0   777     1  0  69   0    -   342 do_sys ?        00:00:00 klogd
140 S    32   797     1  0  69   0    -   380 do_pol ?        00:00:02 portmap
140 S    29   825     1  0  69   0    -   402 do_sel ?        00:00:00 rpc.statd
140 S    38   944     1  0  69   0    -   471 do_sel ?        00:00:51 ntpd
140 S    48 24947  4680  0  69   0    - 20028 semtim ?        00:00:00 httpd
140 S    48 26222  4680  0  69   0    - 20024 semtim ?        00:00:00 httpd
140 S    48 26225  4680  0  69   0    - 19985 semtim ?        00:00:00 httpd
140 S     0  3118  1100  0  69   0    -   899 do_sel ?        00:00:00 sshd
100 S   263  3122  3118  0  73   0    -   862 rt_sig pts/0    00:00:00 tcsh
000 R   263  3272  3122  0  77   0    -   828 -      pts/0    00:00:00 ps
```

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)
```

La funzione crea un nuovo processo. In caso di successo essa restituisce il pid del figlio al padre e zero al figlio; in caso di errore, invece, restituisce -1 al padre (senza creare il figlio). Possibili cause di errore sono l'assenza di risorse sufficienti per creare un altro processo (per allocare la tabella delle pagine e le strutture del task), l'esaurimento dei numeri di processi, oppure l'indisponibilità di memoria per le strutture necessarie al kernel per creare il nuovo processo.

Dopo il successo dell'esecuzione di una fork sia il processo padre che il processo figlio continuano nella loro esecuzione a partire dall'istruzione successiva alla fork; il processo figlio è però una copia del padre, e riceve una copia dei segmenti di testo, stack e dati, ed esegue esattamente lo stesso codice del padre.

Qui di seguito è riportato il codice di un semplice programma che illustra l'uso della funzione `fork()`:

```
/*    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 numero_argomenti,
      char **argomenti
)
{
      int valore = 0;
      if ( 1 < numero_argomenti )
            valore = atoi( argomenti[ 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]> problemi durante creazione del figlio.\n",
                  getpid()
            );
            return 1;
      }
      else
      {
            /* codice eseguito dal padre */
            printf(
                  "padre [pid: %d]> generato un figlio; il suo pid e' %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;
}
```

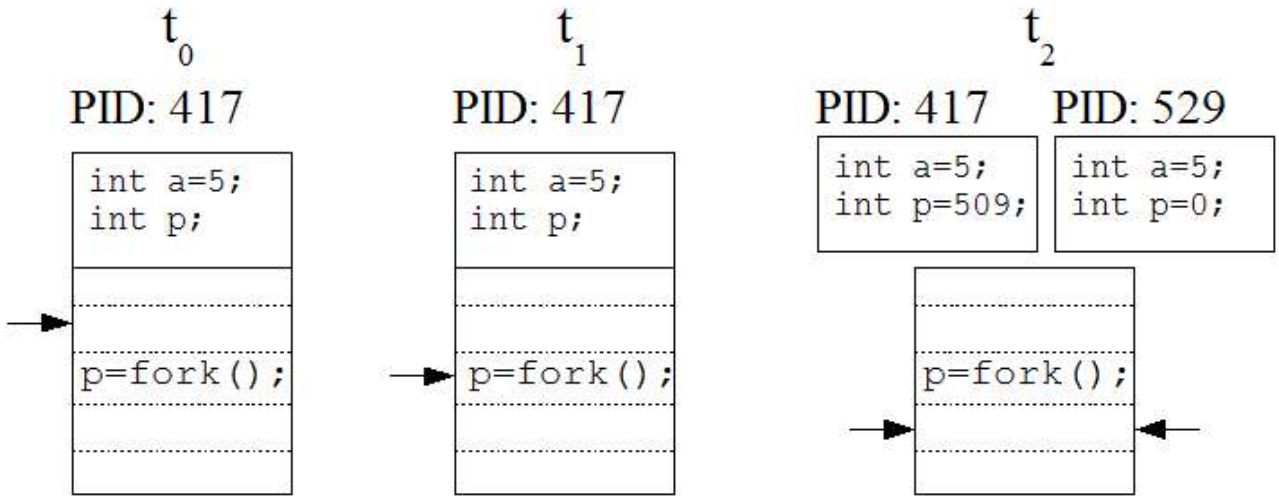

Figura 2: Creazione di un nuovo processo tramite la funzione fork()

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

1. creazione di un nuovo elemento 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 contenuti nel descrittore del padre;
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 per il nuovo processo solo quando ci viene effettuata sopra una scrittura (e si ha quindi una reale differenza fra padre e figlio).

4. duplicazione dei dati di sistema, con qualche eccezione (gli indirizzi dei segmenti dati e stack saranno nuovi);
5. determinazione del valore di ritorno della funzione `fork()` (il PID del figlio per il padre, e 0 per il figlio) e del valore del program counter, per il padre e per il figlio. Si noti come la funzione `fork()` ritorni due volte: una nel padre e una nel figlio (questo lo si ottiene preparando opportunamente i valori di ritorno dello stack su ciascuno dei due processi);
6. lo stato del figlio viene impostato 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, fra 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 fra padre e figlio dopo la fork invece sono:

- il valore di ritorno di `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 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 funzione `exec()`, che in pratica rimpiazza stack, heap dati e testo del processo; il PID del processo non cambia.

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

In dettaglio, i passi svolti dalla funzione `exec()` sono i seguenti:

1. controllo sui diritti di esecuzione del programma specificato nella chiamata alla funzione;
2. viene rilasciata la memoria attualmente allocata al processo (memoria centrale, ma anche memoria la memoria su disco eventualmente utilizzata per lo swapping) e viene allocata la memoria necessaria alla nuova immagine del programma;
3. se il codice è spartibile, si verifica se è già presente in memoria ed eventualmente all'aggiornamento della tabella dei testi altrimenti lo si carica; se non è spartibile, si procede comunque al caricamento;
4. caricamento dell'area dati;
5. inizializzazione dei registri;
6. una volta terminata l'inizializzazione, il sistema esegue la funzione `main()` del programma. Si osservi che la funzione `exec()` non ritorna mai.

Ad esempio, per eseguire un nuovo programma, si potrebbe modificare il codice visto sopra come segue:

```
/* [...omissis...] */

int
main
(
      int numero_argomenti,
      char **argomenti
)
{
      int pid = fork();
      if ( 0 == pid )
      {
            /* codice eseguito dal figlio */
            execve( "/percorso/programma", ..., ... );
      }
      else if ( 0 > pid )
      {
            /* codice eseguito dal padre in caso di errore */
            return 1;
      }
      else
      {
            /* codice eseguito dal padre */
      }
      return 0;
}
```

Il processo mantiene alcune proprietà, fra cui:

- il PID ed il PPID;
- il terminale di controllo;
- il tempo restante ad un allarme;
- 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 di certe variabili che contabilizzano il tempo di esecuzione.

Per quanto riguarda i files aperti, invece, il comportamento dipende dal valore che ha il flag di `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 effettuata utilizzata la funzione `fcntl()` per impostare il flag.

La terminazione dei processi

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 funzione di sistema `_exit()`.

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

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

Le due funzioni terminano il programma, e quindi non ritornano. Entrambe le funzioni accettano un parametro, che rappresenta lo **stato di uscita** (*exit status*) del processo, che sarà poi passato al processo padre.

La funzione `exit()` esegue i seguenti passi, per portare a termine in maniera pulita il processo invocante:

1. esegue tutte le funzioni che sono state registrate con `atexit()` e `on_exit()`;
2. chiude tutti gli stream effettuando il salvataggio dei dati sospesi, invocando la funzione `fclose()`;
3. passa il controllo al kernel invocando la funzione di sistema `_exit()`.

La funzione `_exit()`, invece, opera come segue:

1. chiude tutti i file descriptors (senza salvare i dati rimasti in sospeso);
2. assegna eventuali figli del processo che si sta per terminare al processo `init`. In questo modo è sempre verificata la condizione per cui ogni processo ha un processo padre;

3. invia un segnale `SIGCHLD` al processo padre, ad indicare che il processo invocante sta per terminare;
4. memorizza lo stato di uscita, che potrà essere recuperato dal padre invocando la funzione `wait()`.

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.

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 (es. web server che genera un figlio per ogni richiesta proveniente da un client).

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

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 svolge una serie di operazioni che in qualche modo corrispondono all'invocazione della funzione `_exit()`, e quindi si liberano comunque le risorse allocate ad un processo e si

aggiornano le strutture dati necessarie al funzionamento del sistema operativo (tabella dei processi, tabella dei testi, ecc.).