Il processo demone per FASTI

E.Giani[1]

[1]Osservatorio Astrofisico di Arcetri

Memo del Gruppo infrarosso.
Firenze 2002

1 Processi demoni

Un demone é un processo che viene eseguito in *background* in attesa che si verifichi qualche evento o in attesa di eseguire un certo compito specifico su base periodica. Ad esso non é associato un terminale di controllo o una login di shell.

1.1 Avviamento di un processo demone

Un demone puó essere avviato in diversi modi:

1. durante l'avviamento del sistema. La maggior parte dei demoni di sistema sono avviati dallo script di inizializzazione /etc/rc che viene eseguito da /etc/init quando il sitema é predisposto per la multiutenza. In questo caso il demone avrá i privilegi del superuser.
2. Dal file di sistema crontab su base periodica
3. Dal file di utente crontab su base periodica
4. eseguendo il comando at che pianifica un job per l'esecuzione in qualche istante successivo
5. da un terminale di utente come job in foreground o background. Di solito questo avviene durante la fase di test del demone.

A parte quest'ultimo caso, gli altri esempi generano un demone che non é connesso con un terminale. In effetti sono l'interazione normale di un teminale di login, il gruppo di processo associato a quel terminale e tutte le relative interazioni dei segnali che generano i problemi che devono essere gestiti correttamente quando si scrive un demone.

1.2 Codifica di un demone

La caratteristica dei processi demoni é che il loro tempo di vita é l'intero tempo in cui il sistema é in funzione. Generalmente vengono avviati al *bootstrap* del sistema e vengono terminati solo quando viene eseguito lo *shutdown*.

Esiste un insieme di regole per la codifica di un demone. Queste regole sono volte a fornire un demone "robusto" che possa essere invocato in uno qualsiasi dei modi elencati precedentemente.

1. La prima operazione consiste nell'eseguire la chiamata di sistema *fork* e fare in modo che il processo genitore esca. In questo modo se il demone viene eseguito come semplice comando di shell, la sua terminazione conduce la shell a ritenere che il comando abbia finito. Inoltre il processo figlio eredita il numero identificativo del processo di gruppo (PGID) del padre ma ottiene un diverso numero di processo (PID). In questo modo viene garantito che il processo figlio non é il *leader* del gruppo del processo. Questo costituisce un prerequisito per la chiamata a *setsid*

2. Viene chiamata *setsid()* per creare una nuova sessione. Il processo generato diventa *leader* di sessione di una nuova sessione e *leader* del gruppo di un nuovo gruppo di processo. Inoltre ad esso non é associato alcun terminale.

3. Cambiare la *directory* corrente di lavoro nella *directory* di *root*. La *directory* corrente di lavoro viene ereditata dal processo padre e puó trovarsi su un *filesystem* montato. Dato che il demone generalmente non termina fino a che il sistema non esegue il *reboot*, se il demone si trova su una partizione montata, il *filesystem* non puó essere smontato.

4. azzerare la maschera di creazione dei *files* (*umask(0)*). La maschera di creazione dei *files* viene ereditata dal processo genitore e questa potrebbe negare alcuni permessi che il processo demone potrebbe invece volere abilitare nei *files* creati.

5. I decrittori dei files non utilizzati devono essere chiusi. Ció vale specialmente per lo *standard input*, lo *standard output* e lo *standard error* che il processo puó avere ereditato dal processo genitore.

1.3 Terminazione del demone

Il segnale SIGTERM viene usato dal sistema per notificare a tutti i processi in esecuzione che il sistema sta passando dalla multiutenza alla monoutenza. Il segnale viene inviato dal processo *init*, che aspetta la terminazione dei processi. Se un processo non é terminato dopo un certo lasso di tempo, il segnale SIGKILL viene inviato al processo che non puó ignorarlo. Un demone dovrebbe catturare il segnale SIGTERM ed utilizzarlo per arrestare gradualmente le sue operazioni.

2 Il server demone *ftest*

Abbiamo modificato il programma di laboratorio *ftest* in modo che venga eseguito come processo *server* demone sul sistema *embedded* FASTI.

ftest processa le richieste provenienti dall' applicazione *xnics*, il programma utente di acquisizione, che viene eseguito su una macchina remota con una connessione di rete diretta al sistema embedded.

2.1 Trasformazione di *ftest* in demone di sistema

Il programma *ftest* viene eseguito come demone di sistema chiamando la funzione **daemon()**[1]. Questa esegue i primi 4 passi descritti al § 1.2.

La funzione accetta due argomenti *nochdir* e *noclose* che se posti uguali a zero cambiano rispettivamente la *directory* corrente di lavoro nella *root directory* e redirigono lo *standard input,output* ed *error* su */dev/null*.

[1] La funzione *daemon()* compare la prima volta in BSD4.4

La funzione *daemon()* puó essere implementata nel seguente modo sui sistemi per cui non é definita:

```
#include <fcntl.h>
#include <paths.h>
#include <unistd.h>

int daemon(int nochdir, int noclose) {
        int fd;

        switch (fork()){
                case -1: return -1;
                case 0: break;
                default: _exit(0);
        }
        if (setsid() == -1) return -1;
        if (!nochdir) chdir("/");
        if (noclose) return 0;
        fd = open(_PATH_DEVNULL, O_RDWR, 0);
        if (fd != -1) {
                dup2(fd, STDIN_FILENO);
                dup2(fd, STDOUT_FILENO);
                dup2(fd, STDERR_FILENO);
                if (fd > 2) close(fd);
        }
 return 0;
}
```

Il processo demone *ftest* viene successivamente configurato in modo da ignorare i segnali di terminale SIGTINT, SIGQUIT,SIGHUP,SIGTTIN e SIGTTOU.

In corrispondenza del segnale SIGTERM viene invece registrata la routine *termination_ handler()* che garantisce una conclusione corretta del demone *ftest*.

2.2 Il server *ftest*

Il processo *ftest* gestisce la richiesta proveniente da un unico cliente (*xnics*) quindi lo possiamo classificare come un *server iterativo*.

La struttura generale di un *server iterativo* orientato alla connessione, é la seguente:

1. creazione di una socket
   (a) sock =socket(AF_INET,SOCK_STREAM,0);
2. *bind* ad un indirizzo noto

(a) - bind(sock,localaddr,addrlen);

(b) - port: puó essere un indirizzo noto oppure 0. In questo caso il sistema assegna automaticamente un numero di porta, nell'intervallo da 1024 a 5000.

(c) - address: generalmente viene usato INADDR_ANY. Con tale costante il sistema accetta una connessione su qualsiasi interfaccia Internet;

3. configurazione della socket in modalitá passiva

   (a) - listen(sock,queuelen)

   (b) - queuelen: la lunghezza massima dipende dall'implementazione

4. accetta una connessione dal cliente

   (a) - new_sock =accept(sock,addr,addrlen)

   (b) - la chiamata accept() blocca finché non c'é almeno una richiesta di connessione [2]

5. interazione con il cliente

   (a) - read(new_socket,....)

   (b) - write(new_socket,...)

6. chiusura della socket e ritorno alla chiamata *accept()*(punto 4)

Nel caso specifico, *ftest* apre due *socket,* una per i comandi ed una per i dati, in corrispondenza della porta 8081 e si pone in attesa delle connessioni da parte dell'applicazione cliente *xnics*.

Una volta stabilite le connessioni, queste vengono usate dai due processi per lo scambio di comandi e dei dati, per cui ci riferiamo ad esse con il termine di *socket* di comando e *socket* dei dati.

Le *socket* rimangono aperte per tutta la durata del programma cliente *xnics*. Quando questo termina, le connessioni vengono chiuse da entrambe le parti.

2.3 Interazione con il cliente

Nel programma *ftest* l'interazione con il cliente viene gestita dalla funzione *MainLoopServer()*.

La routine *MainLoopServer()* legge dalla *socket* di comando le richieste inviate dall'applicazione *xnics* ed esegue le corrispondenti operazioni.

I comandi e gli eventuali parametri spediti attraverso la connessione di comando, devono essere specificati seguendo il protocollo di comunicazione stabilito per le due applicazioni: questo richiede che ciascun stringa di comando sia correttamente terminata con il carattere nullo *'\0'*.

All'interno della routine *MainLoopServer()* la funzione *parse_ cmd()* si occupa di individuare i comandi all'interno della stringa letta dalla *socket*.

[2] Questo é vero se *sock* é in modalitá bloccante.

2.4 Trasferimento dei dati acquisiti

I dati vengono spediti all' applicazione *xnics* sulla *socket* dati. I dati sono immagazzinati in una matrice di 1024 righe per 1024 colonne e viene spedita una riga per volta (2048 bytes). Dopo la spedizione viene attesa la conferma della corretta ricezione dei dati.

Nel caso in cui si verifichi un errore la riga viene rispedita per intero.

2.5 Terminazione di ftest

In corrispondenza del segnale SIGTERM abbiamo registrato la routine di gestione *termination_handler()* che stampa un messaggio sul *file* di log ed azzera la *flag keep_gooing* che controlla l'iterazione del server.

Durante la fase di *test* del programma demone, ci siamo accorti che il processo demone *ftest* non termina propriamente quando il *server* esegue la chiamata *accept()* e la socket di controllo[3] é in modalitá bloccante. La *man page* di *accept()* riporta che quando la chiamata é interrotta da un segnale restituisce *-1* con *errno* posto uguale ad EINTR.

Nel nostro caso ció non accade perché *Linux* usa i segnali in stile BSD, cioé alcune chiamate di sistema vengono automaticamente riavviate. Ne segue che il demone puó essere terminato solo dal segnale SIGKILL.

Per evitare questa situazione possiamo agire nei seguenti modi:

1. configurare la *socket* di controllo come non bloccante. In questo caso la chiamata *accept()* non blocca, e restituisce -1 con *errno* uguale ad EGAIN se non ci sono richieste di connessione.

2. usare la chiamata *select()* prima di effettuare *accept()*. In caso di interruzione da parte di un segnale, la chiamata *select()* restituisce -1 con *errno* uguale ad EINTR, e quindi siamo in grado di gestire l'interruzione della chiamata da parte del segnale SIGTERM.

3. registrare il gestore del segnale SIGTERM con la funzione *sigaction()* (invece di *signal()*) non utilizzando la *flag* SA_RESTART. L'assenza di questa opzione implica che la chiamata di sistema non viene riavviata automaticamente.

Noi abbiamo usato il secondo metodo, ma abbiamo verificato che anche gli altri due funzionano correttamente .

[3] quella sulla quale viene effettuata la chiamata accept()