Programmazione avanzata su Linux

Traduzione italiana dell'opera

Advanced Unix Programming

di

Mark Mitchell, Jeffrey Oldham, Alex Samuel

Simone Sollena

2 settembre 2012

Indice

Capitolo 1

Programmazione UNIX avanzata con Linux

Questo capitolo ci mostra come effettuare i passi di base richiesti per creare un programma Linux in C o C++. In particolare, questo capitolo ci mostra come creare e modificare del codice sorgente C e C++, compilare questo codice, ed effettuare il debug del risultato. Se si è già abituati a programmare su Linux, si può andare avanti al Capitolo 2, "Scrivere del buon software GNU/Linux"; si presti molta attenzione alla sezione 2.3, "Scrivere ed usare librerie", in particolare riguardo al linking statico Vs. Dinamico che potresti non ancora conoscere.

Nel corso di questo libro, assumeremo che tu abbia familiarità con i linguaggi di programmazione C e C++. Assumeremo anche che tu sappia come eseguire semplici operazioni da riga di comando shell di Linux, come creare directory e copiare files. Poiché molti programmatori linux hanno cominciato a programmare in ambiente Windows, mostreremo occasionalmente similitudini e contrasti tra Windows e Linux.

1.1 Editare con *Emacs*

Un editor è il programma che si usa per editare il codice sorgente. Molti dei vari editor sono disponibili per Linux, ma l'editor più popolare e maggiormente pieno di funzionalità è probabilmente `Gnu Emacs`.

> **Riguardo Emacs**
> Emacs è molto più che un editor. È un programma incredibilmente potente, tanto che al CodeSourcery, è amichevolmente conosciuto come Un Vero Programma (One True Program) o giusto OTP per abbreviare. Da Emacs è possibile leggere ed inviare email ed è possibile personalizzarlo ed estenderlo in numerosissimi modi, troppi per poterne parlare qui. Da Emacs è possibile persino navigare su web!

Se hai familiarità con un altro editor, puoi certamente usare quello. Nulla nel resto di questo libro dipende dall'utilizzo di Emacs. Se invece non hai ancora un editor preferito su linux, dovresti andare avanti con il mini-tutorial fornito qui di seguito.

Se ti piace Emacs e vuoi imparare qulcosa in più riguardo le sue funzionalità avanzate, dovresti tenere in considerazione il fatto di leggere uno dei libri su Emacs disponibili. Un eccellente Tutorial è *Learning GNU Emacs*, scritto da Debra Cameron, Bill Rosenbaltt, e Eric S. Raymond (O'Reilly, 1996).

1.1.1 Aprire un file sorgente C o C++

È possibile avviare Emacs scrivendo `emacs` nella propria finestra di terminale e premendo il tasto invio. Quando Emacs è avviato, è possibile usare il menù in alto per creare un nuovo file sorgente. Clicca sul menù Files, scegli Apri Files, e quindi scrivi il nome del file che vuoi aprire nel "minibuffer" in basso allo schermo.[1] Se vuoi creare un file sorgente C, usa un nome di file che finisca in `.c` o in `.h`. Se vuoi creare un file sorgente C++, usa un nome di file che finisca in `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.C` o `.H`. Quando il file è aperto, puoi scrivere così come faresti in un programma di word-processing. Per salvare il file, scegli la voce Save Buffer nel menu Files. Quando avrai finito di usare Emacs, potrai scegliere l'opzione Exit Emacs nel menù Files.

Se non ti piace puntare e cliccare, puoi usare le scorciatoie da tastiera per aprire automaticamente i files, salvare i files, ed uscire da Emacs. Per aprire un file digita `C-x C-f` (`C-x` significa premere il tasto Control e quindi premere il tasto x). Per salvare un file digita `C-x C-s`. Per uscire da Emacs digita `C-x C-c`. Se vuoi prendere un po' più confidenza con Emacs scegli la voce "Emacs Tutorial" nel menù Help. Il tutoial ti fornisce alcuni consigli su come usare Emacs efficientemente.

1.1.2 Formattazione automatica

Se sei abituato alla programmazione in un Ambiente di Sviluppo Integrato (Integrated Development Environment – IDE), sarai anche abituato ad avere un editor che ti aiuti a formattare il codice. Emacs può fornire lo stesso tipo di funzionalità. Se apri un file sorgente C o C++, Emacs automaticamente mette in risalto il fatto che il file contiene del codice sorgente, e non del semplice testo. Se premi il tasto Tab in una riga vuota, Emacs muove il cursore in un punto di indentazione appropriato. Se premi il tasto di indentazione in una riga che contiene già del testo, Emacs indenta il testo. Così, per esempio, supponi di aver scritto quanto segue:

```
int main ()
{
printf ("Hello, world\n");
}
```

Se premi il tasto Tab sulla linea con la chiamata a `printf`, Emacs riformatterà il tuo codice cosicché sarà simile a questo:

```
int main ()
{
  printf ("Hello, world\n");
}
```

[1] Se non stai lavorando su un sistema X Window, devi premere F10 per accedere ai menù

Nota come la linea è stata appropriatamente indentata.

Non appena userai Emacs ancora per un po', vedrai come ti tornerà utile nell'esecuzione di tutti i tipi di compiti di formattazione complicati. Se sei ambizioso, puoi programmare Emacs per eseguire letteralmente ogni tipo di formattazione automatica immaginabile. Tanta gente ha usato questo tipo di funzionalità per implementare modelli per Emacs per editare proprio ogni tipo di documento, per implementare giochi[2] e per implementare front ends di database.

1.1.3 Evidenziazione della sintassi

Oltre alla formattazione del codice, Emacs può rendere facile leggere codice C e C++ colorando diversi elementi sintattici. Per esempio, Emacs può far apparire parole chiave di un colore, tipi di dati "built-in", come `int`, in un altro colore, e commenti ancora di un altro colore. L'utilizzo dei colori rende un po' più facile scoprire i comuni errori di sintassi.

Il modo più facile per far apparire la colorazione è editare il file `~/.emacs` ed inserire la seguente stringa:

```
(global-font-lock-mode t)
```

Salva il file, esci da Emacs, e riavvia. Adesso apri un file sorgente C o C++ e divertiti!

Potresti aver notato che la stringa che hai inserito nel tuo `.emacs` somiglia al codice di linguaggio di programmazione LISP. Ciò avviene perché questo *è* codice LISP! Molto di Emacs è attualmente scritto in LISP. È possibile aggiungere funzionalità a Emacs scrivendo altro codice LISP.

1.2 Compilare con GCC

Un compilatore trasforma codice sorgente leggibile dall'uomo in codice oggetto leggibile da macchina che può girare automaticamente. I compilatori di scelta sui sistemi Linux sono tutti parti del GNU Compiler Collection, solitamente conosciuti come GCC.[3] GCC include anche compilatori per C, C++, Java, C ad oggetti, Fortran, e Chill. Questo libro è principalmente focalizzato sulla programmazione C e C++.

Supponi di avere un progetto come quello del listato 1.2 con un file sorgente C++ (`reciprocal.cpp`) e un file sorgente C (`main.c`) come nel listato 1.1. Di questi due file si supponga che vengano compilati e quindi linkati assieme per generare un programma chiamato `recipocal`.[4] Questo programma calcolerà il reciproco di un intero.

Listato 1.1: (*main.c*) – **file sorgente C**

[2] Prova ad eseguire il comando `M-x dunnet` se vuoi giocare ad un vecchio affascinante gioco di avventura su testo

[3] Per ulteriori informazioni su GCC, visita http://gcc.gnu.org

[4] In Windows gli eseguibili solitamente hanno nomi che finisono in `.exe`. I programmi di Linux, d'altro canto, solitamente non hanno estensioni. Così, l'equivalente di Windows di questo programma sarebbe chiamato probabilmente `reciprocal.exe`; la versione Linux è semplicemente `reciprocal`

```
#include <stdio.h>
#include <stdlib.h>
#include "reciprocal.hpp"

int main (int argc, char **argv)
{
  int i;

  i = atoi (argv[1]);
  printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
  return 0;
}
```

Listato 1.2: (*reciprocal.cpp*) – **File sorgente C++**

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
  // I should be non-zero.
  assert (i != 0);
  return 1.0/i;
}
```

C'è anche un file header chiamato `reciprocal.hpp` (vedi listato 1.3).

Listato 1.3: (*reciprocal.hpp*) – **file header**

```
#ifdef __cplusplus
extern "C" {
#endif

extern  double reciprocal (int i);

#ifdef __cplusplus
}
#endif
```

Il primo passo è trasformare il codice sorgente C e C++ in codice oggetto.

1.2.1 Compilare un singolo file sorgente

Il nome del compilatore C è gcc. Per compilare un file sorgente C, usa l'opzione `-c`. Così, per esempio, l'inserimento di quanto segue, al prompt dei comandi, avvia la compilazione del file sorgente `main.c`

```
% gcc -c main.c
```

Il file oggetto risultante è chiamato `main.o`.

Il compilatore C++ è chiamato `g++`. Le sue operazioni sono molto simili a `gcc`; è possibile compilare `reciprocal.cpp` digitando quanto segue:

```
% g++ -c reciprocal.cpp
```

L'opzione `-c` dice a `g++` di compilare il programma solo in un file oggetto; senza questa, `g++` si aspetterebbe di dover fare il link del programma per produrre un eseguibile. Dopo aver digitato questo comando, otterrai un file oggetto chiamato `reciprocal.o`.

Probabilmente avrai bisogno di un altro paio di opzioni per compilare ogni programma ragionevolmente grande. L'opzione `-I` è usata per dire a GCC dove cercare i files header. Di default GCC cerca nella directory corrente e nelle directory dove sono installati gli headers per le librerie standard. Quindi se hai bisogno di includere dei files header che si trovano da qualche altra parte servirà l'opzione `-I`. Supponiamo ad esempio che il tuo progetto abbia una directory per i files sorgenti chiamata `src` ed un'altra chiamata `include`. Dovresti compilare `reciprocal.cpp` come in questo esempio per indicare che `g++` deve usare, in aggiunta, la directory `../include` per trovare `reciprocal.hpp`:

```
% g++ -c -I ../include reciprocal.cpp
```

A volte si vogliono definire delle macro nella riga di comando. Per esempio, nel codice di produzione, non si vuole il sovraccarico dovuto al controllo delle asserzioni, presente in `reciprocal.cpp`; questo serve solo a fare il debug del programma. È possibile disattivare tale controllo definendo la macro `NDEBUG`. Dovresti aggiungere un `#define` esplicito per `reciprocal.cpp`, ma ciò richiederebbe modifiche allo stesso codice sorgente. È più semplice definire piuttosto `NDEBUG` da riga di comando, come in questo caso:

```
% g++ -c -D NDEBUG recipocal.cpp
```

Se hai voluto definire `NDEBUG` con alcuni valori particolari, potresti aver fatto qualcosa del genere:

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

Se si sta compilando il codice per la produzione finale, probabilmente si vuole che GCC ottimizzi il codice cosicché giri il più veloce possibile. Ciò lo si può fare utilizzando l'opzione `-O2` da riga di comando. (GCC ha molti livelli di ottimizzazione; il secondo livello è il più appropriato per la maggior parte dei programmi.) Per esempio, il seguente, compila `reciprocal.cpp` con l'ottimizzazione attivata:

```
% g++ -c -O2 reciprocal.cpp
```

Nota che compilare con l'ottimizzazione può rendere il programma molto difficile da debuggare con un debugger (vedi sezione 1.4, "Debuggare con GNU Debugger (GDB)"). Inoltre, in certi casi, compilare con l'ottimizzazione può far venire fuori dei bugs che il programma non ha manifestato precedentemente.

Si possono passare molte altre opzioni a `gcc` e `g++`. Il miglior modo per ottenere una lista completa è vedere la documentazione online. Lo si può fare digitando quanto segue al prompt dei comandi:

```
% info gcc
```

1.2.2 Fare il link di file oggetto

Adesso che hai compilato `main.c` e `utilities.cpp`, vorrai fare il link. Per fare il link di un programma che contiene codice C++ si dovrebbe usare sempre `g++`, anche se contiene pure del codice C. Se il programma contiene solo codice C, si dovrebbe usare invece `gcc`. Poiché questo programma contiene sia codice C che C++, si dovrebbe usare `g++` in questo modo:

```
% g++ -o reciprocal main.o reciprocal.o
```

L'opzione `-o` fornisce il nome del file da generare come output del passaggio di link. Adesso puoi far girare `reciprocal` come in questo esempio:

```
% ./reciprocal 7
The reciprocal of 7 is 0.142857
```

Come puoi notare, `g++` ha automaticamente fatto il link con la libreria standard C di runtime contenente l'implementazione di printf. Se avessi avuto bisogno di fare il link con un'altra libreria (come una graphical user interface toolkit), avresti dovuto specificare la libreria tramite l'opzione `-l`. In Linux, i nomi di librerie quasi sempre cominciano con `lib`. Per esempio, la libreria Pluggable Authentication Module (PAM) è chiamata `libpam.a`. Per fare il link in `libpam.a`, usa un comando come questo:

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

Il compilatore aggiunge automaticamente il prefisso `lib` ed il suffissto `.a`.

Come per i files header, il linker cerca le librerie in alcune locazioni standard, includendo le directory `/lib` e `/usr/lib` che contengono le librerie standard di sistema. Se vuoi che il linker cerchi in altre directory, dovresti usare l'opzione `-L`, che è la "parallela" dell'opzione `-I` appena discussa. Puoi usare questa riga per dire al linker di cercare le librerie nella directory `/usr/local/lib/pam` prima di cercare nei soliti posti:

```
% g++ -o reciprocal main.o reciprocal.o
                -L/usr/local/lib/pam -lpam
```

Benché non ci sia bisogno di usare l'opzione `-I` per dire al preprocessore di cercare nella directory corrente, è necessario usare l'opzione `-L` per dire al linker di cercare nella directory corrente. In particolare, per dire al linker dove trovare la libreria test nella directory corrente, dovresti usare la seguente istruzione:

```
% gcc -o app app.o -L. -ltest
```

1.3 Automatizzare i processi con GNU Make

Se sei abituato a programmare per il sistema operativo Windows, sei probabilmente abituato a lavorare con un Integrated Developmente Environment (IDE). Aggiungi i files sorgenti al progetto e quindi l'IDE compila il tuo progetto automaticamente. Benché ci siano degli IDE disponibili per Linux, in questo libro non ne parleremo. Invece, questo libro ti mostrerà come usare GNU Make per automatizzare la ricompilazione del codice, che è ciò che fanno molti dei programmatori Linux.

L'idea di base che sta dietro a `make` è semplice. Dici a `make` quali *obiettivi* vuoi compilare, quindi fornisci delle *regole* spiegando come compilarli. Specifichi inoltre le dipendenze che indicano quando un particolare obiettivo dovrebbe essere ricompilato.

Nel nostro progetto di esempio `reciprocal` ci sono tre obiettivi ovvi: `reciprocal.o`, `main.o` ed il `reciprocal` stesso. Hai già le regole in mente per compilare questi obiettivi sotto forma di linea di comando, dati precedentemente. Le dipendenze richiedono qualche piccolo ragionamento in più. Chiaramente, `reciprocal` dipende da `reciprocal.o` e `main.o` poiché non è possibile fare il link del programma completo fino a che non hai compilato ognuno di questi file oggetto. I files oggetto vanno ricompilati ogni qualvolta il corrispondente file sorgente cambia. C'è un altro lato da tenere in considerazione: un cambiamento a `reciprocal.hpp` può causare il dover ricompilare entrambi i files oggetto poiché entrambi i files sorgenti includono quel file header.

In aggiunta agli obiettivi ovvi, dovrebbe esserci sempre un obiettivo `clean`. Quest'obiettivo rimuove tutti i files oggetto e programmi generati cosicché alla fine si abbia tutto pulito. La regola per questo obiettivo usa il comando rm per rimuovere i files.

Puoi dare tutte queste informazioni a make mettendo le informazioni in un file chiamato `Makefile`. Ecco ciò che è contenuto nel `Makefile`

```
reciprocal: main.o reciprocal.o
    g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
    gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
    g++ $(CFLAGS) -c reciprocal.cpp

clean:
    rm -f *.o reciprocal
```

Puoi vedere che gli obiettivi sono elencati sulla sinistra, seguiti da due punti e quindi ogni dipendenza. La regola per compilare quell'obiettivo sta sulla linea successiva. (Per il momento ignora il `$(CFLAGS)`.) La riga contenente la regola deve iniziare con un carattere di tabulazione, altrimenti make farà confusione. Se editi il `Makefile` in Emacs, Emacs ti aiuterà con la formattazione.

Se rimuovi i files oggetto che hai appena compilato, e digiti

```
% make
```

sulla riga di comando, vedrai quanto segue:

```
% make
gcc -c main.c
g++ -c reciprocal.cpp
g++ -o reciprocal main.o reciprocal.o
```

Puoi notare che `make` ha compilato automaticamente i file oggetto e fatto il link tra loro. Se adesso fai qualche modifica a `main.c` e digiti di nuovo `make`, vedrai quanto segue:

```
% make
gcc -c main.c
g++ -o reciprocal main.o reciprocal.o
```

Puoi notare che `make` ha saputo compilare `main.o` e rifare il link del programma, ma non si è preoccupato di ricompilare `reciprocal.cpp` poiché nessuna delle dipendenze per `reciprocal.o` ha subito cambiamenti.

Il `$(CFLAGS)` è una variabile di make. E' possibile definire questa variabile sia nello stesso `Makefile` che su riga di comando. `GNU make` sostituirà il valore della variabile durante l'esecuzione della regola. Così, per esempio, per ricompilare con l'ottimizzazione attivata, dovresti fare ciò:

```
% make clean
rm -f *.o reciprocal
% make CFLAGS=-O2
gcc -O2 -c main.c
g++ -O2 -c reciprocal.cpp
g++ -O2 -o reciprocal main.o reciprocal.o
```

Nota che il flag `-O2` è stato inserito nelle regole al posto di `$(CFLAGS)`.

In questa sezione hai visto solo le capacità più semplici di make. Puoi trovare altro digitando ciò:

```
% info make
```

Nel manale, troverai informazioni su come `make` riesca a mantenere un `Makefile` semplice, come ridurre il numero di regole necessarie, e come calcolare automaticamente le dipendenze. Inoltre puoi trovare altre informazioni in *GNU, Autoconf, Automake, and Libtool* di Gary V.Vaughan, Ben Elliston,Tom Tromey, e Ian Lance Taylor (New Riders Publishing, 2000).

1.4 Debuggare con GNU Debugger (GDB)

Il debugger è il programma che si usa per scoprire perché il proprio programma non si sta comportando nel modo in cui pensi dovrebbe comportarsi. Lo userai spesso.[5] GNU Debugger (`GDB`) è il debugger usato da molti programmatori Linux. puoi usare `GDB` per vedere il tuo codice passo passo, impostare dei punti di interruzione ed esaminare il valore delle variabili locali.

[5] ...finché il tuo porgramma non funzionerà per la prima volta

1.4.1 Compilare con Informazioni di Debugging

Per usare `GDB`, è necessario compilare con le informazioni di debugging abilitate. Per fare questo si aggiunge lo swtich `-g` sulla riga di comando di compilazione. Se stai usando un `Makefile` come descritto precedentemente, puoi giusto settare il `CFLAGS` uguale a `-g` quando esegui `make`, come mostrato qui:

```
% make CFLAGS=-g
gcc -g -c main.c
g++ -g -c reciprocal.cpp
g++ -g -o reciprocal main.o reciprocal.o
```

Quando compili con `-g`, il compilatore include informazioni extra nei files oggetto e negli eseguibili. Il debugger usa queste informazioni per mostrare quali indirizzi corriposndono a quali linee e a quali files sorgente, come stampare variabili locali, e così via

1.4.2 Eseguire GDB

Puoi avviare `gdb` digitando:

```
% gdb reciprocal
```

Quando `gdb` si avvia, dovresti vedere il prompt di `GDB`:

```
(gdb)
```

Il primo passo è quello di eseguire il tuo programma all'interno del debugger, Inserisci giusto il comando run ed ogni argomento del programma. Prova ad eseguire il programma senza nessun argomento, come questo:

```
(gdb) run
Starting program: reciprocal
Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287 strtol.c: No such file or directory.
(gdb)
```

Il problema è che non c'è controllo di errore nel codice `main`. Il programma si aspetta un argomento, ma in questo caso il programma è stato eseguito senza argomento. Il messaggio `SEGSEV` indica un crash del programma. `GDB` sa che il crash attuale è avvenuto in una funzione chiamata `__strtol_internal`. Questa funzione è nella libreria standard, e il sorgente non è installato, come spiega il messaggio "No such file or directory". E' possibile vedere lo stack usando il comando where:

```
(gdb) where
#0 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
#1 0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

Puoi vedere da questa schermata che il main ha chiamato la funzione `atoi` con un puntatore `NULL`, che è ciò che fa nascere i problemi.

Puoi salire di due livelli nello stack usando il comando `up` per raggiungere il `main`:

```
(gdb) up 2
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8 i = atoi (argv[1]);
```

Nota che `gdb` è capace di trovare il sorgente per `main.c` e mostra la riga dove è avvenuta la chiamata alla funzione dell'errore. E' possibile anche vedere il valore di variabili usando il comando `print`:

```
(gdb) print argv[1]
$2 = 0x0
```

Che conferma che il problema è certamente dovuto ad un puntatore `NULL` passato ad `atoi`.

Puoi impostare un breakpoint usando il comando `break`:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

Questo comando imposta un punto di interruzione (breakpoint) sulla prima linea del `main`.[6] Adesso prova ad eseguire il programma con un argomento, come questo:

```
(gdb) run 7
Starting program: reciprocal 7
Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8 i = atoi (argv[1]);
```

Puoi notare che il debugger si è fermato al breakpoint.

Puoi andare avanti dopo la chiamata ad `atoi` usando il comando `next`:

```
(gdb) next
9 printf (?The reciprocal of %d is %g\n?, i, reciprocal (i));
```

Se vuoi vedere cosa sta accadendo dentro `reciprocal`, usa il comando `step` come questo:

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6 assert (i != 0);
```

Adesso sei nel corpo della funzione `reciprocal`.

Potresti trovare molto conveniente eseguire `gdb` dall'interno di Emacs piuttosto che usare `gdb` da riga di comando. Usa il comando `M-x gdb` per avviare `gdb` in una finestra di Emacs. Se vieni fermato ad un breakpoint, Emacs tira fuori automaticamente il file appropriato. È facile capire cosa sta accadendo quando dai uno sguardo all'intero file piuttosto che giusto una riga di testo.

[6] Certa gente ha commentato che dire di interrompere il `main` è un po' bizzarro perchè solitamente questo lo si vuol fare solo quando il `main` è già interrotto

1.5 Trovare altre informazioni

Praticamente ogni distribuzione di Linux è provvista di una grande quantità di documentazione utile. Puoi imparare molto di ciò di cui parleremo in questo libro leggendo la documentazione della tua distribuzione Linux (anche se ti occuperà molto tempo). La documentazione non è sempre ben organizzata, per cui, il compito più difficile è quello di trovare ciò di cui hai bisogno. La documentazione è anche spesso datata, quindi prendi tutto ciò che leggi *"con le pinze"*. Se per esempio il sistema non si comporta esattamente come una *man page* (pagine di manuale) dice che dovrebbe comportarsi può darsi che la pagina di manuale sia obsoleta.

Per aiutarti a navigare, qui ci sono le più utili fonti di informazioni riguardo la programmazione Linux avanzata.

1.5.1 Pagine Man

Le distribuzioni Linux includono pagine di manuale per la maggior parte dei comandi standard, chiamate di sistema, e funzioni di librerie standard. Le pagine di manuale sono divise in sezioni numerate; per i programmatori, le più importanti sono queste:

- (1) Comandi utente
- (2) Chiamate di sistema
- (3) Funzioni di librerie standard
- (8) comandi di sistema/amministrativi

I numeri indicano le sezioni delle pagine di manuale. Le pagine di manuale di Linux sono installate nel tuo sistema; usa il comando `man` per accedere ad esse. Per trovare una pagina di manuale digita semplicemente `man nome`, dove `nome` è il nome di un comando o di una funzione. In alcuni casi lo stesso nome si trova in più di una sezione; puoi specificare esplicitamente la sezione anteponendo il numero della sezione al nome. Per esempio, se digiti quanto segue, otterrai la pagina di manuale per il comando `sleep` (nella sezione 1 delle pagine di manuale di Linux):

```
% man sleep
```

Per vedere la pagina di manuale per la funzione di libreria sleep, usa questo comando:

```
% man 3 sleep
```

Ogni pagina di manuale include un sommario on-line del comando o funzione. Il comando `whatis nome` mostra tutte le pagine di manuale (in tutte le sezioni) per un comando o una funzione corrispondente a quel `nome`. Se non sei sicuro di quale comando o funzione vuoi, puoi effettuare una ricerca per parola chiave sulle linee di sommario delle pagine di manuale, usando `man -k parolachiave`.

Le pagine di manuale includono una grande quantità di informazioni molto utili e dovrebbero essere il primo posto nel quale cercare aiuto. La pagina di manuale per un comando

descrive opzioni per la linea di comando e argomenti, input e output, codici di errori, configurazioni, e altre cose simili. La pagina di manuale per una chiamata di sistema o una funzione di libreria descrive parametri e valori di ritorno, lista di codici di errori ed effetti collaterali, e specifica quali files include usare per chiamare la funzione.

1.5.2 Info

Il sistema di documentazione Info contiene documentazione più dettagliata per molti componenti del cuore del sistema GNU/Linux, più molti altri programmi. Le pagine Info sono documenti ipertestuali, simili a pagine web. Per lanciare il browser Info basato su testo, digita `info` in una nuova finestra di shell. Ti verrà presentato un menù di documenti Info installati nel sistema. (Premi Control+H per visualizzare i tasti per navigare all'interno di un documento Info).

Tra i più utili documenti Info ci sono questi:

- `gcc` – Il compilatore gcc
- `libc` – La libreria GNU C, che include molte chiamate di sistema
- `gdb` – Il debugger GNU
- `emacs` – L'editor di testo Emacs
- `info` – Lo stesso sistema Info

Praticamente tutti gli strumenti standard di programmazione di Linux (inclusi `ld`, il linker; `as`, l'assemblatore, e `gprof`, il profiler) sono corredati di utili pagine Info. Puoi saltare direttamente ad un particolare documento Info specificando il nome della pagina da riga di comando:

```
% info libc
```

Se fai la maggior parte della tua programmazione in Emacs, puoi accedere direttamente al broswer interno Info digitando `M-x info` o `C-h i`.

1.5.3 File Header

È possibile imparare molto sulle funzioni di sistema disponibili e su come usarle guardando i files header del sistema. Questi risiedono in `/usr/include` e in `/usr/include/sys`. Se, per esempio, ottieni degli errori di compilazione dall'utilizzo di una chiamata di sistema, dai un'occhiata al file header corrispondente per verificare che la signature della funzione sia uguale a quella visualizzata nella pagina di manuale.

Sui sistemi linux, gran parte dei dettagli fondamentali su come lavorano le chiamate di sistema si riflette nei files header nelle directory `/usr/include/bits`, `/usr/include/asm`, e `/usr/include/linux`. Per esempio, i valori numerici dei segnali (descritti nella sezione 3.3, "Segnali", nel Capitolo 3, "Processi") sono definiti in `/usr/include/bits/signum.h`. Questi file header facilitano la lettura alle menti curiose. Tuttavia, non includere direttamente questi

nel programma; usa sempre i file header in `/usr/include` come menzionato nella pagina di manuale per la funzione che stai usando.

1.5.4 Codice sorgente

Siamo in ambito open source, no? Ciò che regola il modo in cui il sistema lavora alla fine è lo stesso codice sorgente del sistema e fortunatamente per i programmatori Linux, quel codice sorgente è gratuitamente disponibile. C'è la possibilità che la tua distribuzione Linux includa tutto il codice sorgente dell'intero sistema e di tutti i programmi inclusi in essa; se non è così, sei autorizzato, secondo i termini della GNU General Public License, a richiederlo al distributore. (Comunque, il codice sorgente potrebbe non essere stato installato sull'hard disk. Guarda la documentazione della tua distribuzione per le istruzioni su come installarlo).

Il codice sorgente per lo stesso kernel di linux è di solito memorizzato in `/usr/src/linux`. Se questo libro ti incuriosisce sui dettagli di come lavorano i processi, la memoria condivisa e le periferiche di sistema, puoi sempre imparare direttamente dal codice sorgente. Molte delle funzioni descritte in questo libro sono implementate con librerie GNU C; controlla la documentazione della tua distribuzione per la posizione del codice sorgente delle librerie C.

Capitolo 2

Scrivere del buon software GNU/Linux

Questo capitolo copre alcune tecniche di base che usano molti programmatori GNU/Linux. Seguendo le linee guida presentate, sarai in grado di scrivere programmi che lavorano bene nell'ambiente GNU/Linux ed andare incontro alle aspettative degli utente GNU/Linux su come dovrebbero funzionare i programmi.

2.1 Interazione con l'ambiente di esecuzione

Quando hai cominciato a studiare C o C++, avrai appreso che la funzione speciale `main` è il punto di inizio primario di un programma. Quando il sistema operativo esegue il tuo programma, esso fornisce automaticamente certe funzionalità che aiutano il programma a comunicare con il sistema operativo e con l'utente. Probabilmente avrai anche appreso riguardo ai due parametri del `main`, solitamente chiamati `argc` e `argv`, che ricevono l'input per il tuo programma. Hai appreso anche a riguardo dello `stdout` e `stdin` (o gli stream `cout` e `cin` in C++) che forniscono funzioni di input e output per la console. Queste funzionalità sono fornite dai linguaggi C e C++, ed essi interagiscono in certi modi con il sistema operativo GNU/Linux. GNU/Linux fornisce anche altri modi per interagire con l'ambiente operativo.

2.1.1 La lista degli argomenti

Esegui un programma dal prompt di shell digitando il nome del programma. Opzionalmente, puoi fornire al programma informazioni aggiuntive digitando una o più parole dopo il nome del programma, separate da spazi. Queste sono chiamate *argomenti da riga di comando* (*command-line arguments*). (Puoi anche includere un argomento che contiene spazi, racchiudendolo tra virgolette). Più generalmente, si fa riferimento a questi anche come la *lista degli argomenti* del programma perché non è necessario che essi abbiano origine dalla riga di comando di shell. Nel Capitolo 3, "Processi", vedrai un altro modo di chiamare un programma,

nel quale un programma può specificare direttamente la lista degli argomenti di un altro programma.

Quando un programma è chiamato da shell, la lista degli argomenti contiene l'intera linea di comando, includendo il nome del programma ed ogni argomento della riga di comando che può essere stato fornito. Supponi, per esempio, di chiamare il comando `ls` nella shell per mostrare il contenuto della directory root e le corrispondenti dimensioni dei files con questa riga:

```
% ls -s /
```

La lista di argomenti che il programma riceve ha tre elementi. Il primo è il nome del programma stesso, come specificato dalla riga di comando, chiamato `ls`. Il secondo ed il terzo elemento della lista degli argomenti sono due argomenti da riga di comando, `-s` e `/`.

La funzione `main` del tuo programma può accedere alla lista degli argomenti tramite i parametri `argc` e `argv` del main (se non li usi, li puoi semplicemente omettere). Il primo parametro, `argc`, è un intero che è settato al numero di elementi nella lista degli argomenti. Il secondo parametro, `argv`, è un `array` di puntatori a caratteri. La dimensione dell'array è argc e gli elementi dell'array puntano agli elementi della lista degli argomenti, come stringhe di caratteri "NUL-terminated".

Usare gli argomenti da riga di comando è tanto facile quanto esaminare il contenuto di `argc` e `argv`. Se non sei interessato al nome sesso del programma, non dimenticare di saltare il primo elemento.

Il listato 2.1 dimostra come usare `argc` e `argv`.

Listato 2.1: (***arglist.c***) – **Usare `argc` e `argv`**

```
1  #include <stdio.h>
2
3  int main (int argc, char* argv[])
4  {
5    printf ("The name of this program is '%s'.\n", argv[0]);
6    printf ("This program was invoked with %d arguments.\n", argc - 1);
7
8    /* Were any command-line arguments specified? */
9    if (argc > 1) {
10     /* Yes, print them. */
11     int i;
12     printf ("The arguments are:\n");
13     for (i = 1; i < argc; ++i)
14       printf ("  %s\n", argv[i]);
15   }
16
17   return 0;
18 }
```

2.1.2 Convenzioni GNU/Linux della riga di comando

Quasi tutti i programmi GNU/Linux obbediscono ad alcune convenzioni su come sono interpretati gli argomenti della riga di comando. Gli argomenti che i programmi si aspettano rientrano in due categorie: opzioni (o flag) e altri argomenti. Le opzioni modificano il modo in cui si comporta il programma, mentre gli altri argomenti forniscono gli input (per esempio, i nomi dei file di input).

Le opzioni sono di due forme:

- opzioni brevi consistono in un singolo trattino ed un singolo carattere (di solito una lettere minuscola o una lettera maiuscola). Le opzioni brevi sono veloci da digitare.
- Opzioni lunghe consistono in due trattini, seguiti da un nome fatto di lettere minuscole e maiuscole e trattini. Le opzioni lunghe sono facili da ricordare e facili da leggere (ad esempio, negli script di shell).

Di solito, un programma le fornisce entrambe; una forma breve ed una forma lunga per molte delle opzioni che supporta, una per brevità e l'altra per chiarezza. Per esempio, molti programmi capiscono le opzioni `-h` e `--help`, e le trattano in modo identico. Normalmente, quando un programma è invocato dalla shell, ogni opzione che si vuole segue immediatamente il nome del programma. Alcune opzioni si aspettano di essere seguite immediatamente da un argomento. Molti programmi, per esempio, interpretano l'opzione `--output foo` per specificare che l'output del programma dovrebbe essere memorizzato in un file chiamato foo. Dopo le opzioni, possono seguire altri argomenti di riga di comando, tipicamente files di input o input di dati.

Per esempio, il comando `ls -s /` mostra il contenuto della directory root. L'opzione `-s` modifica il comportamento di default di `ls` dicendogli di mostrare la dimensione (in kilobytes) di ogni voce. L'argomento `/` dice a `ls` di quale directory fare il listato. L'opzione `--size` è sinonima di `-s`, così lo stesso comando può essere invocato come `ls --size /`.

Gli standard di codifica GNU (*GNU Coding Standards*) elencano i nomi di alcune opzioni di riga di comando comunemente usati. Se prevedi di fornire ogni opzione simile a queste, sarebbe una buona idea usare i nomi specificati negli standard di codifica. Puoi vedere le linee guida degli standard di codifica GNU per le opzioni di riga di comando invocando il seguente da un prompt di shell in molti sistemi GNU/Linux:

```
% info "(standards)User Interfaces"
```

2.1.3 Usare `getopt_long`

Analizzare le opzioni di riga di comando è un compito fastidioso. Fortunatamente, le librerie GNU C forniscono una funzione che puoi usare nei programmi C e C++ per rendere questo lavoro un po' più facile (anche se resta sempre un po' noioso). Questa funzione, `getopt_long`, capisce entrambe le opzioni, brevi e lunghe. Se usi questa funzione, includi il file header `<getopt.h>`. Supponi, per esempio, di stare scrivendo un programma che accetta le tre opzioni mostrate nella tabella 2.1.

Tabella 2.1: Esempi di opzioni di programmi

Forma breve	Forma lunga	Scopo
`-h`	`--help`	Mostra un sommario d'uso ed esce
`-o nomefile`	`--output nomefile`	Specifica il nome del file di output
`-v`	`--verbose`	Stampa messaggi dettagliati

In più, il programma accetta zero o più argomenti da linea di comando aggiuntivi, che sono i nomi dei files di input.

Per specificare le opzioni lunghe disponibili, puoi costruire un array di elementi `struct option`

Per usare `getopt_long`, devi fornire due strutture dati, La prima è una stringa di caratteri che contiene le opzioni brevi valide, ognuna una singola lettera. Un'opzione che richiede un argomento è seguita da due punti, Per il tuo programma, la stringa `ho:v` indica che le opzioni valide sono `-h`, `-o` e `-v`, con la seconda di queste opzioni seguita da un argomento.

Per specificare le opzioni lunghe disponibili, puoi costruire un array di elementi `struct option`. Ogni elemento corrisponde a una opzione lunga ed ha quattro campi. In circostanze normali, il primo campo è il nome dell'opzione lunga (come una stringa di caratteri, senza i due trattini); il secondo è 1 se l'opzione prende un argomento o 0 altrimenti; Il terzo è NULL; e il quarto è un carattere costante che specifica l'opzione breve sinonima per quella lunga. L'ultimo elemento dell'array dovrebbe essere tutto zeri. Puoi costruire un array come questo:

```
const struct option long_options[] = {
  { "hel"?, 0, NULL, 'h' },
  { "output", 1, NULL, 'o' },
  { "verbose", 0, NULL, 'v' },
  { NULL, 0, NULL, 0 }
};
```

Invochi la funzione `getopt_long`, passantole gli argomenti `argc` e `argv` dal main, la stringa di caratteri che descrive le opzioni brevi e l'array di elementi `struct option` descrive le opzioni lunghe.

- Ogni volta che chiami `getopt_long`, esso verifica una singola opzione, restituendo la lettera dell'opzione breve per quell'opzione o `-1` se non sono trovate altre opzioni.

- Tipicamente, chiamerai `getopt_long` in un `loop`, per processare tutte le opzioni che l'utente ha specificato e tratterai le opzioni specifiche all'interno di uno switch.

- Se `getopt_long` incontra una opzione non valida (un'opzione che non hai specificato come una opzione breve o lunga valida), stampa un messaggio di errore e restituisce il carattere `?` (un marcatore di domanda). Molti programmi escono come risposta a questo, possibilmente dopo aver mostrato un messaggio di informazioni sull'utilizzo.

- Quando usi un'opzione che prende un argomento, la variabile globale `optarg` punta al testo di quell'argomento.
- Dopo che `getopt_long` ha finito di elaborare tutte le opzioni, la variabile globale `optind` contiene l'indice (in `argv`) del primo argomento che non sia un'opzione.

Il listato 2.2 mostra un esempio di come dovresti usare `getopt_long` per processare i tuoi argomenti.

Listato 2.2: (***getopt_long.c***) – **Usare** `getopt_long`

```
1  #include <getopt.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  /* The name of this program.  */
6  const char* program_name;
7
8  /* Prints usage information for this program to STREAM (typically
9     stdout or stderr), and exit the program with EXIT_CODE.  Does not
10    return.  */
11
12 void print_usage (FILE* stream, int exit_code)
13 {
14   fprintf (stream, "Usage:␣␣%s␣options␣[␣inputfile␣...␣]\n", program_name);
15   fprintf (stream,
16            "␣␣-h␣␣--help␣␣␣␣␣␣␣␣␣␣␣␣Display␣this␣usage␣information.\n"
17            "␣␣-o␣␣--output␣filename␣␣Write␣output␣to␣file.\n"
18            "␣␣-v␣␣--verbose␣␣␣␣␣␣␣␣␣Print␣verbose␣messages.\n");
19   exit (exit_code);
20 }
21
22 /* Main program entry point.  ARGC conains number of argument list
23    elements; ARGV is an array of pointers to them.  */
24
25 int main (int argc, char* argv[])
26 {
27   int next_option;
28
29   /* A string listing valid short options letters.  */
30   const char* const short_options = "ho:v";
31   /* An array describing valid long options.  */
32   const struct option long_options[] = {
33     { "help",     0, NULL, 'h' },
34     { "output",   1, NULL, 'o' },
35     { "verbose",  0, NULL, 'v' },
36     { NULL,       0, NULL, 0   }   /* Required at end of array.  */
37   };
38
```

```
39  /* The name of the file to receive program output, or NULL for
40     standard output.  */
41  const char* output_filename = NULL;
42  /* Whether to display verbose messages.  */
43  int verbose = 0;
44
45  /* Remember the name of the program, to incorporate in messages.
46     The name is stored in argv[0].  */
47  program_name = argv[0];
48
49  do {
50    next_option = getopt_long (argc, argv, short_options,
51                               long_options, NULL);
52    switch (next_option)
53    {
54    case 'h':   /* -h or --help */
55      /* User has requested usage information.  Print it to standard
56         output, and exit with exit code zero (normal termination).  */
57      print_usage (stdout, 0);
58
59    case 'o':   /* -o or --output */
60      /* This option takes an argument, the name of the output file.  */
61      output_filename = optarg;
62      break;
63
64    case 'v':   /* -v or --verbose */
65      verbose = 1;
66      break;
67
68    case '?':   /* The user specified an invalid option.  */
69      /* Print usage information to standard error, and exit with exit
70         code one (indicating abonormal termination).  */
71      print_usage (stderr, 1);
72
73    case -1:    /* Done with options.  */
74      break;
75
76    default:    /* Something else: unexpected.  */
77      abort ();
78    }
79  }
80  while (next_option != -1);
81
82  /* Done with options.  OPTIND points to first non-option argument.
83     For demonstration purposes, print them if the verbose option was
84     specified.  */
85  if (verbose) {
86    int i;
87    for (i = optind; i < argc; ++i)
```

```
88          printf ("Argument: %s\n", argv[i]);
89      }
90
91      /* The main program goes here.  */
92
93      return 0;
94  }
```

Usare `getopt_long` può sembrare un po' laborioso, ma scrivere del codice da soli per analizzare le opzioni della riga di comando potrebbe richiedere anche più tempo. La funzione `getopt_long` è molto sofisticata e permette una grande flessibilità nello specificare che tipo di opzioni accettare. Comunque, è una buona idea starsene lontani dalle funzionalità molto avanzate e lavorare con le opzioni della struttura di base descritte.

2.1.4 I/O Standard

Le librerie C standard forniscono stream di input e output (rispettivamente `stdin` e `stdout`). Queste sono usate da `scanf`, `printf` e altre funzioni di libreria. Nella tradizione UNIX, l'uso di standard input ed output è d'abitudine per i programmi GNU/Linux. Ciò permette di concatenare programmi multipli usando le pipes di shell e la ridirezione di input ed output. (Vedi la pagina di manuale della tua shell per impararne la sintassi).

La libreria C fornisce anche lo `stderr`, lo stream standard error. I programmi dovrebbero stampare warning e messaggi di errore sullo standard error invece dello standard output. Questo permette agli utenti di separare output normali e messaggi di errore, per esempio, redirezionando lo standard output ad un file mentre permettendo allo standard error di stampare sulla console. La funzione `fprintf` può essere usata per stampare sullo `stderr`, per esempio:

```
fprintf (stderr, ("Error: ..."));
```

Questi tre stream sono anche accessibili con i comandi di base di UNIX I/O (read, write, e così via) tramite i descrittori di file o pipe. La sintassi per fare ciò varia a seconda delle shell; per le shell stile Bourne (inclusa bash, la shell di default in molte distribuzioni GNU/Linux), la sintassi è questa:

```
% program > output_file.txt 2>&1
% program 2>&1 | filter
```

La sintassi `2>&1` indica che il descrittore di file 2 (`stderr`) dovrebbe essere unita al descrittore di file 1 (`stdout`). Nota che `2>&1` deve seguire una ridirezione di file (il primo esempio) ma deve precedere una ridirezione di pipe (il secondo esempio).

Nota che `stdout` è bufferizzato. I dati scritti sullo `stdout` non sono inviati alla console (o altre periferiche, se è rediretto) fino a che il buffer non si riempie, finché il programma esce normalmente, o finché stdout è chiuso. Puoi azzerare il buffer chiamando la seguente:

```
fflush (stdout);
```

Di contro, `stderr` non è bufferizzato; i dati che sono scritti sullo `stderr` vanno direttamente alla conole.[1]

Questo può produrre qualche risultato sorprendente. Per esempio, questo loop non stampa una frase al secondo; invece, le frasi sono bufferizzate ed un po' di queste vengono stampate assieme quando il buffer è pieno.

```
while (1) {
    printf (".");
    sleep (1);
}
```

In questo loop, comunque, il punto compare una volta ogni secondo:

```
while (1) {
    fprintf (stderr, ".");
    sleep (1);
}
```

2.1.5 Codici di uscita del programma

Quando un programma termina, indica il proprio stato con un codice di uscita. Il codice di uscita è un piccolo numero intero; per convenzione, un codice di uscita pari a zero indica un'esecuzione senza problemi, mentre un codice di uscita diverso da zero indica che c'è stato un errore. Alcuni programmi usano differenti valori diversi da zero per i codici di uscita per distinguere errori specifici.

Con molte shell è possibile ottenere i codici di uscita del programma eseguito più recentemente con la variabile speciale `$?`. Qui c'è un esempio in cui il comando `ls` viene chiamato due volte ed i suoi codici di uscita sono stampati dopo ogni chiamata. Nel primo caso, `ls` viene eseguito correttamente e restituisce il codice di uscita zero. Nel secondo caso, `ls` restituisce un errore (perché il nome del file specificato sulla riga di comando non esiste) e quindi restituisce un codice di errore diverso da zero.

```
% ls /
bin coda etc lib misc nfs proc sbin usr
boot dev home lost+found mnt opt root tmp var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Un programma C o C++ specifica il proprio codice di uscita restituendo quel valore dalla funzione `main`. Ci sono altri metodi usati per fornire codici di uscita, e codici di uscita speciali

[1] In C++ la stessa distinzione persiste, rispettivamente per `cout` e `cerr`. Nota che il token `endl` svuota uno stream ed in più stampa un carattere di nuova riga; se non vuoi svuotare lo stream (per esempio, per ragioni di performance), usa la costante di nuova linea `'\n'`

sono assegnati al programma che non termina in modo normale (per un segnale). Di questi si parlerà più avanti nel Capitolo 3.

2.1.6 L'ambiente

GNU/Linux fornisce ogni programma in esecuzione con un *ambiente*. L'ambiente è una raccolta di coppie variabile/valore. Entrambi, nomi delle variabili d'ambiente ed i loro valori sono stringhe di caratteri. Per convenzione, i nomi delle variabili d'ambiente sono scritti con tutte le lettere maiuscole.

Probabilmente avrai già familiarità con molte delle comuni variabili d'ambiente. Per esempio:

- USER Contiene il tuo nome utente.
- HOME Coniene il percorso della tua home directory.
- PATH Contiene una lista di directory separate dai due punti nelle quali Linux cerca i comandi che invochi
- DISPLAY Contiene il nome ed il numero di display del server X Window nel quale appariranno le finestre dell'interfaccia grafica X Window.

La tua shell, come altri programmi, ha un ambiente. Le Shell forniscono metodi per esaminare e modificare direttamente l'ambiente. Per stampare l'ambiente corrente nella tua shell invoca il programma `printenv`. Diverse shell hanno diverse sintassi per l'utilizzo delle variabili d'ambiente; la seguente è la sintassi per le shell stile Bourne.

- La shell crea automaticamente una variabile di shell per ogni variabile d'ambiente che trova, così puoi accedere ai valori delle variabili d'ambiente usando la sintassi `$varname`. Per esempio:

```
% echo $USER
samuel
% echo $HOME
/home/samuel
```

- Puoi usare il comando `export` per esportare una variabile di shell nell'ambiente. Per esempio, per settare la variabile d'ambiente `EDITOR`, dovresti usare questo:

```
% EDITOR=emacs
% export EDITOR
```

  O più brevemente

```
% export EDITOR=emacs
```

In un programma, puoi accedere ad una variabile d'ambiente con la funzione `getenv` in `<stdlib.h>`. Questa funzione prende il nome di una variabile e ne restituisce il valore corrispondente come stringa di caratteri, o NULL se la variabile non è definita nell'ambiente. Per settare o cancellare variabili d'ambiente, usa rispettivamente le funzioni `setenv` e `unsetenv`.

Enumerare tutte le variabili d'ambiente è un po' difficile. Per farlo puoi accedere ad una speciale variabile globale chiamata `environ`, definita nella libreria GNU C. Questa variabile, di tipo `char**`, è un array di puntatori a stringhe di caratteri che terminano con NULL. Ogni stringa contiene una variabile d'ambiente, nella forma `VARIABILE=valore`.

Il programma del listato 2.3, per esempio, stampa semplicemente l'intero ambiente facendo un loop nell'`array environ`.

Listato 2.3: (***print-env.c***) – **Stampa l'ambiente di esecuzione**

```
1  #include <stdio.h>
2
3  /* The ENVIRON variable contains the environment.  */
4  extern char** environ;
5
6  int main ()
7  {
8    char** var;
9    for (var = environ; *var != NULL; ++var)
10     printf ("%s\n", *var);
11   return 0;
12 }
```

Non modificare `environ` tu stesso, ma piuttosto usa le funzioni `setevn` e `unsetenv`.

Di solito, quando viene avviato un nuovo programma, esso eredita una copia dell'ambiente del programma che lo ha invocato (il programma shell, se è stato invocato interattivamente). Così, per esempio, i programmi che esegui dalla shell, possono esaminare i valori delle variabili d'ambiente che hai settato nella shell.

Le variabili d'ambiente sono comunemente usate per comunicare informazioni di configurazione ai programmi. Supponi, per esempio, di star scrivendo un programma che si collega ad un server internet per ottenere alcune informazioni. Potresti scrivere il programma in modo tale che il nome del server sia specificato da riga di comando. Comunque, supponi che il nome del server non sia qualcosa che l'utente cambia molto spesso. Puoi usare in questo caso una speciale variabile d'ambiente – detta `SERVER_NAME` – per specificare il nome del server; se questa variabile non esiste, viene usato un valore di default. Una parte del tuo programma potrebbe somigliare a quello mostrato nel listato 2.4.

Listato 2.4: (***client.c***) – **Parte di un programma client di rete**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main ()
5  {
```

```
6   char* server_name = getenv ("SERVER_NAME");
7   if (server_name == NULL)
8     /* The SERVER_NAME environment variable was not set.  Use the
9        default.  */
10    server_name = "server.my-company.com";
11
12  printf ("accessing server %s\n", server_name);
13  /* Access the server here...  */
14
15  return 0;
16 }
```

Sopponi che il programma sia chiamato `client`. Assumendo che non hai settato la variabile `SERVER_NAME`, viene utilizzato il valore di default per il nome del server:

```
% client
accessing server server.my-company.com
```

Ma è facile specificare un server diverso:

```
% export SERVER_NAME=backup-server.elsewhere.net
% client
accessing server backup-server.elsewhere.net
```

2.1.7 Usare files temporanei

A volte un programma necessita di creare un file temporaneo, per memorizzare dati per un po' o passare dati ad un altro programma. Sui sistemi GNU/Linux, i file temporanei sono memorizzati nella directory /tmp. Quando usi i file temporanei, dovresti essere già informato delle seguente insidie:

- Più di una istanza del tuo programma può essere eseguita simultaneamente (dallo stesso utente o diversi utenti). Le istanze dovrebbero usare diversi nomi dei files temporanei cosicché non vadano in collisione.
- I permessi dei file temporanei dovrebbero essere settati in modo che utenti non autorizzati non possano alterare l'esecuzione del programma modificando o sostituendo un file temporaneo.
- I nomi dei file temporanei dovrebbero essere generati in modo tale che non possano essere previsti esternamente; altrimenti un attaccante potrebbe sfruttare il ritardo nel testing se un nome dato è già in uso ed aprire un nuovo file temporaneo.

GNU/Linux fornisce delle funzioni, `mkstemp` e `tmpfile` che si curano da sole di questi problemi al posto tuo (in aggiunta a molte funzioni che non lo fanno). Quale userai dipende dal fatto che tu voglia passare il file temporaneo ad un altro programma e se vuoi usare le funzioni I/O di UNIX (`open`, `write`, e così via) o le funzioni di stream I/O delle librerie C (`fopen`, `fprintf`, e così via).

Usare *mkstemp* La funzione `mkstemp` crea un file temporaneo il cui nome è univoco da un modello di nomi di file, crea il file con gli opportuni permessi cosicché solo l'utente corrente vi possa accedere e aprire il file per lettura/scrittura. Il modello del nome del file è una stringa di caratteri che finisce con "XXXXXX" (sei X maiuscole); `mkstemp` sostituisce le X con dei caratteri in modo tale che il nome del file sia univoco. Il valore di ritorno è un descrittore di file; a questo punto usa le funzioni della famiglia `write` per scrivere sul file temporaneo.

I file temporanei creati con `mkstemp` non vengono cancellati automaticamente. E' compito tuo rimuovere il file temporaneo quando non è più usato. (I programmatori dovrebbero fare attenzione a cancellare i file temporanei; altrimenti il file system /tmp si riempirà, rendendo il sistema non funzionante). Se il file temporaneo serve solo per l'utilizzo interno e non verrà passato ad un altro programma, sarebbe una buona idea chiamare `unlink` nel file temporaneo immediatamente. La funzione `unlink` rimuove la voce della directory corrispondente al file, ma poiché i file in un file system hanno dei riferimenti, il file stesso non verrà rimosso fino a che non ci saranno più dei descrittori di file per quel file. Per questi motivi il tuo programma potrà continuare ad usare il file temporaneo, ed il file andrà via automaticamente non appena chiuderai il descrittore del file. Poiché Linux chiude i descrittori di file quando un programma termina, il file temporaneo verrà rimosso anche se il programma dovesse terminare in modo imprevisto.

La coppia di funzioni nel listato 2.5 dimostra `mkstemp`. Usate insieme, queste funzioni facilitano la scrittura di un buffer di memoria su un file temporaneo (in modo tale che la memoria possa essere liberata o riutilizzata) e rileggerlo successivamente.

Listato 2.5: **(*temp_file.c*) – utilizzo di mkstemp**

```
1  #include <stdlib.h>
2  #include <unistd.h>
3
4  /* A handle for a temporary file created with write_temp_file.  In
5     this implementation, it's just a file descriptor.  */
6  typedef int temp_file_handle;
7
8  /* Writes LENGTH bytes from BUFFER into a temporary file.  The
9     temporary file is immediately unlinked.  Returns a handle to the
10    temporary file.  */
11
12 temp_file_handle write_temp_file (char* buffer, size_t length)
13 {
14   /* Create the filename and file.  The XXXXXX will be replaced with
15      characters that make the filename unique.  */
16   char temp_filename[] = "/tmp/temp_file.XXXXXX";
17   int fd = mkstemp (temp_filename);
18   /* Unlink the file immediately, so that it will be removed when the
19      file descriptor is closed.  */
20   unlink (temp_filename);
21   /* Write the number of bytes to the file first.  */
22   write (fd, &length, sizeof (length));
```

```
23    /* Now write the data itself.  */
24    write (fd, buffer, length);
25    /* Use the file descriptor as the handle for the temporary file.  */
26    return fd;
27  }
28
29  /* Reads the contents of a temporary file TEMP_FILE created with
30     write_temp_file.  The return value is a newly-allocated buffer of
31     those contents, which the caller must deallocate with free.
32     *LENGTH is set to the size of the contents, in bytes.  The
33     temporary file is removed.  */
34
35  char* read_temp_file (temp_file_handle temp_file, size_t* length)
36  {
37    char* buffer;
38    /* The TEMP_FILE handle is a file descriptor to the temporary file.  */
39    int fd = temp_file;
40    /* Rewind to the beginning of the file.  */
41    lseek (fd, 0, SEEK_SET);
42    /* Read the size of the data in the temporary file.  */
43    read (fd, length, sizeof (*length));
44    /* Allocate a buffer and read the data.  */
45    buffer = (char*) malloc (*length);
46    read (fd, buffer, *length);
47    /* Close the file descriptor, which will cause the temporary file to
48       go away.  */
49    close (fd);
50    return buffer;
51  }
```

Usare *tmpfile* Se stai usando le funzioni di I/O della libreria C e non hai necessità di passare il file temporaneo ad un altro programma, puoi usare la funzione tmpfile. Questa crea ed apre un file temporaneo, e restituisce un puntatore di file che punta a questo. Il file temporaneo è già *"unlinked"*, come nell'esempio precedente, così viene cancellato automaticamente quando il puntatore al file viene chiuso (con `fclose`) o quando il programma termina.

GNU/Linux fornisce molte altre funzioni per generare file temporanei e nomi di file temporanei, inclusi `mktemp`, `tmpnam` e `tempnam`. Comunque, non usare queste funzioni poiché sono soggette ai problemi di sicurezza già descritti.

2.2 Codifica difensiva

Scrivere programmi che girano correttamente durante l'utilizzo "normale" è difficile; scrivere programmi che si comportino decentemente in situazioni errori è ancora più difficile. Questa sezione dimostra alcune tecniche di codifica per scovare velocemente i bug e per trovare e riprendersi dai problemi in un programma in esecuzione.

Gli esempi di codice presentati successivamente in questo libro saltano volutamente il codice di controllo degli errori e di recupero poiché ciò potrebbe oscurare il funzionamento di base che viene presentato. Comunque, l'esempio finale nel **??**, "**??**", ritorna a dimostrare come usare queste tecniche per scrivere programmi robusti.

capitolo 11, "Un'applicazione di esempio GNU/Linux"

2.2.1 Usare assert

Una buona cosa da tenere in mente quando si codificano programmi applicativi è che i bug o gli errori inaspettati potrebbero causare il fallimento dei programmi drammaticamente, il più presto possibile. Ciò ti aiuterà a trovare bug al più presto durante i cicli di sviluppo e testing. I fallimenti che non vengono fuori da soli sono spesso dimenticati e non si mostrano fino a che l'applicazione non è nelle mani dell'utente.

Uno dei metodi più semplici per scovare condizioni inaspettate è la macro `assert` dello standard C. L'argomento da passare a questa macro è un'espressione Booleana. Il programma viene terminato se ciò che viene valutato dall'espressione è falso, dopo aver stampato un messaggio d'errore contenente il file sorgente, il numero di riga ed il testo dell'espressione. La macro `assert` è molto utile per una grande varietà di controlli di consistenza interni al programma. Per esempio, usa `assert` per testare la validità degli argomenti di una funzione, per testare le precondizioni e postcondizioni delle chiamate di funzioni (e chiamate di metodi in C++), e per testare dei valori di ritorno inaspettati.

Ogni utilizzo di `assert` lavora non solo come un controllo di una condizione a runtime, ma anche come documentazione sul funzionamento del programma all'interno del codice sorgente. Se il tuo programma contiene un `assert (condizione)` ciò dice a qualcuno che sta leggendo il tuo codice sorgente che quella `condizione` dovrebbe essere sempre vera in quel punto del programma, e se la `condizione` non è vera, probabilmente c'è un bug nel programma.

Per il codice che deve avere delle prestazioni ottimali, le verifiche a runtime come l'uso di `assert` può imporre significanti penalità in termini di prestazioni. In questo caso, puoi compilare il codice definendo la macro `NDEBUG`, tramite l'utilizzo del flag `-NDEBUG` da riga di comando del compilatore. Con `NDEBUG` settato, le occorrenze della macro `assert` verranno preprocessate via. E' una buona idea farlo solo quando necessario per ragioni di prestazioni e solo in casi critici di file sorgenti che devono avere certe prestazioni.

Poiché è possibilie preprocessare via le macro assert, stai molto attento al fatto che ogni espressione utilizzata con asert non abbia effetti collaterali. Specialmente, non andrebbero chiamate delle funzioni all'interno dell'espressione assert, assegnare variabili o usare operatori di modifica come ++. Supponi, per esempio, di chiamare una funzione `do_something` ripetutamente in un loop. La funzione `do_something` restituisce zero in caso di successo e un valore diverso da zero in caso di fallimento, ma ti aspetti che la funzione non fallisca mai nel tuo programma. Potresti quindi essere tentato di scrivere:

```
for (i = 0; i < 100; ++i)
    assert (do_something () == 0);
```

Comunque, potresti scoprire che questo controllo a runtime impone al programma troppe penalità di prestazioni e decidere successivamente di compilare con `NDEBUG` definito. Ciò rimuo-

verà interamente la chiamata `assert`, così l'espressione non sarà mai valutata e `do_something` non verrà mai chiamata. Piuttosto potresti scrivere questo:

```
for (i = 0; i < 100; ++i) {
int status = do_something ();
assert (status == 0);
}
```

Un'altra cosa da tenere in mente è che non si dovrebbe usare assert per testare l'input non valido da parte dell'utente. Agli utenti non piace quando l'applicazione va in crash semplicemente con un messaggio di errore critico in riposta all'input non corretto. Dovresti sempre controllare l'input non corretto e produrre dei messaggi d'errore adeguati in risposta all'input. Utilizza `assert` solo per controlli interni a runtime.

Alcuni buoni punti per l'utilizzo di `assert` sono questi:

- controlla che non ci siano puntatori nulli, per esempio, un argomento di funzione non valido. Il messaggio di errore generato da `{assert (pointer != NULL) }`,

  ```
  Assertion 'pointer != ((void *)0)' failed.
  ```

  Fornisce molte più informazioni rispetto al messaggio di errore che ne risulterebbe se il tuo programma facesse riferimento ad un puntatore nullo:

  ```
  Segmentation fault (core dumped)
  ```

- Controlla le condizioni nei valori dei parametri delle funzioni. Per esempio, se una funzione dovrebbe essere chiamata solo con un valore positivo per il parametro foo, usa questo all'inizio del corpo della funzione:

  ```
  assert (foo > 0);
  ```

  Ciò ti aiuterà a trovare utilizzi scorretti della funzione, e rende anche molto chiaro a chi sta leggendo il codice sorgente della funzione che c'è una restrizione sul valore del parametro.

Non fermari solo a questo; usa `assert` liberamente nei tuoi programmi.

2.2.2 Fallimenti delle chiamate di sistema

Molti di noi hanno imparato originariamente come scrivere programmi che vanno dall'esecuzione fino al completamento lungo un percorso ben definito. Dividiamo il programma in compiti (task) e sotto-compiti (subtask), ed ogni funzione finisce un compito invocando altre funzioni per compiere i corrispondenti sotto-compiti. Dando gli input appropriati, ci aspettiamo che una funzione produca il corretto output e relativi effetti collaterali.

La realtà dell'hardware e del software dei computer si intromette in questo sogno idealizzato. I computer hanno risorse limitate; l'hardware sbaglia; molti programmi vengono eseguiti

allo stesso tempo; sia utenti che programmatori commettono degli errori. È spesso al confine tra applicazione e sistema operativo che si manifestano queste realtà. Quindi, quando si utilizzano le chiamate di sistema per accedere alle risorse di sistema, per eseguire operazioni di I/O, o per altri scopi, è importante capire non solo cosa accade quando le chiamate di sistema hanno successo, ma anche come e perché le chiamate possono fallire.

Le chiamate di sistema possono fallire in molti modi. Per esempio:

- Il sistema può girare al di fuori delle risorse (o il programma può eccedere i limiti delle risorse assegnate dal sistema per un singolo programma). Per esempio, il programma potrebbe tentare di allocare troppa memoria, per scrivere troppo sull'hard disk, o aprire troppi file allo stesso tempo.

- Linux può bloccare certe chiamate di sistema quando un programma tenta di eseguire un'operazione per la quale non ha i permessi. Per esempio, un programma può tentare di scrivere su un file marcato per la sola lettura, accedere alla memoria di un altro processo, o uccidere il programma di un altro utente.

- Gli argomenti per una chiamata di sistema potrebbero non essere validi, sia perché l'utente potrebbe aver fornito un input non valido o per un bug del programma. Per esempio, il programma potrebbe passare un indirizzo di memoria non valido o un descrittore di file non valido ad una chiamata di sistema. O un programma potrebbe tentare di aprire una direcotory come fosse un file ordinario, o potrebbe passare il nome di un file ordinario ad una chiamata di sistema che si aspetta una directory.

- Una chiamata di sistema può fallire per motivi esterni al programma. Ciò accade molto spesso quando una chiamata di sistema accede ad una periferica hardware. La periferica potrebbe essere difettosa o potrebbe non supportare una particolare operazione, o forse un disco non è inserito nel drive.

- Una chiamata di sistema può qualche volta essere interrotta da un evento esterno, come l'invio di un segnale. Questo può non indicare un totale fallimento, ma è la causa del fatto che il programma chiamante riavvii la chiamata di sistema, se lo si desidera.

In un programma ben scritto, che fa ampio uso delle chiamate di sistema, si verifica spesso il caso in cui la maggior parte del codice ha il compito di scovare e gestire gli errori ed altre circostanze critiche piuttosto che occuparsi del lavoro principale del programma.

2.2.3 Codici di errore dalle chiamate di sistema

La maggior parte delle chiamate di sistema restituiscono zero se l'operazione ha avuto successo o un valore diverso da zero se l'operazione è fallita. (Molti, comunque, usano diverse convenzioni per i valori di ritorno; per esempio, `malloc` restituisce un puntatore nullo per indicare un fallimento. Leggi sempre le pagine di manuale con attenzione quando usi una chiamata di sistema). Sebbene quest'informazione possa essere sufficiente per determinare quando il programma dovrebbe continuare l'esecuzione come di solito, probabilmente non fornisce abbastanza informazioni per un ripristino sensibile dagli errori.

Molte chiamate di sistema usano una variabile speciale chiamata `errno` per memorizzare ulteriori informazioni in caso di fallimento.[2] Quando una chiamta di sistema fallisce, il sistema setta la variabile `errno` ad un valore che indica cosa è andato storto. Poiché tutte le chiamate di sistema usano la stessa variabile `errno` per memorizzare le informazioni, dovresti copiare immediatamente il valore in un'altra variabile dopo il fallimento della chiamata di sistema. Il valore di `errno` verrà sovrascritto la prossima volta che farai una chiamata di sistema.

I valori di errore sono degli interi; valori possibili sono assegnati dalle macro di preprocessore, per convenzione con tutte lettere maiuscole ed inizianti con "E" – per esempio, `EACCES` e `EINVAL`. Usa sempre queste macro per fare riferimento ai valori di `errno` piuttosto che valori interi. Se usi i valori di `errno` includi l'header `<errno.h>`.

GNU/Linux fornisce una conveniente funzione, `strerror`, che restituisce una descrizione in stringa di caratteri di un codice di errore `errno`, adatto per l'utilizzo nei messaggi di errore. Se usi `strerror` includi `<string.h>`.

GNU/Linux fornisce anche `perror`, che stampa la descrizione dell'errore direttamente sullo stream `stderr`. Passa a perror un prefisso come stringa di caratteri da stampare prima della descrizione dell'errore, che solitamente dovrebbe includere il nome della funzione che è fallita. Se usi `perror` includi `<stdio.h>`.

Questo frammento di codice cerca di aprire un file; se l'apertura fallisce, esso stampa un messaggio di errore ed esce dal programma. Nota che la chiamata `open` restituisce un descrittore di file aperto se l'operazione di apertura ha avuto successo, o `-1` se l'operazione fallisce.

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
/* The open failed. Print an error message and exit. */
fprintf (stderr, ?error opening file: %s\n?, strerror (errno));
exit (1);
}
```

Dipendentemente dal tuo programma e dalla natura della chiamata di sistema, l'azione appropriata in caso di fallimento può essere quella di stampare un messaggio di errore, di cancellare l'operazione, di far chiudere il programma, di ritentare oppure di ignorare l'errore. È importante comunque inserire in un modo o nell'altro una logica che gestisca tutte le possibili modalità di fallimento.

Un possibile codice di errore al quale dovresti prestare attenzione, specialmente con le funzioni di I/O, è `EINTR`. Alcune funzioni, come `read`, `select` e `sleep`, possono impiegare molto tempo per essere eseguite. Queste sono considerate funzioni *bloccanti* perché l'esecuzione del programma è bloccata fino a che la chiamata non è completata. Comunque, se il programma riceve un segnale mentre è bloccato in una di queste chiamate, la chiamata finirà senza aver completato l'operazione. In questo caso, `errno` è settato a `EINTR`. Di solito, in questi casi, vorrai ritentare la chiamata di sistema.

Ecco un frammento di codice che usa la chiamata `chown` per cambiare il proprietario di un file dato da `path` all'utente da `user_id`. Se la chiamata fallisce, il programma esegue l'azione in

[2] Attualmente, per ragioni di sicurezza dei thread, `errno` è implementata come macro, ma usata come una variabile globale.

base al valore di `errno`. Nota che quando troviamo un possibile bug nel programma, usciamo usando `abort` o `assert`, che causa la generazione di un file *core*. Ciò può esser utile per un debug post-mortem. Per altri errori non recuperabili, come la condizione di out-of-memory, usciamo usando `exit` ed un valore diverso da zero perché un file core non sarebbe molto utile.

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* Save errno because it's clobbered
     by the next system call. */
  int error_code = errno;
  /* The operation didn?t succeed; chown
     should return -1 on error. */
  assert (rval == -1);
  /* Check the value of errno, and take appropriate action. */
  switch (error_code) {
  case EPERM: /* Permission denied. */
  case EROFS: /* PATH is on a read-only file system. */
  case ENAMETOOLONG: /* PATH is too long. */
  case ENOENT: /* PATH does not exit. */
  case ENOTDIR: /* A component of PATH is not a directory. */
  case EACCES: /* A component of PATH is not accessible. */
      /* Something?s wrong with the file.
       Print an error message. */
    fprintf (stderr, "error changing ownership of %s: %s\n",
        path, strerror (error_code));
    /* Don?t end the program; perhaps give the user a chance to
      choose another file... */
    break;
  case EFAULT:
    /* PATH contains an invalid memory address.
       This is probably a bug. */
  abort ();
  case ENOMEM:
    /* Ran out of kernel memory. */
       fprintf (stderr, ?%s\n?, strerror (error_code));
    exit (1);
  default:
    /* Some other, unexpected, error code.
       We?ve tried to handle all possible error codes;
       if we've missed one, that?s a bug! */
    abort ();
  };
}
```

Potresti semplicemente aver usato questo codice, che si comporta allo stesso modo se la chiamata ha successo:

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

Ma se la chiamata fallisce, quest'alternativa non fornisce nessun aiuto per il riportare, gestire, o riprendersi dagli errori.

Se usi la prima forma, la seconda forma o qualche altra cosa intermedia dipende dai requisiti di trovare gli errori e gestirli del tuo programma.

2.2.4 Errori ed allocazione delle risorse

Spesso, quando una chiamata di sistema fallisce, sarebbe opportuno annullare l'operazione corrente ma non terminare il programma poiché potrebbe essere possibile un ripristino dall'errore. Un modo per farlo è quello di ritornare dalla funzione corrente, restituendo un codice alla chiamante che indichi l'errore.

Se decidi di ritornare mentre l'esecuzione è nel mezzo della funzione, è importante assicurarsi che ogni risorsa nella funzione precedentemente allocata con successo sia prima deallocata. Le risorse possono includere memoria, descrittori di file, puntatori a file, file temporanei, sincronizzazione di oggetti, e così via. Altrimenti, se il tuo programma continua a girare, le risorse allocate prima dell'errore verrebbero perse.

Considera, per esempio, una funzione che legge da un file in un buffer. La funzione potrebbe seguire questi passi:

1. Allocare il buffer.
2. Aprire il file.
3. Leggere dal file nel buffer.
4. Chiudere il file.
5. Restituire il buffer.

Se il file non esiste, il passo 2 fallirà. Un'azione appropriata potrebbe essere quella di restituire NULL dalla funzione. Comunque, se il buffer è stato già allocato al passo 1, c'è il rischio di perdere quella memoria. Devi ricordarti di deallocare il buffer da qualche parte lungo ogni flusso di controllo dal quale non ritorni. Se il passo 3 fallise, non devi solo deallocare il buffer prima di ritornare, ma devi anche chiudere il file.

Il listato 2.6 mostra un esempio di come dovresti scrivere questa funzione.

Listato 2.6: (***readfile.c***) – **Liberare le risorse in condizioni non normali**

```
1 #include <fcntl.h>
2 #include <stdlib.h>
3 #include <sys/stat.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 char* read_from_file (const char* filename, size_t length)
8 {
9   char* buffer;
```

```
10    int fd;
11    ssize_t bytes_read;
12
13    /* Allocate the buffer.  */
14    buffer = (char*) malloc (length);
15    if (buffer == NULL)
16      return NULL;
17    /* Open the file.  */
18    fd = open (filename, O_RDONLY);
19    if (fd == -1) {
20      /* open failed.  Deallocate buffer before returning.  */
21      free (buffer);
22      return NULL;
23    }
24    /* Read the data.  */
25    bytes_read = read (fd, buffer, length);
26    if (bytes_read != length) {
27      /* read failed.  Deallocate buffer and close fd before returning.  */
28      free (buffer);
29      close (fd);
30      return NULL;
31    }
32    /* Everything's fine.  Close the file and return the buffer.  */
33    close (fd);
34    return buffer;
35  }
```

Linux libera la memoria allocata, file aperti, e molte altre risorse quando un programma termina, così non è necessario dellocare buffer e chiudere file prima di chiamare `exit`. Potresti aver bisogno di liberare manualmente altre risorse condivise, comunque, come file temporanei e memoria condivisa, che potrebbero potenzialmente persistere nel programma.

2.3 Scrivere ed usare librerie

Virtualmente tutti i programmi sono linkati con una o più librerie. Ogni programma che usa una funzione C (come `printf` o `malloc`) sarà linkato con la libreria di runtime C. Se il tuo programma ha un'interfaccia utente grafica (GUI), esso sarà linkato con le librerie delle finestre. Se il tuo programma usa un database, il fornitore del database ti darà le librerie che potrai usare per accedere al database.

In ognuno di questi casi, devi decidere come linkare la libreria, statica o dinamica. Se scegli di fare il link statico, i tuoi programmi saranno grandi e difficili da aggiornare, ma probabilmente facili da sviluppare. Se fai il link dinamico, i tuoi programmi saranno piccoli, facili da aggiornare, ma difficili da sviluppare. Questa sezione spiega come fare i link, sia statici che dinamici, esamina le differenze più dettagliatamente, e fornisce alcune "regole a naso" per decidere che tipo di link è meglio per te.

2.3.1 Archivi

Un *archivio* (o libreria statica) è semplicemente una collezione di file oggetto memorizzati in un singolo file. (Un archivio approssimativamente è l'equivalente di un file `.LIB` di Windows). Quando fornisci un archivio al linker, il linker cerca nell'archivio i file oggetto che gli servono, li estrae, e quindi ne fa il link nel tuo programma come se tu gli avessi fornito quei file direttamente.

Puoi creare un archivio usando il comando `ar`. I file di archivio tradizionalmente usano l'estensione .a piuttosto che l'estensione .o usata comunemente dai file oggetto. Ecco come puoi combinare `test1.o` e `test2.o` in un singolo archivio `libtest.a`:

```
% ar cr libtest.a test1.o test2.o
```

Il flag `cr` dice ad `ar` di creare un archivio.[3] Adesso puoi fare il link con questo archivio usando l'opzione `-ltest` con gcc o g++, come descritto nella sottosezione 1.2.2, "Fare il link di file oggetto", nel Capitolo 1, "Programmazione UNIX avanzata con Linux".

Quando il linker incontra un archivio sulla riga di comando, cerca nell'archivio tutte le definizioni di simboli (funzioni o variabili) che sono referenziate dai file oggetto, che sono già stati processati ma non ancora definiti. I file oggetto che definiscono questi simboli sono estratti dall'archivio ed inclusi nell'eseguibile finale. Poiché il linker cerca l'archivio solo quando lo incontra sulla riga di comando, di solito ha senso mettere l'archivio alla fine della riga di comando. Per esempio, supponiamo che `test.c` contenta il codice nel listato 2.7 e `app.c` contenga il codice nel listato 2.8.

Listato 2.7: (*test.c*) – **Contenuto della libreria**

```
1 int f ()
2 {
3   return 3;
4 }
```

Listato 2.8: (*app.c*) – **Un programma che usa le funzioni della libreria**

```
1 extern int f ();
2
3 int main ()
4 {
5   return f ();
6 }
```

Adesso supponi che `test.o` sia combinato con alcuni altri file oggetto per creare l'archivio libtest.a. Il seguente comando non funzionerà:

```
% gcc -o app -L. -ltest app.o
app.o: In function 'main':
```

[3] È possibile usare altri flag per rimuovere file da una archivio o per eseguire altre operazione nell'archivio. Queste operazioni sono usate raramente ma sono documentate nella pagina di manuale di `ar`.

```
app.o(.text+0x4): undefined reference to 'f'
collect2: ld returned 1 exit status
```

Il messaggio di errore indica che anche se `libtest.a` contiene una definizione di `f`, il linker non la trova. Ciò accade perché `libtest.a` è stato cercato quando è stato incontrato all'inizio, ed in quel punto il linker non ha visto alcun riferimento a `f`.

D'altro canto, se usiamo questa linea, non avremo nessun errore:

```
%gcc -o app app.o -L. -ltest
```

Il motivo è che il riferimento ad `f` in app.o causa che il linker includa il file oggetto `test.o` dall'archivio `libtest.a`.

2.3.2 Librerie condivise

Una libreria condivisa (conosciuta anche come oggetto condiviso o libreria linkata dinamicamente) è simile a un archivio nel quale c'è un gruppo di file oggetto. Comunque, ci sono molte differenze importanti. La differenza principale è che quando una libreria condivisa è linkata in un programma, l'eseguibile finale non contiene il codice che è presente nella libreria condivisa. L'eseguibile invece contiene un riferimento alla libreria condivisa. Se nel sistema molti programmi sono linkati alla stessa libreria condivisa, essi faranno tutti riferimento alla stessa libreria, ma nessuno sarà incluso. Così, la libreria è "condivisa" da tutti i programmi che sono linkati ad essa.

Una seconda importante differenza è che la libreria condivisa non è semplicemente una collezione di file oggetto nel quale il linker sceglie quelli che servono a soddisfare i riferimenti indefiniti. Piuttosto, i file oggetto che compongono la libreria condivisa sono combinati in un singolo file oggetto cosicché un programma che linka assieme una libreria condivisa include sempre tutto il codice della libreria piuttosto che solo quelle porzioni di cui ha bisogno.

Per creare una libreria condivisa, devi compilare il file oggetto che formerà la libreria usando l'opzione `-fPIC` per il compilatore, come questa:

```
% gcc -c -fPIC test1.c
```

L'opzione `-fPIC` dice al compilatore che stai per usare `test.o` come parte di un oggetto condiviso.

> **Position-Independent Code (PIC)**
> PIC sta per codice a posizione indipendente (position-independen code). Le funzioni in una libreria condivisa possono essere caricate ad indirizzi diversi in diversi programmi, così il codice nell'oggetto condiviso non deve dipendere dall'indirizzo (o posizione) al quale è stato caricato. Questa considerazione non ha nessun impatto su di te, come programmatore, eccetto che devi ricordare di usare il flag -fPIC quando compili del codice che userai in una libreria condivisa.

Quindi combini il file oggetto in una libreria condivisa, come questa:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

L'opzione `-shared` dice al linker di prourre una libreria piuttosto che un eseguibile come di solito. Le librerie condivise usano l'estensione `.so`, che sta per shared object (oggetto condiviso). Come gli archivi statici, il nome inizia sempre con `lib` per indicare che il file è una libreria.

Fare il link con una libreria condivisa è come fare il link con un archivio statico. Per esempio, la seguente riga farà il link con libtest.so se esso è nella directory corrente o in una delle directory nei percorsi di ricerca standard del sistema:

```
% gcc -o app app.o -L. -ltest
```

Supponiamo che siano disponibili entrambi `libtest.a` e `libtest.so`. A questo punto il linker dovrà scegliere una delle libreria e scartare l'altra. Il linker cerca in ogni directory (prima quelle specificate dall'opzione `-L`, e poi quelle nelle directory standard). Quando il linker trova una directory che contiene o `libtest.a` o `libtest.so`, smette la ricerca nelle directory. Se solo una delle due forme è presente nella directory, il linker sceglierà quella forma. Altrimenti, il linker sceglierà la versione in libreria condivisa, a meno che tu non gli dica di fare altrimenti. Puoi usare l'opzione `-static` per far utilizzare gli archivi statici. Per esempio, la seguente linea userà l'archivio `libtest.a`, anche se la libreria `libtest.so` è pure disponibile.

```
% gcc -static -o app app.o -L. -ltest
```

Il comando `ldd` mostra le librerie condivise che sono linkate nell'eseguibile. È necessario che queste librerie siano disponibili quando l'eseguibile è in esecuzione. Nota che `ldd` elencherà una libreria aggiuntiva chiamata `ld-linux.so`, che è parte del meccanismo di linking dinamico di GNU/Linux.

Usare *LD_LIBRARY_PATH* Quando fai il link di un programma con una libreria condivisa, il linker non mette il percorso completo per la libreria condivisa nell'eseguibile finale. Piuttosto, mette solo il nome della libreria condivisa. Quando il programma è in esecuzione, il sistema cerca la libreria condivisa e la carica. Il sistema, di default, cerca solo in `/lib` e `/usr/lib`. Se la libreria condivisa linkata al tuo programma è installata fuori da queste directory, non verrà trovata, ed il sistema si rifiuterà di eseguire il programma.

Una soluzione a questo problema sta nell'utilizzo dell'opzione `-Wl, -rpath` quando si fa il link del programa. Supponi di usare la seguente:

```
% gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

In questo modo, quando il programma è in esecuzione, il sistema cercherà in `/usr/local/lib` per ogni libreria condivisa richiesta.

Un'altra soluzione al problema è impostare la variabile d'ambiente `LD_LIBRARY_PATH` quando si esegue il programma. Come la variabile d'ambiente `PATH`, `LD_LIBRARY_PATH` è una lista di directory separate dai due punti. Per esempio, se `LD_LIBRARY_PATH` è `/usr/local/lib:/opt/lib`, allora la ricerca verrà fatta in `/usr/local/lib` e in `/opt/lib` prima delle directory standard `/lib` e `/usr/lib`. Dovresti notare anche che se hai

`LD_LIBRARY_PATH`, il linker nel creare l'eseguibile cercherà nelle directory fornite qui oltre che nelle directory fornite dall'opzione `-L`.[4]

2.3.3 Librerie standard

Anche se non non hai specificato nessuna libreria quando hai fatto il link del tuo programma, quasi certamente questo userà una libreria condivisa. Ciò accade perché GCC fa il link automaticamente nelle librerie standard C, `libc`, per te. Le funzioni matematiche delle librerie standard C non sono incluse in `libc`; invece, stanno in una libreria separata, `libm`, che hai bisogno di specificare esplicitamente. Per esempio, per compilare e fare il link di un programma `compute.c` che usa funzioni trigonometriche come `sin` e `cos`, devi richiamare questo codice:

```
% gcc -o compute compute.c -lm
```

Se scrivi un programma C++ e fai il link usando il comando `c++` o `g++`, prenderai automaticamente anche la libreria standard C++, `libstdc++`.

2.3.4 Dipendenze delle librerie

Una libreria spesso dipenderà da un'altra libreria. Per esempio, molti sistemi GNU/Linux includono `libtiff`, una libreria contenente funzioni per leggere e scrivere file di immagini in formato TIFF. Questa libreria, a sua volta, usa le librerie `libjpeg` (procedure per immagini JPEG) e `libz` (procedure di compressione)

Il listato 2.9 mostra un programma molto piccolo che usa libtiff per aprire un file di immagine TIFF.

Listato 2.9: (*tifftest.c*) – Utilizzo di *libtiff*

```
1  #include <stdio.h>
2  #include <tiffio.h>
3
4  int main (int argc, char** argv)
5  {
6    TIFF* tiff;
7    tiff = TIFFOpen (argv[1], "r");
8    TIFFClose (tiff);
9    return 0;
10 }
```

Salva questo file sorgente come `tifftest.c` Per compilare questo programma e fare il link con `libtiff`, specifica `-ltiff` sulla tua riga di comando del linker:

```
% gcc -o tifftest tifftest.c -ltiff
```

[4] Potresti trovare dei riferimenti a `LD_RUN_PATH` in alcune documentazioni online. Non credere a quello che leggi; questa variabile non fa nulla sotto GNU/Linux.

Di default, questo si collegherà alla versione della libreria condivisa di `libtiff`, trovata in `/usr/lib/libtiff.so`. Poiché `libtiff` usa `libjpeg` e `libz`, le versioni delle librerie condivise di queste due sono anche qui (una libreria condivisa può puntare ad altre librerie condivise dalle quali dipende). Per verificare ciò, usa il comando `ldd`:

```
% ldd tifftest
    libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
    libc.so.6 => /lib/libc.so.6 (0x40060000)
    libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
    libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Le librerie statiche, d'altro canto, non possono puntare ad altre librerie. Se decidi di fare il link con la versione statica di `libtiff` specificando `-static` sulla riga di comando, otterrai dei simboli non risolti:

```
% gcc -static -o tifftest tifftest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function
        'TIFFjpeg_error_exit':
tif_jpeg.o(.text+0x2a): undefined reference to 'jpeg_abort'
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function
        'TIFFjpeg_create_compress':
tif_jpeg.o(.text+0x8d): undefined reference to 'jpeg_std_error'
tif_jpeg.o(.text+0xcf): undefined reference to 'jpeg_CreateCompress'
...
```

Per fare il link di questo programma devi specificare le altre due librerie tu stesso:

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Occasionalmente, due librerie saranno mutuamente dipendenti. In altre parole, il primo archivio farà riferimento a simboli definiti nel secondo archivio e vice versa. La situazione generalmente mostra la scarsità di design, ma lo fa occasionalmente. In questo caso, nella riga di comando puoi indicare più volte una singola libreria. Il linker cercherà la libreria ogni volta che ne ha bisogno. Per esempio, questa linea causa la ricerca in `libfoo.a` più volte:

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

Così, anche se libfoo.a fa riferimento a simboli in `libbar.a` e vice versa, il link del programma verrà fatto con successo.

2.3.5 Pro e contro

Adesso che sai tutto riguardo gli archivi statici e le librerie condivise, probabilmente ti chiederai come usarle. Ci sono poche altre importanti considerazioni da tenere a mente.

Il principale vantaggio di una libreria condivisa è quello di risparmiare spazio sul sistema nel quale il programma è installato. Se stai installando 10 programmi, ed essi fanno tutti uso della stessa libreria condivisa, risparmi molto spazio usando una libreria condivisa. Se invece

hai utilizzato un archivio statico, l'archivio è incluso in tutti e 10 i programmi. Così, usando una libreria condivisa risparmi spazio su disco. Ciò riduce anche il tempo di download se il tuo programma viene scaricato dal web.

Un vantaggio correlato alle librerie condivise è che gli utenti possono aggiornare le librerie senza aggiornare tutti i programmi che dipendono da esse. Per esempio, supponi che hai creato una libreria condivisa che gestisce le connessioni HTTP. Molti programmi potrebbero dipendere da questa libreria. Se trovi un bug in questa libreria puoi aggiornare la libreria. Istantaneamente tutti i programmi che dipendono da questa libreria saranno riparati; non dovrai rifare il link di tutti i programmi come faresti per un archivio statico.

Questi vantaggi potrebbero farti pensare che dovresti sempre usare le librerie condivise. Comunque, sostanzialmente esistono delle ragioni per le quali usare invece gli archivi statici. Il fatto che un aggiornamento di una libreria condivisa ha effetto su tutti i programmi che dipendono da essa può essere uno svantaggio. Per esempio, se stai sviluppando un software mission-critical, farai piuttosto il link con un archivio statico in modo che un aggiornamento delle librerie condivise nel sistema non produca effetti sul tuo programma. (Altrimenti, gli utenti potrebbero aggiornare la libreria condivisa, danneggiando con ciò il tuo programma e quindi chiamare l'assistenza clienti, attribuendoti le colpe!).

Se non riesci ad installare le tue librerie in `/lib` o `/usr/lib`, dovresti pensarci due volte prima di usare una libreria condivisa. (Non sarai in grado di installare le librerie in quelle directory se ti aspetti che l'utente installi il tuo software senza permessi di amministrazione). In particolare, il trucco `-Wl, -rpath` non funzionerà se non sai dove andranno a finire le librerie. E chiedere al tuo utente di settare `LD_LIBRARY_PATH` significa un passo in più per loro. Poiché ogni utente deve farlo individualmente, ciò è sostanzialmente un peso.

Devi pesare questi vantaggi e svantaggi per ogni programma che distribuisci.

2.3.6 Loading e Unloading dinamico

A volte vuoi caricare del codice a tempo di esecuzione senza fare il link esplicito in quel codice. Per esempio, considera un'applicazione che supporti moduli "plug-in" come un browser web. Il browser permette a terze parti di creare plug-in per fornire funzionalità aggiuntive. Gli sviluppatori di terze parti creano librerie condivise e le mettono in posizioni conosciute. Il browser web quindi carica automaticamente il codice di queste librerie.

Questa funzionalità è disponibile su Linux usando la funzione `dlopen`. Puoi aprire una libreria condivisa chiamata `libtest.so` chiamando `dlopen` come qui:

```
dlopen ('libtest.so', RTLD_LAZY)
```

(Il secondo parametro è un flag che indica come collegare i simboli nella libreria condivisa. Se vuoi altre informazioni puoi consultare le pagine di manuale online di `dlopen`, ma `RTLD_LAZY` è l'impostazione che ti serve di solito). Per usare le funzioni di caricamento dinamico, includi il file header `<dlfcn.h>` e fai il link con l'opzione `-ldl` per avere il collegamento alla libreria libdl.

Il valore di ritorno di questa funzione è un `void *` che è usato come aggancio per la libreria condivisa. Puoi passare questo valore alla funzione `dlsym` per ottenere l'indirizzo

di una funzione che è stata caricata con la libreria condivisa. Per esempio, se `libtest.so` definisce una funzione chiamata `my_function`, puoi chiamarla come in questo modo:

```
void* handle = dlopen ('libtest.so', RTLD_LAZY);
void (*test)() = dlsym (handle, 'my_function');
(*test)();
dlclose (handle);
```

La chiamata di sistema `dlsym` può anche essere usata per ottenere un puntatore ad una variabile statica nella libreria condivisa.

Enrambi `dlopen` e `dlsym` restituiscono `NULL` se non hanno successo. In questo caso, puoi chiamare `dlerror` (senza parametri) per ottenere un messaggio di errore comprensibile con la descrizione del problema.

La funzione `dlcole` chiude la libreria condivisa. Tecnicamente, `dlopen` attualmente carica la libreria solo se non è già stata caricata. Se la libreria è già stata caricata, `dlopen` incrementa semplicemente il contatore di riferimento alla libreria. Similmente, `dlclose` decrementa il contatore di riferimento e chiude la libreria solo se il contatore di riferimento ha raggiunto zero.

Se stai scrivendo il codice nella tua libreria condivisa in C++, probabilmente vorrai dichiarare quelle funzioni e variabili alle quali vorrai che si acceda altrove con lo specificatore di link `extern C`. Per esempio, se la funzione C++ `my_function` è in una libreria condivisa e vuoi accedere ad essa con `dlsym`, la dovresti dichiarare come questa:

```
extern "C" void foo ();
```

Ciò evita che il compilatore C++ "distrugga" il nome della funzione, cambiando il nome da `foo` ad uno differente, per trasformarlo in uno diverso che rappresenti codificate informazioni extra riguardo la funzione. Un compilatore C non distruggerà i nomi; userà qualunque nome che darai alla tua funzione o variabile.

Capitolo 3

Processi

Un'istanza di un programma in esecuzione è chiamata processo. Se hai due finestre di terminale visualizzate a schermo, con buone probabilità stai eseguendo due volte lo stesso programma terminale – hai due processi di terminale. Ogni finestra di terminale è probabilmente una shell in esecuzione; ogni shell in esecuzione è un altro processo. Quando invochi un comando da una shell, il programma corrispondente è eseguito in un nuovo processo; il processo schell si riprende quando i processi sono finiti.

I programmatori avanzati spesso usano più processi cooperanti in una singola applicazione per abilitare l'applicazione a fare più di una cosa per volta, per aumentare la robustezza dell'applicazione e per fare uso di programmi già esistenti.

Molte delle funzioni di manipolazione dei processi che sono descritte in questo capitolo sono simili a quelle in altri sistemi UNIX. Molti sono dichiarati nel file header `<unistd.h>`; controlla la pagina di manuale per ogni funzione per esserne sicuro.

3.1 Uno sguardo ai processi

Anche se sei seduto davanti al tuo computer, ci sono processi in esecuzione. Ogni programma in esecuzione usa uno o più processi. Iniziamo dando un'occhiata ai processi già in esecuzione sul tuo computer.

3.1.1 ID dei processi

Ogni processo in un sistema Linux è identificato dal suo ID di processo univoco, al queale spesso ci si riferisce come *pid*. Gli ID dei processi sono numeri a 16 bit assegnati sequenzialmente da Linux quando viene creato un nuovo processo.

Ogni processo inoltre ha un processo padre (ad eccezione del processo speciale `init`, descritto nella sottosezione 3.4.3, "Processi zombie"). Quini, puoi immaginare che i processi in un sistema Linux siano impostati in un albero, con il processo `init` come radice. Il *parent process ID*, o ppid, è semplicemente l'ID del padre del processo.

Quando si fa riferimento agli ID dei processi in un programma C o C++, si usa sempre il tipo di dato `pid_t`, che è definito in `<sys/types.h>`. Un programma può ottenere l'ID di

un processo in esecuzione con la chiamata di sistema `getpid()` e ottenere l'ID del padre di questo processo con la chiamata di sistema `getppid()`. Per esempio, il programma nel listato 3.1 stampa gli ID dei suoi processi e gli ID dei loro genitori.

Listato 3.1: (*print-pid.c*) – **Stampa gli ID dei processi**

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main ()
5 {
6   printf ("The process id is %d\n", (int) getpid ());
7   printf ("The parent process id is %d\n", (int) getppid ());
8   return 0;
9 }
```

Nota che se invochi questo programma molte volte, viene riportato un ID di processo diverso perché ogni invocazione crea un nuovo processo. Comunque, se lo invochi ogni volta dalla stessa shell, l'ID del processo padre (che è l'ID di processo del processo shell) è lo stesso.

3.1.2 Vedere i processi attivi

Il comando `ps` mostra i processi che stanno girando nel sistema. La versione GNU/Linux di `ps` ha molte opzioni perché cerca di essere compatibile con le versioni di `ps` in molte altre varianti di UNIX. Queste opzioni controllano quali processi sono listati e le informazioni a riguardo per ogni visualizzazione.

Per default, l'invocazione di `ps` mostra i processi mostrati dal terminale o dalla finestra di terminale nella quale `ps` è invocato. Per esempio:

```
% ps
PID    TTY      TIME     CMD
21693  pts/8    00:00:00 bash
21694  pts/8    00:00:00 ps
```

L'invocazione di `ps` mostra due processi. Il primo, `bash`, è la shell in esecuzione in questo terminale. Il secondo è l'istanza in esecuzione del programma `ps` stesso. La prima colonna, etichettata PID, mostra l'ID di processo di ognuno.

Per una vista più dettagliata e ciò che sta girando sul sistema GNU/Linux, invoca questo:

```
% ps -e -o pid,ppid,command
```

L'opzione `-e` dice a `ps` di mostrare tutti i processi che stanno girando sul sistema. L'opzione `-o pid,ppid,command` indica a `ps` quali informazioni mostrare a riguardo di ogni processo – in questo caso, l'ID di processo, l'ID del processo padre, ed il comando in esecuzione in questo processo.

Formati dell'output di ps

Con l'opzione `-o` al comando `ps`, specifichi le informazioni riguardanti i

processi che vuoi nell'output come lista separata da virgole. Per esempio,
`ps -o pid,user,start_time,command`
mostra l'ID di processo, il nome del proprietario del processo, l'orario al quale il processo è stato avviato ed il comando eseguito in quel processo. Vedi la pagina di manuale di `ps` per la lista completa di codici di campo. Puoi usare invece le opzioni -f (lista completa), -l (lista lunga) o -j (lista dei lavori) per ottenere tre diversi formati di liste preimpostati.

Ecco le prime righe ed ultime righe di output da questo comando nel mio sistema. Puoi notare output differenti, in base a cosa c'è in esecuzione nel tuo sistema.

```
% ps -e -o pid,ppid,command
PID    PPID   COMMAND
1      0      init [5]
2      1      [kflushd]
3      1      [kupdate]
...
21725  21693  xterm
21727  21725  bash
21728  21727  ps -e -o pid,ppid,command
```

Nota che l'ID del processo padre del comando ps, 21727, è l'ID di processo di bash, la shell dalla quale ho invocato `ps`. L'ID del processo padre di bash a sua volta è 21725, l'ID di processo del programma xterm nel quale la shell sta girando.

3.1.3 Uccidere un processo

Puoi uccidere un processo in esecuzione con il comando `kill`. Specifica semplicemente sulla linea di comando l'ID di processo del processo da uccidere.

Il comando `kill` lavora inviando al processo un `SIGTERM`, o segnale di termine.[1] Ciò fa terminare il processo, a meno che il programma in esecuzione non prenda esplicitamente o mascheri il senale SIGTERM. I segnali sono descritti nella sezione 3.3, "Segnali".

3.2 Creare processi

Vengono usate due tecniche comuni per creare un nuovo processo. La prima è relativamente semplice ma dovrebbe essere usata con moderazione perché non è efficiente ed ha considerevoli rischi di sicurezza. La seconda tecnica è più complessa ma fornisce maggiore flessibilità, velocità e sicurezza.

3.2.1 usando system

La funzione `system` nella libreria standard C fornisce una via facile per eseguire un comando dall'interno di un programma, come se il comando sia stato digitato in una shell. Di fatto,

[1] puoi anche usare il comando `kill` per inviare altri segnali a un processo. Questo è descritto nella sezione 3.4, "Terminazione dei processi".

`system` crea un sottoprocesso eseguendo la Boune shell standard (/bin/sh) e cede il comando a quella shell per l'esecuzione. Per esempio, questo programma nel listato 3.2 invoca il comando `ls` per mostrare il contenuto della directory root come se avessi digitato `ls -l /` in una shell.

Listato 3.2: (*system.c*) – **Utilizzo della chiamata *system***

```
1 #include <stdlib.h>
2
3 int main ()
4 {
5   int return_value;
6   return_value = system ("ls -l /");
7   return return_value;
8 }
```

La funzione `system` restituisce lo stato di uscita del comando shell. Se la shell stessa non può essere eseguita, `system` restituisce 127; se si verifica un altro errore, `system` restituisce `-1`.

Poiché la funzione `system` usa una shell per invocare il tuo comando, esso è soggetto alle caratteristiche, limitazioni, e problemi di sicurezza della shell di sistema. Non puoi fare riferimento all'abilità di ogni particolare versione della shell Bourne. Su molti sistemi UNIX, `/bin/sh` è un link simbolico ad un'altra shell. Per esempio, su molti sistemi GNU/Linux, `/bin/sh` punta alla `bash` (Bourne-Again Shell), e diverse distribuzioni GNU/Linux usano diverse versioni di bash. Invocare un programma con permessi di `root` con la funzione `system`, per esempio, può avere diversi risultati in diversi sistemi GNU/Linux. Quindi, è preferibile usare il metodo `fork` ed `exex` per creare processi.

3.2.2 Usando *fork* ed *exec*

Le API di DOS e Windows contengono la famiglia di funzioni `span`. Queste funzioni prendono come argomento il nome di un programma da eseguire e creano una nuova istanza di processo di quel programma. Linux non contiene una singola funzione che fa tutto ciò in un solo passo. Piuttosto, Linux fornisce una funzione, `fork`, che crea un processo figlio che è una copia esatta del suo processo padre. Linux fornisce un altro insieme di funzioni, la famiglia `exec`, che fanno in modo che un particolare processo smetta di essere un'istanza di un programma e diventare invece una copia del processo corrente. Quindi puoi usare `exec` per trasformare uno di questi processi in un'istanza del programma che vuoi avviare.

Chiamare *fork* Quando un programma chiama `fork`, viene creato un processo duplicato, chiamato processo figlio. Il processo padre continua l'esecuzione del programma dal punto il cui è stato chiamato il `fork`. Il processo figlio, inoltre, esegue lo stesso programma dallo stesso posto.

Così, in cosa differiscono i due processi? Primo, il processo figlio è un nuovo processo e quindi ha un nuovo ID di processo, diverso dall'ID del suo processo padre. Un modo per

un programma per distinguere quando è nel processo padre o nel processo figlio è chiamare `getpid`. Comunque, la funzione `fork` fornisce diversi valori di ritorno ai processi padre e figlio – un processo "va dentro" la chiamata `fork` e due processi "vengono fuori", con differenti valori di ritorno. Il valore di ritorno nel processo padre è l'ID di processo del figlio. Il valore di ritorno nel processo figlio è zero. Poiché nessun processo ha un ID di processo pari a zero, ciò rende facile per il programma che sta girando sapere se è nel processo padre o figlio.

Il listato 3.3 è un esempio di utilizzo di `fork` per duplicare il processo di un programma. Nota che il primo blocco della condizione if è eseguito solo nel processo padre, mentre la clausola else è eseguita nel processo figlio.

Listato 3.3: **(*fork.c*) – Usare *fork* per duplicare il processo di un programma**

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process id is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process id is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());

  return 0;
}
```

Usare la famiglia *exec* Le funzioni `exec` sostituiscono il programma in esecuzione in un processo con un altro programma. Quando un programma chiama la funzione `exec`, quel processo cessa immediatamente l'esecuzione di quel programma ed inizia ad eseguire un nuovo programma dall'inizio, assumendo che la chiamata `exec` non incontri nessun errore.

All'interno della famiglia `exec` ci sono funzioni che variano leggermente nelle loro capacità e su come sono chiamate.

- Funzioni che contengono la lettera *p* nel loro nome (`execvp` ed `execpl`) accettano il nome di un programma e cercano un programma con quel nome nella directory della corrente esecuzione del programma; Alle funzioni che non contengono la p bisogna dare il percorso completo del programma che deve essere eseguito.

- Funzioni che contengono la lettera *v* nel loro nome (`execv`, `execvp` ed `execve`) accettano la lista di argomenti del nuovo programma come un array terminato da NULL di puntatori a stringhe. Le funzioni che contengono la letara *l* (`execl`, `execlp` ed `execle`) accettano la lista di argomenti usando il meccanismo varargs del linguaggio C.

- Funzioni che contengono la lettera *e* nel loro nome (`execve` ed `execle`) accettano un ulteriore argomento, un array di variabili di ambiente. L'argomento dovrebbe essere un array terminato da NULL di puntatori a stringhe di caratteri. Ogni stringa di caratteri dovrebbe essere nella forma "VARIABILE=valore".

Poiché `exec` sostituisce il programma chiamante con un altro, esso non ritorna mai finché non viene incontrato un errore.

La lista degli argomenti passata al programma è analoga agli argomenti da riga di comando che dai ad un programma quando lo esegui dalla shell. Essi sono disponibili tramite i parametri del main `argc` ed `argv`. Ricorda che quando un programma viene invocato dalla shell, la shell setta il primo elemento della lista degli argomenti (`argv[1]`) al primo argomento della riga di comando e così via. Quando usi una funzione `exec` nel tuo programma, anche tu, dovresti passare il nome della funzione come primo elemento della lista degli argomenti.

Usare *fork* ed *exec* insieme Un comune modello per eseguire un sottoprogramma all'interno di un programma è prima fare il fork del processo e quindi exec del sottoprogramma. Ciò permette al programma chiamante di continuare la propria esecuzione nel processo padre mentre il programma chiamante è sostituito dal sottoprogramma nel processo figlio.

Il programma nel listato 3.4, come il listato 3.2, lista il contenuto della directory radice usando il comando `ls`. Diversamente dall'esempio precedente, comunque, esso invoca il comando ls direttamente, passandogli gli argomenti della riga di comando `-l` e `/` piuttosto che invocarlo tramite una shell.

Listato 3.4: (***fork-exec.c***) – **Usare *fork* ed *exec* insieme**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  /* Spawn a child process running a new program.  PROGRAM is the name
7     of the program to run; the path will be searched for this program.
8     ARG_LIST is a NULL-terminated list of character strings to be
9     passed as the program's argument list.  Returns the process id of
10    the spawned process.  */
11
12 int spawn (char* program, char** arg_list)
13 {
14   pid_t child_pid;
15
16   /* Duplicate this process.  */
```

```
17    child_pid = fork ();
18    if (child_pid != 0)
19      /* This is the parent process.  */
20      return child_pid;
21    else {
22      /* Now execute PROGRAM, searching for it in the path.  */
23      execvp (program, arg_list);
24      /* The execvp function returns only if an error occurs.  */
25      fprintf (stderr, "an error occurred in execvp\n");
26      abort ();
27    }
28  }
29
30  int main ()
31  {
32    /* The argument list to pass to the "ls" command.  */
33    char* arg_list[] = {
34      "ls",       /* argv[0], the name of the program.  */
35      "-l",
36      "/",
37      NULL        /* The argument list must end with a NULL.  */
38    };
39
40    /* Spawn a child process running the "ls" command.  Ignore the
41       returned child process id.  */
42    spawn ("ls", arg_list);
43
44    printf ("done with main program\n");
45
46    return 0;
47  }
```

3.2.3 Scheduling dei processi

Linux schedula i processi padre e figlio indipendentemente; non c'è nessuna garanzia su quale dei due girerà prima o per quanto tempo esso girerà prima che Linux lo interrompa e faccia girare l'altro processo (o qualche altro processo nel sistema). In particolare, nessuna parte, oppure tutte, del comando ls, potrebbero girere nel processo foglio prima che il processo padre non sia stato completato.[2] Linux promette che ogni processo verrà eventualmente eseguito – nessun processo verrà completamente impoverito delle risorse di esecuzione.

Devi specificare che un processo è meno importante – e gli dovrebbe essere data una priorità più bassa – assegnandogli un valore alto di *niceness* (futilità). Per default, ogni processo ha un niceness pari a zero. Un alto valore di niceness indica che al processo viene

[2] Un metodo per serializzare i due processi è presentato nella sottosezione 3.4.1, "Attendere che un processo termini".

data una priorità di esecuzione minore; viceversa, un processo con un basso (negativo) niceness ottiene maggior tempo di esecuzione.

Per eseguire un programma con un niceness diverso da zero, usa il comando `nice`, specificando il valore niceness con l'opzione `-n`. Per esempio, questo è come dovresti invocare il comando "`sort input.txt > output.txt`", una lunga operazione di ordinamento con una priorità ridotta in modo che non rallenti troppo il sistema:

```
% nice -n 10 sort input.txt > output.txt
```

Puoi usare il comando `renice` da riga di comando per cambiare il niceness di un processo in esecuzione.

Per cambiare il niceness di un processo in esecuzione programmaticamente, usa la funzione `nice`. Il suo argomento è un valore di incremento, che è aggiunto al valore niceness del pocesso che esso chiama. Ricorda che un valore positivo incrementa il valore niceness e quindi riduce la priorità di esecuzione del processo.

Nota che solo un processo con privilegi di root può eseguire un processo con un valore niceness negativo o ridurre il valore niceness di un processo in esecuzione. Ciò significa che puoi specificare valori negativi ai comandi nice e renice solo quando sei loggato come root, e solo un processo che è in esecuzione come root può passare valori negativi alla funzione nice. Ciò evita che gli utenti ordinari si accaparrino la priorità di esecuzione togliendola ad altri che stanno usando il sistema.

3.3 Segnali

I segnali sono meccanismi per comunicare e manipolare i processi in Linux. L'argomento dei segnali è molto ampio; qui discutiamo su alcuni dei più importanti segnali e tecniche che sono usate per controllare i processi.

Un segnale è un messaggio speciale inviato ad un processo. I segnali sono asincroni; quando un processo riceve un segnale, esso processa il segnale immediatamente, senza completare la funzione corrente o anche la corrente riga di codice. Ci sono diverse dozzine di segnali divesi, ognuno con un diverso significato. Ogni tipo di segnale è specificato dal suo numero di segnale, ma nei programmi, di solito si fa riferimento ai segnali tramite il loro nome. Su Linux, questi sono definiti in `/usr/include/bits/signum.h` (Non dovresti includere questo file header direttamente nei tuoi programmi, piuttosto usa `<signal.h>`.)

Quando un processo riceve un segnale, può fare una delle diverse cose, dipendentemente dalle disposizioni del segnale. Per ogni segnale ci sono disposizioni di default, che determinano cosa accade al processo se il programma non specifica alcuni altri comportamenti. Per molti tipi di segnali, un programma può specificare alcuni altri comportamenti – tra ignorare il segnale o chiamare una funzione speciale `signal-handler` per rispondere al segnale. Se viene usato un gestore di segnale, l'esecuzione corrente del programma è messa in pausa, viene eseguito gestore del segnale e quando il gestore del segnale ritorna il programma riprende.

Il sistema Linux invia i segnali ai processi in risposta a condizioni specifiche. Per esempio, `SIGBUS` (errore del bus), `SIGSEGV` (segmentation violation) e `SIGFPE` (floating point exception) possono essere inviati ad un processo che tenta di eseguire un'operazione illegale. Le

disposizioni di default di questi segnali tentano di terminare il processo e produrre un file riassuntivo.

Un processo può anche inviare un segnale ad un altro processo. Un uso comune di questo meccanismo è quello di terminare un altro processo inviandogli un segnale `SITGERM` o `SIGKILL`.[3] Un altro uso comune è di inviare un comando ad un programma in esecuzione. Due segnali "userdefined" (definiti dall'utente) sono riservati per questo scopo: `SIGUSR1` e `SIGUSR2`. Qualche volta viene usato per questo scopo anche il segnale `SIGHUP`, comunemente per svegliare un programma in pausa o causare ad un programma la rilettura del suo file di configurazione.

La funzione `sigaction` può essere usata per settare una disposizione di un segnale. Il primo parametro è il numero del segnale. I successivi due parametri sono puntatori a strutture `sigaction`; il primo di essi contiene la disposizione desiderata per quel numero di segnale mentre il secondo riceve la disposizione precedente. Il campo più importante nella prima o seconda struttura `sigaction` è `sa_handler`. Esso può prendere uno dei tre valori:

- `SIG_DFL`, che specifica la disposizione di default per il segnale.
- `SIG_IGN`, che specifica che il segnale dovrebbe essere ignorato.
- Un puntatore ad una funzione che gestisce un segnale (signal-handler). La funzione dovrebbe prendere un parametro, il numero di segnale e restituire un `void`.

Poiché i segnali sono asincroni, il programma principale può trovarsi in uno stato molto fragile nel momento in cui viene processato un segnale e quindi mentre viene eseguita una funzione che gestisce il segnale.

Un gestore di segnale dovrebbe effettuare il minimo lavoro necessario per rispondere al segnale e quindi restituire il controllo al programma principale (o terminare il programma). In molti casi, esso consiste semplicemente nel registrare il fatto che si è ricevuto un segnale. Il programma principale quindi verifica periodicamente se è stato ricevuto un segnale e reagisce di conseguenza.

È possibile che un gestore di segnale venga interrotto dall'invio di un altro segnale. Anche se può sembrare un'occorrenza rara, se accade, sarà molto difficile diagnosticare e fare il debug del problema. (Questo è un esempio di una condizione rara, discussa nel Capitolo 4, "Threads", sezione 4.4, "Sincronizzazione e Sezioni Critiche". Quindi, dovresti stare molto attento su cosa fa il tuo programma in un gestore di segnale.

Anche assegnare un valore ad una variabile globale può essere pericoloso poiché l'assegnazione potrebbe essere effettuata in due o più istruzioni, e tra una e l'altra potrebbe presentarsi un secondo segnale, lasciando la variabile in uno stato corrotto. Se usi una variabile globale per memorizzare un segnale da una funzione gestore di segnali, essa dovrebbe essere del tipo speciale `sig_atomic_t`. Linux garantisce che gli assegnamenti a variabili di questo tipo sono effettuati in un'istruzione singola e quindi non possono essere interrotti a metà strada. In linux, `sig_atomic_t` è un `int` ordinario; infatti, gli assegnamenti ad interi che rappresentano

[3] Qual'è la differenza? Il segnale `SIGTERM` chiede ad un processo di terminare; il processo può ignorare la richiesta mascherando o ignorando il segnale. Il segnale `SIGKILL` uccide sempre il processo immediatamente poiché il processo non può mascherare o ignorare il `SIGKILL`.

la dimensione di un `int` o più piccolo, o a puntatori, sono atomici. Se vuoi scrivere un programma che sia portabile verso ogni sistema UNIX standard, comunque, usa `sig_atomic_t` per queste variabili globali.

Questo abbozzo di programma nel listato 3.5, per esempio, usa una funzione gestore di segnali per contare il numero di volte che il programma riceve `SIGUSR1`, uno dei segnali riservati per l'utilizzo nella applicazioni.

Listato 3.5: (*sigusr1.c*) – **Usare un gestore di segnale**

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

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

3.4 Terminazione dei processi

Normalmente, un processo termina in uno di due modi. O il programma in esecuzione chiama la funzione `exit`, o la funzione `main` del programma finisce. Ogni processo ha un codice di uscita: un numero che il processo restituisce al suo genitore. Il codice di uscita è l'argomento passato alla funzione `exit`, o il valore restituito dal `main`.

Un processo può anche terminare non normalmente, in risposta ad un segnale. Per esempio, i segnali `SIGBUS`, `SIGSEGV` e `SIGFPE` menzionati precedentemente causano la terminazione del processo. Altri segnali sono usati per terminare il processo esplicitamente. Il segnale `SIGINT` è inviato ad un processo quando l'utente cerca di terminarlo premendo Ctrl+C nel proprio

terminale. Il segnale `SIGTERM` è inviato dal comando `kill`. La direttiva di default per entrambi è quella di teminare il processo. Chiamando la funzione `abort`, un processo stesso si manda il senale `SIGABRT`, che emina il processo e produce un file core. Il segnale di terminazione più potente è `SIGKILL`, che termina un processo immediatamente e non può essere bloccato o gestito da un programma.

Ognuno di questi segnali può essere inviato usando il comando `kill` specificando un argomento extra da riga di comando; per esempio, per terminare un processo che ha dei problemi inviandogli un `SIGKILL`, invoca la seguente, dove `pid` è il suo ID di processo:

```
% kill -KILL pid
```

Per inviare un segnale da un programma, usa la funzione `kill`. Il primo parametro è l'ID del processo obiettivo. Il secondo parametro è il numero di segnale; usa `SIGTERM` per simulare il comportamento di default del comando `kill`. Per sempio, dove `child pid` contiene l'iID di processo del processo figlio, puoi usare la funzion `kill` per terminare un processo figlio da un processo padre chiamandolo in un modo simile a questo:

```
kill (child_pid, SIGTERM);
```

Se usi la funzione `kill` includi gli header `<sys/types.h>` e `<signal.h>`.

Per convenzione, il codice di uscita è usato per indicare se un programma è stato eseguito correttamente. Un codice di uscita pari a zero indica la corretta esecuzione, mentre un codice di uscita diverso da zero indica che c'è stato un errore. Nell'ultimo caso, il valore particolare restituito può dare alcune indicazione sulla natura dell'errore. Sarebbe una buona idea riportare questa convenzione nei tuoi programmi poiché altri componenti del sistema GNU/Linux assumono che ci sia questo comportamento. Per esempio, la shell assume la presenza di questa convenzione quando colleghi più programmi con gli operatori `&&` (and logico) e `||` (or logico. Quindi, dovresti restituire zero dal `main` a meno che non si verifichi un errore.

Con molte shell è possibile ottenere il codice di uscita del programma eseguito più recentemente usando la variabile speciale `$?`. Ecco un esempio nel quale il comando `ls` è invocato due volte ed è mostrato il codice di uscita dopo ogni chiamata. Nel primo caso, `ls` viene eseguito correttamente e restituisce il codice di uscita zero. Nel secondo caso, `ls` incontra un errore (perché il nome del file specificato sulla riga di comando non esiste) e quindi restituisce un codice di uscita diverso da zero.

```
% ls /
bin   coda  etc   lib         misc  nfs proc  sbin  usr
boot  dev   home  lost+found  mnt   opt root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Nota che benché il tipo di parametro della funzione `exit` sia un `int` e la funzione `main` restituisca un `int`, Linux non riserva tutti i 32 bit del codice di ritorno. In fatti, dovresti

usare solo codici di uscita compresi tra zero e 127. I codici di uscita superiori a 128 hanno uno speciale significato – quando un processo è terminato da un segnale, il suo codice di uscita è 128 più il numero del segnale.

3.4.1 Attendere che un processo termini

Se hai scritto ed eseguito l'esempio `fork` ed `exec` nel listato 3.4, dovresti aver notato che l'output del programma `ls` spesso appare dopo che il "programma principale" è già stato completato. Ciò accade perché il processo figlio, nel quale gira `ls`, è schedulato indipendentemente dal processo padre. Poiché Linux è un sistema operativo multitasking, entrambi i processi appaiono come eseguiti simultaneamente e non puoi predire quando il programma `ls` avrà una possibilità di essere eseguito, prima o dopo che venga eseguito il processo padre.

In alcune situazioni, comunque, è preferibile che il processo padre aspetti fino a che uno o più processi figli non siano stati completati. Ciò può essere fatto con la famiglia di chiamate di sistema `wait`. Queste funzioni ti permettono di aspettare che un processo completi la propria esecuzione ed abilitare il processo padre ad ottenere informazioni sulla terminazione dei suoi processi figli. Ci sono quattro diverse chiamate di sistema nella famiglia `wait`; puoi scegliere di ottenere poche o molte informazioni sul processo che è terminato e puoi scegliere se interessarti su quale processo figlio è stato terminato.

3.4.2 Le chiamate di sistema *wait*

La funzione più semplice è chiamata semplicemente `wait`. Essa blocca il processo chiamante fino a che uno dei suoi processi figli non esce (o non c'è un errore). Esso restituisce un codice di stato tramite un argomento di tipo puntatore ad intero, dal quale puoi estrarre informazioni su come il processo figlio è uscito. Per esempio, la macro `WEXITSTATUS` estrae il codice di uscita del processo figlio.

Puoi usare la macro `WIFEXITED` per determinare dallo stato di uscita di un processo figlio se il processo è terminato normalmente (tramite la funzione `exit` o uscendo dal `main`) o è morto in seguito ad un segnale non gestito. Nell'ultimo caso, usa la macro `WTERMSIG` per estrarre, dal suo stato di uscita, il numero di segnale per il quale è morto.

Qui c'è nuovamente la funzione `main` dall'esempio `fork` ed `exec`. Questa volta, il processo padre chiama `wait` per attendere fino a che il processo figlio, che è l'esecuzione del comando `ls`, non è finito.

```
int main ()
{
  int child_status;

  /* The argument list to pass to the "ls" command. */
  char* arg_list[] = {
    "ls", /* argv[0], the name of the program. */
    "-l",
    "/",
    NULL /* The argument list must end with a NULL. */
};
```

```
/* Spawn a child process running the "ls" command. Ignore the
   returned child process ID. */
spawn ("ls", arg_list);
/* Wait for the child process to complete. */
wait (&child_status);
if (WIFEXITED (child_status))
  printf ("the child process exited normally, with exit code %d\n",
          WEXITSTATUS (child_status));
else
  printf ("the child process exited abnormally\n");

return 0;
}
```

In Linux sono disponibili molte chiamate di sistema simili, che sono più flessibili o forniscono più informazioni sull'uscita del processo figlio. La funzione `waitpid` può essere usata per attendere che finisca uno specifico processo figlio piuttosto che ogni processo figlio. La funzione `wait3` restituisce statistiche di utilizzo della CPU riguardo l'uscita del processo figlio e la funzione `wait4` ti permette di specificare altre opzioni riguardo a quale processo attendere.

3.4.3 Processi zombie

Se un processo figlio termina mentre il suo genitore sta chiamando una funzione `wait`, il processo figlio scompare ed il suo stato di uscita è passato al suo padre tramite la chiamata `wait`. Ma cosa accade quando un processo figlio termina ed il processo figlio non sta chiamando `wait`? Esso semplicemente scopare? No, perché l'informazione sulla sua terminazione – come quella se è terminato normalmente e, se così, qual è il suo stato di uscita – andrebbe persa. Invece, quando un processo figlio termina, diventa un processo zombie.

Un *processo zombie* è un processo che è terminato ma non è stato ancora cancellato. È responsabilità del processo padre cancellare il proprio figlio zombie. Le funzioni `wait` anno anche questo, così non è necessario verificare se il tuo processo figlio sta ancora girando prima di mettersi in attesa per questo. Supponi, per esempio, che un programma faccia il fork in un processo figlio, esegua qualche altra operazione, e quindi chiami `wait`. Se il processo figlio in quel momento non è ancora terminato, il processo padre resterà bloccato nella chiamata `wait` fino a che il processo figlio non abbia finito. Se il processo figlio finisce prima che il processo padre chiami `wait`, il processo figlio diventa uno zombie. Quando il processo padre chiama `wait`, viene estratto lo stato di uscita del figlio zombie, il processo figlio è cancellato e la chiamata `wait` ritorna immediatamente.

Cosa accade se il processo padre non cancella il proprio figlio? Essi stanno in giro per il sistema, come processi zombie. Il programma nel listato 3.6 fa il fork in un processo figlio, che termina immediatamente e quindi aspetta per un minuto senza mai cancellare il processo figlio.

Listato 3.6: (*zombie.c*) – **Creare un processo zombie**

```
1 #include <stdlib.h>
```

```
2  #include <sys/types.h>
3  #include <unistd.h>
4
5  int main ()
6  {
7    pid_t child_pid;
8
9    /* Create a child process.  */
10   child_pid = fork ();
11   if (child_pid > 0) {
12     /* This is the parent process.  Sleep for a minute.  */
13     sleep (60);
14   }
15   else {
16     /* This is the child process.  Exit immediately.  */
17     exit (0);
18   }
19   return 0;
20 }
```

Prova a compilare questo file in un eseguibile chiamato `make-zombie`. Eseguilo e, mentre è ancora in esecuzione, visualizza l'elenco dei processi nel sistema invocando il seguente comando in un'altra finestra:

```
% ps -e -o pid,ppid,stat,cmd
```

Questo elenca gli ID di processo, ID di processo padre, stato del processo e riga di comando del processo. Osserva che, in aggiunta al processo padre `make-zombie`, c'è un altro processo visualizzato `make-zombie`. Questo è il processo figlio; nota che l'ID del suo processo padre è l'ID del processo principale `make-zombie`. Il processo figlio è markato come `<defunct>`, ed il suo codice di uscita è Z, che sta per zombie.

Cosa accade quando il programma principale `make-zombie` termina nel momento in cui il processo padre esce senza aver mai chiamato `wait`? Il processo zombie sta ancora in giro? No – prova ad eseguire nuovamente `ps` e nota che entrambi i processi `make-zombie` sono andati via. Quando un programma termina, il suo figlio è ereditato da un processo speciale, il programma `init`, che gira sempre con l'ID di processo pari a 1 (Esso è il primo processo avviato quando Linux si avvia). Il processo `init` cancella automaticamente ogni processo figlio zombie che esso eredita.

3.4.4 Cancellare i figli in maniera asincrona

Se stai usando un processo figlio, semplicemente per eseguire (`exec`) un altro programma, è bene chiamare `wait` immediatamente nel processo padre, che si bloccherà fino a che il processo figlio non è stato completato. Ma speso, vorrai che il processo padre continui la sua esecuzione mentre uno o più figli vengono eseguiti in maniera asincrona. Come puoi essere sicuro di cancellare i processi figli che hanno completato la loro esecuzione in modo da non lasciare processi zombie, che consumano risorse di sistema, e stanno in giro?

Un approccio per il processo padre potrebbe essere chiamare `wait3` o `wait4` periodicamente per cancellare figli zombie. Chiamare `wait` per questo scopo non funziona bene perché, se nessun figlio ha terminato, la chiamata bloccherà il processo padre finché uno dei figli non avrà terminato. Comunque, `wait3` e `wait4` prendono un parametro flag aggiuntivo, al quale puoi passare il valore flag `WNOHANG`. Con questo flag, la funzione gira in *modalità nonblocking* – esso cancellerà un processo figlio terminato, se c'è, o semplicemente dice se non ce ne sono. Il valore di ritorno della chiamata è l'ID di processo del figlio terminato nel caso più comune, o zero nell'ultimo caso.

Una soluzione più elegante è notificare al processo padre quando un figlio termina. Ci sono molti modi per farlo usando i metodi discussi nel Capitolo 5, "Comunicazione tra processi", ma fortunatamente Linux lo fa al posto tuo, usando i segnali. Quando un processo figlio termina, Linux invia un segnale al processo padre, il segnale `SIGCHILD`. La disposizione di default di questo segnale è quella di non fare nulla, che è il motivo per il quale puoi non averlo notato prima.

Comunque, un modo semplice per cancellare i processi figli è catturando il segnale `SIGCHILD`. Di certo, quando si cancella un processo figlio, è importante memorizzare il suo stato di terminazione, se questa informazione è necessaria, perché quando il processo sarà stato cancellato usando `wait`, questa informazione non sarà più disponibile. Il listato 3.7 mostra un esempio di un programma che usa il gestore `SIGCHILD` per cancellare i suoi processi figli.

Listato 3.7: (***sigchld.c***) – **Cancellare i figli gestendo SIGCHILD**

```
1  #include <signal.h>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5
6  sig_atomic_t child_exit_status;
7
8  void clean_up_child_process (int signal_number)
9  {
10   /* Clean up the child process.  */
11   int status;
12   wait (&status);
13   /* Store its exit status in a global variable.  */
14   child_exit_status = status;
15 }
16
17 int main ()
18 {
19   /* Handle SIGCHLD by calling clean_up_child_process.  */
20   struct sigaction sigchld_action;
21   memset (&sigchld_action, 0, sizeof (sigchld_action));
22   sigchld_action.sa_handler = &clean_up_child_process;
23   sigaction (SIGCHLD, &sigchld_action, NULL);
24
```

```
25    /* Now do things, including forking a child process.  */
26    /* ...  */
27
28    return 0;
29  }
```

Nota come il gestore di segnale memorizza lo stato di uscita del processo figlio in una variabile globale, per mezzo della quale il programma principale vi può accedere. Poiché la variabile è assegnata in un gestore di segnale, il suo tipo è `sig_atomic_t`.

Capitolo 4

Threads

I thread, come i processi sono un meccanismo per permettere ad un programma di fare più di una cosa allo stesso tempo. Come con i processi, i thread appaiono in esecuzione concorrente; Il kernel di Linux li schedula in maniera sincrona, interrompendo ogni thread da un momento all'altro per dare agli altri la possibilità di andare in esecuzione.

Concettualmente, un thread esiste in un processo. I thread sono unità di esecuzione più piccole dei processi. Quando invochi un programma, Linux crea un nuovo processo ed in quel processo crea un singolo thread, che esegue il programma sequenzialmente. Questo thread può creare altri thread; tutti questi thread eseguono lo stesso programma nello stesso processo, ma ogni thread può essere l'esecuzione di una diversa parte del programma in qualsiasi momento.

Abbiamo visto come un programma può fare il fork in un processo figlio. Il processo figlio inizialmente gira nel suo programma padre, con la memoria virtuale del padre, descrittori di file e così via copiati. Il processo figlio può modificare la propria memoria, chiudere descrittori di file, e simili, senza effetti sul padre, e vice versa. Quando un programma crea un altro thread, comunque, non viene copiato nulla. Il thread creatore ed il thread creato condividono lo stesso spazio di memoria, descrittori di file ed altre risorse di sistema con l'originale. Se un thread cambia il valore di una variabile, per esempio, l'altro thread di conseguenza vedrà il valore modificato. Similmente, se un thread chiude un descrittore di file, gli altri thread non potranno leggere o scrivere in quel descrittore di file. Poiché un processo e tutti i suoi thread possono eseguire solo un programma per volta, se ogni thread all'interno di un processo chiama una delle funzioni `exec`, tutti gli altri thread vengono terminati (il nuovo programma può, di certo, creare nuovi thread).

GNU/Linux implementa le API di thread standard POSIX (conosciute come *pthreads*). Tutte le funzioni di thread e tipi di dati sono dichiarati nel file header `<pthread.h>`. Le funzioni pthread non sono incluse nella libreria C standard. Piuttosto, esse sono in `libpthread`, quindi devi aggiungere `-lpthread` alla riga di comando quando fai il link del tuo programma.

4.1 Creazione dei thread

Ogni thread in un processo è identificato da un *thread ID*. Quando si fa riferimento agli ID dei thread nei programmi C o C++, si usa il tipo `pthread_t`.

Dopo la creazione, ogni thread esegue una *funzione thread*. Questa è una funzione ordinaria e contiene il codice che il thread dovrebbe eseguire. Quando la funzione ritorna, il thread esce. Su GNU/Linux, le funzioni thread prendono un singolo parametro, di tipo `void*`, ed hanno un tipo di ritorno `void*`. Il parametro è l'*argomento del thread*: GNU/Linux passa il valore nel thread senza guardarlo. Il tuo programma può usare questo parametro per passare i dati ad un nuovo thread. Similmente, il tuo programma può usare il valore di ritorno per passare dati da un thread esistente al suo programma creatore.

La funzione `pthread_create` crea un nuovo thread. La usi assieme ai seguenti:

1. Un puntatore alla variabile `pthread_t`, nel quale è memorizzato l'ID di thread del nuovo thread

2. Un puntatore ad un oggetto *attributo thread*. Questo oggetto controlla i dettagli di come i thread interagiscono con il resto del programma. Se passi `NULL` come attributo di thread, verrà creato un thread con gli attributi di thread di default. Gli attributi di thread sono discussi nella sottosezione 4.1.5, "Attributi dei thread".

3. Un puntatore alla funzione thread. Questo è una funzione puntatore ordinaria, di questo tipo:

```
void* (*) (void*)
```

4. Un argomento di thread di tipo `void*`. Ogni cosa che passi è semplicemente passata come argomento della funzione thread quando il thread inizia la sua esecuzione.

Una chiamata a `pthread_create` ritorna immediatamente ed il thread originale continua ad eseguire le istruzioni seguenti la chiamata. Nel frattempo, il nuovo thread inizia ad eseguire la funzione thread. Linux schedula entrambi i thread in maniera asincrona e il tuo programma non deve fare affidamento sull'ordine relativo nel quale sono eseguite le istruzioni nei due threads.

Il programma nel listato 4.1 crea un thread che stampa continuamente `x` sullo standard error. Dopo aver chiamato `pthread_create`, il thread principale stampa continuamente `o` sullo standard error.

Listato 4.1: (***thread-create.c***) – **Creare un thread**

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 /* Prints x's to stderr.  The parameter is unused.  Does not return.  */
5
6 void* print_xs (void* unused)
```

```
7  {
8    while (1)
9      fputc ('x', stderr);
10   return NULL;
11 }
12
13 /* The main program.  */
14
15 int main ()
16 {
17   pthread_t thread_id;
18   /* Create a new thread.  The new thread will run the print_xs
19      function.  */
20   pthread_create (&thread_id, NULL, &print_xs, NULL);
21   /* Print o's continuously to stderr.  */
22   while (1)
23     fputc ('o', stderr);
24   return 0;
25 }
```

Compila e fai il link di questo programma usando il seguente codice:

```
% cc -o thread-create thread-create.c -lpthread
```

Prova ad eseguirlo per vedere cosa accade. Nota il modello di `x` ed `o` imprevedibile, come Linux schedula alternativamente i due thread.

In circostanze normali, un thread esce in uno di due modi. Un modo, come illustrato precedentemente, è ritornando dalla funzione thread. Il valore di ritorno dalla funzione thread è considerato essere il valore di ritorno del thread. In alternativa, un thread può uscire esplicitamente chiamando `pthread_exit`. Questa funzione può essere chiamata dall'interno della funzione thread. L'argomento di `pthread_exit` è il valore di ritorno del thread.

4.1.1 Passare dati ai thread

L'argomento del thread fornisce un modo conveniente per passare i dati ai thread. Poiché il tipo dell'argomento è `void*`, comunque, non puoi passare molti dati direttamente tramite l'argomento. Piuttosto, usa l'argomento del thread per passare un puntatore ad alcune strutture o array di dati. Una tecnica comunemente usata è quella di definire una struttura per ogni funzione thread, che contiene i "parametri" che la funzione thread si aspetta.

Usando l'argomento del thread, è facile riutilizzare la stessa funzione thread per molti thread. Tutti questi thread eseguono lo stesso codice, ma con dati diversi.

Il programma nel listato 4.2, è simile all'esempio precedente. Questa volta vengono creati due nuovi thread, uno per stampare `x` e l'altro per stampare `o`. Invece di stampare infinitamente, comunque, ogni thread stampa un numero fisso di caratteri e quindi esce ritornando dalla funzione thread. La stessa funzione thread, `char_print`, è usata da entrambi i thread, ma ognuno è configurato diversamente usando `struct char_print_parms`.

Listato 4.2: (***thread-create2.c***) – **Creare due threads**

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  /* Parameters to print_function.  */
5
6  struct char_print_parms
7  {
8    /* The character to print.  */
9    char character;
10   /* The number of times to print it.  */
11   int count;
12 };
13
14 /* Prints a number of characters to stderr,
15    as given by PARAMETERS, which is a
16    pointer to a struct char_print_parms.  */
17
18 void* char_print (void* parameters)
19 {
20   /* Cast the cookie pointer to the right type.  */
21   struct char_print_parms* p =
22           (struct char_print_parms*) parameters;
23   int i;
24
25   for (i = 0; i < p->count; ++i)
26     fputc (p->character, stderr);
27   return NULL;
28 }
29
30 /* The main program.  */
31
32 int main ()
33 {
34   pthread_t thread1_id;
35   pthread_t thread2_id;
36   struct char_print_parms thread1_args;
37   struct char_print_parms thread2_args;
38
39   /* Create a new thread to print 30000 x's.  */
40   thread1_args.character = 'x';
41   thread1_args.count = 30000;
42   pthread_create (&thread1_id, NULL,
43           &char_print, &thread1_args);
44
45   /* Create a new thread to print 20000 o's.  */
46   thread2_args.character = 'o';
47   thread2_args.count = 20000;
48   pthread_create (&thread2_id, NULL,
```

```
49            &char_print, &thread2_args);
50
51    return 0;
52  }
```

Ma aspetta! Il programma nel listato 4.2 contiene un bug serio. Il thread principale (che esegue la funzione `main`) crea le strutture del parametro del thread (`thread1_args` e `thread2_args`) come variabili loacali, e quindi passa i puntatori a queste strutture ai thread che esso crea. Cosa fare per evitare che Linux scheduli i tre thread in modo che `main` completi la sua esecuzione prima che uno degli altri due thread sia completato? *Nulla!* Ma se ciò accade, la memoria contenente le strutture del parametro del thread verranno deallocate mentre gli altri due thread staranno ancora accedendo ad esse.

4.1.2 Unire i thread

Una soluzione è quella di forzare il `main` ad aspettare fino a che gli altri due thread non siano compleatati. Ciò di cui abbiamo bisogno è una funzione simile a `wait` che aspetti che finisca un thread piuttosto che un processo. La funzione è `pthread_join`, che prende due argomenti: l'ID del thread da aspettare ed un puntatore ad una variabile `void*` che riceverà il valore di ritorno del thread finito. Se non ti interessa il valore di ritorno del thread, passa `NULL` come secondo argomento.

Il listato 4.3 mostra la funzione `main` corretta per l'esempio sbagliato nel listato 4.2. In questa versione, il `main` non esce fino a che entrambi i thread che stampano x e o non sono completati, così essi non useranno più le strutture degli argomenti.

Listato 4.3: (***thread-create2.c***) – **Funzione *Main* rivista per *thread-create2.c***

```
1  int main ()
2  {
3    pthread_t thread1_id;
4    pthread_t thread2_id;
5    struct char_print_parms thread1_args;
6    struct char_print_parms thread2_args;
7
8    /* Create a new thread to print 30,000 x's. */
9    thread1_args.character = 'x';
10   thread1_args.count = 30000;
11   pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
12
13   /* Create a new thread to print 20,000 o's. */
14   thread2_args.character = 'o';
15   thread2_args.count = 20000;
16   pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
17
18   /* Make sure the first thread has finished. */
19   pthread_join (thread1_id, NULL);
20   /* Make sure the second thread has finished. */
```

```
21    pthread_join (thread2_id, NULL);
22
23    /* Now we can safely return. */
24    return 0;
25  }
```

Morale della favola: Assicurati che ogni dato che passi a un thread per riferimento non sia deallocato, *anche da un diverso thread*, fino a che non sei sicuro che il thread abbia finito con esso. Ciò è vero sia per variabili locali, che sono deallocate quando esse vanno fuori dal proprio raggio di azione, che per gruppi di variabili allocate in blocco che vengono deallocate chiamando `free` (o usando `delete` in C++).

4.1.3 Valori di ritorno dei thread

Se il secondo argomento che passi a `pthread_join` non è nullo, il valore di ritorno del thread verrà messo nella locazione puntata da quell'argomento. Il valore di ritorno del thread, come l'argomento del thread, è di tipo `void*`. Se vuoi passare un singolo `int` o altri numeri piccoli, puoi farlo facilmente tramite il cast del valore a `void*` e quindi rifacendo il cast al tipo appropriato dopo aver chiamato `pthread_join`.[1]

Il programma nel listato 4.4 calcola l'ennesimo numero primo in un thread separato. Quel thread restituisce il numero primo desiderato come suo valore di ritorno di thread. Il thread principale, nel frattempo, è libero di eseguire altro codice. Nota che l'agoritmo di divisione successivo usato in `compute_prime` è un po' inefficiente; se nel tuo programma hai bisogno di calcolare molti numeri primi consulta un libro su algoritmi numerici.

Listato 4.4: ***(primes.c)*** **– Calcola numeri primi in un thread**

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  /* Compute successive prime numbers (very inefficiently). Return the
5     Nth prime number, where N is the value pointed to by *ARG. */
6
7  void* compute_prime (void* arg)
8  {
9    int candidate = 2;
10   int n = *((int*) arg);
11
12   while (1) {
13     int factor;
14     int is_prime = 1;
15
16     /* Test primality by successive division. */
17     for (factor = 2; factor < candidate; ++factor)
```

[1] Nota che ciò non è portabile ed è compito tuo assicurarti che per il valore può essere effettuato il cast a `void*` e tornare indietro senza perdita di bit.

```
18        if (candidate % factor == 0) {
19          is_prime = 0;
20          break;
21        }
22      /* Is this the prime number we're looking for? */
23      if (is_prime) {
24        if (--n == 0)
25          /* Return the desired prime number as the thread return value. */
26          return (void*) candidate;
27      }
28      ++candidate;
29    }
30    return NULL;
31  }
32
33  int main ()
34  {
35    pthread_t thread;
36    int which_prime = 5000;
37    int prime;
38
39    /* Start the computing thread, up to the 5000th prime number. */
40    pthread_create (&thread, NULL, &compute_prime, &which_prime);
41    /* Do some other work here... */
42    /* Wait for the prime number thread to complete, and get the result. */
43    pthread_join (thread, (void*) &prime);
44    /* Print the largest prime it computed. */
45    printf("The %dth prime number is %d.\n", which_prime, prime);
46    return 0;
47  }
```

4.1.4 Altro sugli ID dei thread

Occasionalmente, è utile per una sequenza di codice determinare quale thread lo sta eseguendo. La funzione `pthread_self` restituisce l'ID di thread del thread nel quale è chiamata. Questo ID di thread può essere confrontato con altri ID di thread usando la funzione `pthread_equal`

Queste funzioni possono essere utili per determinare quando un particolare ID di thread corrisponde al thread corrente. Per esempio, c'è un errore per un thread nel chiamare `pthread_join` per unirlo a se stesso. (In questo caso, `pthread_join` restituirebbe il codice di errore `EDEADLK`.) Per verificare questa condizione in anticipo, puoi usare del codice come questo:

```
if (!pthread_equal (pthread_self (), other_thread))
  pthread_join (other_thread, NULL);
```

4.1.5 Attributi dei thread

Gli attributi dei thread forniscono un meccanismo per mettere a punto il comportamento di thread individuali. Ricorda che `pthread_create` accetta un argomento che è un puntatore ad un oggetto attributo di thread. Se passi un puntatore nullo, per configurare il nuovo thread vengono utilizzati gli attributi di default dei thread. Comunque, puoi creare e personalizzare l'oggetto attributo di un thread per specificare altri valori per gli attributi.

Per specificare gli attributi dei thread è necessario seguire i seguenti passi:

1. Creare un oggetto `pthread_attr_t`. La maniera più facile è dichiarare semplicemente una variabile automatica di questo tipo.

2. Chiamare `pthread_attr_init`, passando un puntatore a questo oggetto. Ciò inizializza gli attributi ai loro valori di default.

3. Modificare l'oggetto attributo in modo che contenga i valori desiderati.

4. Passare un puntatore all'oggetto attributo quando si chiama `pthread_create`.

5. Chiamare `pthread_attr_destroy` per rilasciare l'oggetto attributo. La variabile `pthread_attr_t` da sola non è deallocata; può essere reinizializzata con `pthread_attr_init`.

Un singolo oggetto attributo di thread può essere usato per avviare diversi thread. Non è necessario mantenere un oggetto attributo di thread in giro dopo che i thread sono stati creati.

Per molti lavori di programmazione delle applicazioni in GNU/Linux, è tipicamente di interesse un solo attributo di thread (gli altri attributi disponibili servono principalmente per la programmazione realtime specializzata). Questo attributo è lo *stato distaccato* (*detached state*) del thread. Un thread può essere creato come un *thread congiungibile* (di default) o come un *thread distaccato*. Un thread congiungibile (*joinable thread*), come un processo, non è cancellato automaticamente da GNU/Linux quando termina. Invece, lo stato di uscita del thread rimane in sospeso nel sistema (qualcosa di simile ad un processo zombie) finché un altro thread non chiama `pthread_join` per ottenere il suo valore di ritorno. Solo allora le sue risorse vengono rilasciate. Un thread distaccato, di contro, viene cancellato automaticamente quando termina. Poiché un thread distaccato è immediatamente cancellato, un altro thread può non riuscire ad essere sincronizzato con il suo completamento usando `pthread_join` o ottenere il suo valore di ritorno.

Per impostare lo stato distaccato in un oggetto attributo di thread, usa `pthread_attr_setdetachsta`
Il primo argomento è un puntatore all'oggetto attributo del thread, e il secondo è lo stato distaccato desiderato. Poiché lo stato joinable è quello di default, è necessario fare questa chiamata solo per creare thread distaccati; passa `PTHREAD_CREATE_DETACHED` come secondo argomento.

Il codice nel listato 4.5 crea un thread distaccato impostando l'attributo di thread allo stato detach per il thread.

Listato 4.5: **(*detached.c*) – Scheletro di un programma che crea un thread distaccato**

```
#include <pthread.h>

void* thread_function (void* thread_arg)
{
  /* Do work here...  */
  return NULL;
}

int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /* Do work here...  */

  /* No need to join the second thread.  */
  return 0;
}
```

Anche se un thread è creato in uno stato joinable, può essere successivamente cambiato in thread distaccato. Per fare ciò, chiama `pthread_detach`. Una volta che un thread diventa distaccato, non può più tornare ad essere nuovamente congiungibile (*joinable*).

4.2 Cancellazione di thread

In circostanze normali, un thread finisce quando esce normalmente, sia ritornando dalla sua funzione thread sia chiamando `pthread_exit`. Comunque, è possibile per un thread richiedere che un altro thread di termini. Ciò è chiamato *cancellazione* (*canceling*) di un thread.

Per cancellare un thread, chiama `pthread_cancel` passandogli l'ID del thread che deve essere cancellato. Un thread cancellato può essere successivamente congiunto (joined); infatti, dovresti collegarti ad un thread cancellato per liberarne le risorse, fino a che il thread è disgiunto (vedi sottosezione 4.1.5, "Attributi dei thread"). Il valore di ritorno di un thread cancellato è il valore speciale dato da `PTHREAD_CANCELED`.

Spesso un thread può trovarsi in del codice che può essere eseguito in maniera completa o nulla, senza vie di mezzo. per esempio, il thread può allocare delle risorse, usarle e quindi deallocarle. Se il thread viene cancellato nel mezzo di questo codice, può non avere l'opportunità di dellocare le risorse e quindi le risorse resteranno bloccate. Per cercare di evitare che ciò accada è possibile per un thread controllare se e quando esso può essere cancellato.

Un thread, per quanto riguarda la cancellazione, può trovarsi in uno di tre stati.

- Il thread può essere *cancellabile in modo asincrono* (*asynchronously cancelable*). Il thread può essere cancellato in ogni punto della sua esecuzione.
- Il thread può essere *cancellabile in modo sincrono* (*synchronously cancelable*). Il thread può essere cancellato, ma non proprio ad ogni punto della sua esecuzione. Piuttosto, le richieste di cancellazione sono accodate e il thread è cancellato solo quando raggiunge punti specifici nella sua esecuzione.
- un thread può essere *incancellabile* (*uncancelable*). I tentativi di cancellare il thread sono silenziosamente ignorati

Quando viene inizialmente creato, un thread è cancellabile in modo sincrono.

4.2.1 Thread sincroni ed asincroni

Un thread cancellabile in modo asincrono può essere cancellato in ogni punto della sua esecuzione. Un thread cancellabile in modo sincrono, di contro, può essere cancellato solo in particolari punti della sua esecuzione. Questi punti sono chiamati *punti di cancellazione* (*cancellation points*). Nel thread verrà accodata una richiesta di cancellazione fino a che esso non raggiunge il prossimo punto di cancellazione.

Per rendere un thread cancellabile in modo asincrono, usa `pthread_setcanceltype`. Ciò ha effetto sui thread che attualmente chiamano le funzioni. Il primo argomento dovrebbe essere `PTHREAD_CANCEL_ASYNCHRONOUS` per rendere il thread cancellabile in modo asincrono, o `PTHREAD_CANCEL_DEFERRED` per riportarlo nello stato di cancellabile in modo sincrono. Il secondo argomento, se diverso da null, è un puntatore ad una variabile che riceve il tipo di cancellazione precedente per il thread. Questa chiamata, per esempio, rende il thread chiamante cancellabile in modo asincrono.

```
pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Cosa costituisce un punto di cancellazione e dove dovrebbero essere posti? La via più diretta per creare un punto di cancellazione è chiamare `pthread_testcancel`. Questo non fa nulla ad eccetto il fatto di processare una cancellazione in sospeso in un thread cancellabile in modo sincrono. Dovresti chiamare `pthread_testcancel` periodicamente durante i calcoli più lunghi in una funzione thread, nei punti in cui il thread può essere cancellato senza bloccare nessuna risorsa o produrre altri effetti negativi. Certe altre funzioni hanno implicitamente dei punti di cancellazione. Questi sono elencati nella pagina di manuale di `pthread_cancel`. Nota che altre funzioni possono usare queste funzioni internamente e quindi ci saranno indirettamente dei punti di cancellazione.

4.2.2 Sezioni critiche non cancellabili

Un thread può disabilitare la cancellazione di se stesso completamente con la funzione `pthread_setcance`
Come per `pthread_setcanceltype`, questa ha effetto sul thread chiamante. Il primo argomento è `PTHREAD_CANCEL_DISABLE` per disabilitare la cancellazione, o `PTHREAD_CANCEL_ENABLE`

per riabilitare la cancellazione. Il secondo argomento, se diverso da null, punta ad una variabile che riceverà lo stato precedente di cancellazione. Questa chiamata, per esempio, disabilita la cancellazione del thread nel thread chiamante.

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

L'uso di `pthread_setcancelstate` ti permette di implementare *sezioni critiche* (*critical section*). Una sezione critica è una sequenza di codice che può essere eseguito solo interamente o per nulla; in altre parole, se un thread inizia ad eseguire la sezione critica, deve continuare fino alla fine della sezione critica senza essere cancellato.

Per esempio, supponi di stare scrivendo una routine per una programma bancario che trasferisce denaro da un conto ad un altro. Per farlo, devi aggiungere valori sul bilancio di un conto e detrarre gli stessi valori dal bilancio di un altro conto. Se succede che il thread che sta eseguendo la tua routine è stato cancellato proprio nel momento sbagliato, tra le due operazioni, il programma potrebbe avere incrementato falsamente il totale del deposito bancario facendo fallire il completamento della transazione. Per prevenire questa possibilità poni le due operazioni in una sezione critica.

Puoi implementare il trasferimento con una funzione come `process_transaction`, mostrata nel listato 4.6. Questa funzione disabilita la cancellazione del thread per avviare una sezione critica prima di modificare uno dei bilanci del conto.

Listato 4.6: (***critical-section.c***) – **Protegge una transazione bancaria con una sezione critica**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  /* An array of balances in accounts, indexed by account number. */
6
7  float* account_balances;
8
9  /* Transfer DOLLARS from account FROM_ACCT to account TO_ACCT. Return
10    0 if the transaction succeeded, or 1 if the balance FROM_ACCT is
11    too small. */
12
13 int process_transaction (int from_acct, int to_acct, float dollars)
14 {
15   int old_cancel_state;
16
17   /* Check the balance in FROM_ACCT. */
18   if (account_balances[from_acct] < dollars)
19     return 1;
20
21   /* Begin critical section. */
22   pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
23   /* Move the money. */
```

```
24    account_balances[to_acct] += dollars;
25    account_balances[from_acct] -= dollars;
26    /* End critical section.  */
27    pthread_setcancelstate (old_cancel_state, NULL);
28
29    return 0;
30  }
```

Nota che è importante ripristinare il vecchio stato di cancellato alla fine della sezione critica piuttosto che impostarlo incondizionatamente a `PTHREAD_CANCEL_ENABLE`. Ciò ti permette di chiamare la funzione `process_transaction` senza problemi dall'interno di un'altra sezione critica – in questo caso, la tua funzione lascerà lo stato cancel allo stesso modo di come è stato trovato.

4.2.3 Quando usare la cancellazione di thread

In genere, è una buona idea non usare la cancellazione di thread per terminare l'esecuzione di un thread, tranne che in circostanze particolari. Durante una normale operazione, la migliore strategia è quella di comunicare al thread che dovrebbe uscire, e quindi aspettare che il thread esca da solo in maniera ordinata. Discuteremo delle tecniche per comunicare con i thread più avanti in questo capitolo e nel Capitolo 5, "Comunicazione tra processi".

4.3 Dati specifici dei thread

Diversamente dai processi, tutti i thread in un singolo programma condividono lo stesso spazio di indirizzi. Ciò significa che se un thread modifica una locazione di memoria (per esempio, una variabile globale), la modifica è visibile a tutti gli altri thread. Ciò permette a diversi thread di operare sugli stessi dati senza usare i meccanismi della comunicazione tra processi (che sono descritti nel Capitolo 5).

Ogni thread, comunque, ha la sua lista di chiamate. Ciò permette ad ogni thread di eseguire diversi pezzi di codice e di chiamare e ritornare da subroutines nel suo modo usuale. Come in un programma a thread singolo, ogni invocazione di una subroutine in ogni thread ha il proprio insieme di variabili locali, che sono memorizzate nella lista (stack) per quel trhead.

A volte, comunque, è desiderabile duplicare certe variabili in modo che ogni thread abbia una copia separata. GNU/Linux lo permette fornendo ad ogni thread un'area dei *dati sepcifici dei thread*. Le variabili memorizzante in quest'area sono duplicate per ogni thread, ed ogni thread può modificare la sua copia di una variabile senza avere effetto sugli altri thread. Poiché tutti i thread condividono lo stesso spazio di memoria, i dati specifici dei thread non possono essere accessibili usando normali variabili di riferimento. GNU/Linux fornisce funzioni speciali per assegnare e ritrovare valori dall'area di dati specifica dei thread.

Puoi creare tanti elementi di dati specifici per i thread quanti ne vuoi, ognuno di tipo `void*`. ogni elemento è referenziato da una chiave. Per creare una nuova chiave, e quindi un nuovo elemento di dati per ogni thread, usa `pthread_key_create`. Il primo argomento è un puntatore a una variabile `pthrea_key_t`. Questo valore della chiave può essere usato da

ogni thread per accedere alla propria copia dell'elemento di dati corrispondente. Il secondo argomento di `pthread_key_t` è una funzione di pulizia. Se qui passi un puntatore a funzione, GNU/Linux chiama automaticamente quella funzione quando ogni thread esce, passando il valore specifico di thread corrispondente a quella chiave. Questo è particolarmente comodo perché la funzione di pulizia è chiamata anche se il thread è cancellato ad un punto arbitrario durante la sua esecuzione. Se il valore specifico di thread è null, la funzione di pulizia dei thread non viene chiamata. Se non hai bisogno di una funzione di pulizia, puoi passare null piuttosto che un puntatore a funzione.

Dopo che hai creato una chiave, ogni thread può impostare il suo valore specifico del thread corrispondente a quella chiave, chiamando `pthread_setspecific`. Il primo argomento è la chiave ed il secondo è il valore specifico di thread `void*` da memorizzare. Per ritrovare un elemento di dati specifico di thread, chiama `pthread_getspecific`, passando la chiave come suo argomento.

Supponi, per esempio, che la tua applicazione divida un compito tra diversi thread. Per scopi di verifica, ogni thread deve avare un file di log separato, nel quale vengono registrati messaggi di esecuzione per i compiti di quel thread. L'area di dati specifica del thread è una zona conveniente per memorizzare il puntatore di file per il file di log di ogni thread individuale.

Il listato 4.7 mostra come dovresti implementarlo. La funzione `main` in questo programma di esempio crea una chiave per memorizzare il puntatore di file specifico del thread e quindi lo memorizza in `thread_log_key`. Poiché questa è una variabile globale, è condivisa da tutti i thread. Quando ogni thread comincia ad eseguire la sua funzione thread, esso apre un file di log e memorizza il puntatore a file in quella chiave. Successivamente, ognuno di questi thread può chiamare `write_to_thread_log` per scrivere un messaggio nel file di log specifico del thread. Questa funzione ritrova il puntatore di file per il file di log del thread dai dati specifici del thread e scrive il messaggio.

Listato 4.7: ***(tsd.c)*** **– Files di Log implementati per ogni thread con i dati specifici dei thread**

```
1  #include <malloc.h>
2  #include <pthread.h>
3  #include <stdio.h>
4
5  /* The key used to assocate a log file pointer with each thread.  */
6  static pthread_key_t thread_log_key;
7
8  /* Write MESSAGE to the log file for the current thread.  */
9
10 void write_to_thread_log (const char* message)
11 {
12   FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
13   fprintf (thread_log, "%s\n", message);
14 }
15
```

```
16 /* Close the log file pointer THREAD_LOG.  */
17
18 void close_thread_log (void* thread_log)
19 {
20   fclose ((FILE*) thread_log);
21 }
22
23 void* thread_function (void* args)
24 {
25   char thread_log_filename[20];
26   FILE* thread_log;
27
28   /* Generate the filename for this thread's log file.  */
29   sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
30   /* Open the log file.  */
31   thread_log = fopen (thread_log_filename, "w");
32   /* Store the file pointer in thread-specific data under thread_log_key.
   */
33   pthread_setspecific (thread_log_key, thread_log);
34
35   write_to_thread_log ("Thread starting.");
36   /* Do work here... */
37
38   return NULL;
39 }
40
41 int main ()
42 {
43   int i;
44   pthread_t threads[5];
45
46   /* Create a key to associate thread log file pointers in
47      thread-specific data.  Use close_thread_log to clean up the file
48      pointers.  */
49   pthread_key_create (&thread_log_key, close_thread_log);
50   /* Create threads to do the work.  */
51   for (i = 0; i < 5; ++i)
52     pthread_create (&(threads[i]), NULL, thread_function, NULL);
53   /* Wait for all threads to finish.  */
54   for (i = 0; i < 5; ++i)
55     pthread_join (threads[i], NULL);
56   return 0;
57 }
```

Nota che `thread_funcion` non ha bisogno di chiudere il file di log. Ciò accade perché quando la chiave file di log è stata creata, è stato specificato `close_thread_log` come funzione di pulizia per quella chiave. Ogni volta che un thread esce, GNU/Linux chiama quella funzione, passando il valore specifico di thread per la chiave log del thread. Questa funzione si preoccupa

di chiudere il file di log.

4.3.1 Gestori di pulizia

Le funzioni di pulizia per le chiavi dei dati specifiche dei thread possono risultare molto comodi per assicurare che le risorse non vengano disperse quando un thread esce o è cancellato. A volte, comunque, può risultare utile specificare delle funzioni di pulizia senza creare un nuovo elemento di dati specifico di thread che è duplicato per ogni thread. GNU/Linux fornisce per questo scopo dei *gestori di pulizia* (*cleanup handlers*).

Un gestore di pulizia è semplicemente una funzione che dovrebbe essere invocata quando un thread esce. Il gestore prende un solo parametro `void*` ed il valore del suo argomento è fornito quando il gestore è registrato – Ciò rende facile usare la stessa funzione di gestione per deallocare più istanze di risorse.

Un gestore di pulizia è una misura temporanea, usata per deallocare una risorsa solo se il thread esce o è cancellato senza aver terminato l'esecuzione di una particolare porzione di codice. In circostanze normali, quando il thread non esce e non è cancellato, le risorse dovrebbero essere deallocate esplicitamente e il gestore di pulizia dovrebbe venire rimosso.

Per registrare un gestore di pulizia, chiama `pthread_cleanup_push`, passando un puntatore alla funzione di pulizia ed il valore `void*` del suo argomento. La chiamata a `pthread_cleanup_push` deve essere bilanciata da una corrispondente chiamata a `pthread_cleanup_pop`, che termina il gestore di pulizia. Per comodità, `pthread_cleanup_pop` prende un argomento `int`; se l'argomento è diverso da zero, l'azione di pulizia è eseguita e quindi chiusa.

Il frammento di programma nel listato 4.8 mostra come dovresti usare un gestore di pulizia per assicurarti che un buffer allocato dinamicamente venga ripulito se il thread termina.

Listato 4.8: **(*cleanup.c*) – Frammento di programma che dimostra un gestore di pulizia di thread**

```
1  #include <malloc.h>
2  #include <pthread.h>
3
4  /* Allocate a temporary buffer.  */
5
6  void* allocate_buffer (size_t size)
7  {
8    return malloc (size);
9  }
10
11 /* Deallocate a temporary buffer.  */
12
13 void deallocate_buffer (void* buffer)
14 {
15   free (buffer);
16 }
17
18 void do_some_work ()
```

```
19 {
20   /* Allocate a temporary buffer.  */
21   void* temp_buffer = allocate_buffer (1024);
22   /* Register a cleanup handler for this buffer, to deallocate it in
23      case the thread exits or is cancelled.  */
24   pthread_cleanup_push (deallocate_buffer, temp_buffer);
25
26   /* Do some work here that might call pthread_exit or might be
27      cancelled...  */
28
29   /* Unregister the cleanup handler.  Since we pass a non-zero value,
30      this actually performs the cleanup by calling
31      deallocate_buffer.  */
32   pthread_cleanup_pop (1);
33 }
```

Poiché l'argomento di `pthread_cleanup_pop` in questo caso non è zero, la funzione di pulizia `deallocate_buffer` viene chiamata automaticamente qui e non ha bisogno di essere chiamata esplicitamente. In questo semplice caso, potremmo aver usato direttamente la funzione di libreria standard `free` come nostro gestore di pulizia al posto di `deallocate_buffer`.

4.3.2 Pulizia del Thread in C++

I programmatori C++ sono abituati ad ottenere la pulizia "gratuitamente" raggruppando le azioni di pulizia in oggetti distruttori. Quando gli oggetti vanno al di fuori del loro campo di operabilità, o perché un blocco è eseguito fino al completamento o perché viene lanciata un'eccezione, il C++ si assicura che i distruttori siano chiamati per quelle variabili automatiche che loro hanno. Ciò fornisce un meccanismo manuale per assicurarsi che il codice di pulizia venga chiamato, senza curarsi di come il blocco è andato in uscita.

Se un thread chiama `pthread_exit`, tuttavia, il C++ non garantisce che i distruttori siano chiamati per tutte le variabili automatiche nello stack del thread. Un modo ingegnoso per recuperare questa funzionalità è quello di invocare `pthread_exit` al livello alto della funzione thread lanciando una speciale eccezione.

Il programma nel listato 4.9 lo dimostra. Usando questa tecnica, una funzione indica la sua intenzione di uscire dal thread lanciando una `ThreadExitException` piutosto che chiamare direttamente `pthread_exit`. Poiché l'eccezione è catturata nella funzione del thread ad alto livello, tutte le variabili locali nello stack del trhread verranno distrutte appropriatamente quando l'eccezione si propaga.

Listato 4.9: (***cxx-exit.cpp***) – **Implemente l'uscita sicura dal Thread con le eccezioni C++**

```
1 #include <pthread.h>
2
3 extern bool should_exit_thread_immediately ();
4
5 class ThreadExitException
```

```
6  {
7  public:
8    /* Create an exception signalling thread exit with RETURN_VALUE.  */
9    ThreadExitException (void* return_value)
10     : thread_return_value_ (return_value)
11   {
12   }
13
14   /* Actually exit the thread, using the return value provided in the
15      constructor.  */
16   void* DoThreadExit ()
17   {
18     pthread_exit (thread_return_value_);
19   }
20
21 private:
22   /* The return value that will be used when exiting the thread.  */
23   void* thread_return_value_;
24 };
25
26 void do_some_work ()
27 {
28   while (1) {
29     /* Do some useful things here...  */
30
31     if (should_exit_thread_immediately ())
32       throw ThreadExitException (/* thread's return value = */ NULL);
33   }
34 }
35
36 void* thread_function (void*)
37 {
38   try {
39     do_some_work ();
40   }
41   catch (ThreadExitException ex) {
42     /* Some function indicated that we should exit the thread.  */
43     ex.DoThreadExit ();
44   }
45   return NULL;
46 }
```

4.4 Sincronizzazione e Sezioni Critiche

La programmazione con i thread è molto complessa perché molti programmi con i thread sono programmi concorrenti. In particolare, non c'è modo di sapere quando il sistema schedulerà un thread da eseguire e quando ne eseguirà un altro. Un thread può girare per un tempo

molto lungo, o il sistema può commutare tra i thread molto velocemente. In un sistema con più processori, il sistema può anche schedulare thread multipli da eseguire letteralmente allo stesso tempo.

Debuggare un programma con thread è difficile perché non puoi sempre e facilmente riprodurre il comportamento che ha causato il problema. Puoi eseguire il programma una volta ed avere ogni cosa che funziona bene; la prossima volta che lo esegui può andare in crash. Non c'è verso di fare in modo che il sistema scheduli i thread esattamente allo stesso modo in cui lo ha fatto prima.

La maggior causa di molti bug che riguardano i thread è che i thread cercano di accedere agli stessi dati. Come detto precedentemente, questo è uno dei più potenti aspetti dei thread, ma può anche essere pericoloso. Se un thread è solo a metà strada dell'aggiornamento di una struttura dati mentre un altro thread accede alla stessa struttura dati, è facile che ne venga fuori il caos. Spesso, programmi con thread buggati contengono codice che funzionerà solo se un thread viene schedulato più spesso – o più raramente – di un altro trhead. Questi bug sono chiamati *condizioni in competizione* (*race conditions*); I thread stanno gareggiando con altri per modificare la stessa struttura dati.

4.4.1 Condizioni in competizione

Supponi che il tuo programma abbia usa serie di lavori in coda che vengono processati da molti thread concorrenti. La coda di lavori è rappresentata da una lista collegata di oggetti `struct job`.

Dopo che ogni thread termina un'operazione, esso controlla la coda per vedere se è disponibile un altro lavoro. Se `job_queue` è diverso da null, il thread rimuove l'intestazione della lista collegata e setta `job_queue` al prossimo lavoro nella lista.

La funzione thread che processa il lavoro nella coda dovrebbe somigliare al listato 4.10

Listato 4.10: (***job-queue1.c***) – **Funzione Thread per processare un lavoro dalla coda**

```
1  #include <malloc.h>
2
3  struct job {
4    /* Link field for linked list.  */
5    struct job* next;
6
7    /* Other fields describing work to be done... */
8  };
9
10 /* A linked list of pending jobs.  */
11 struct job* job_queue;
12
13 extern void process_job (struct job*);
14
15 /* Process queued jobs until the queue is empty.  */
16
17 void* thread_function (void* arg)
18 {
```

```
19    while (job_queue != NULL) {
20      /* Get the next available job.  */
21      struct job* next_job = job_queue;
22      /* Remove this job from the list.  */
23      job_queue = job_queue->next;
24      /* Carry out the work.  */
25      process_job (next_job);
26      /* Clean up.  */
27      free (next_job);
28    }
29    return NULL;
30  }
```

Adesso supponi che a due thread accada di finire un lavoro più o meno allo stesso tempo, ma un solo lavoro rimanga in coda. Il primo thread verifica se `job_queue` è null; vedendo che non lo è, il thread entra nel loop e memorizza il puntatore all'oggetto lavoro in `netx_job`. A questo punto, succede che Linux interrompe il primo thread e schedula il secondo. Anche il secondo thread verifica `job_queue` vedendo che non è null, assegna pure lo stesso puntatore lavoro a `next_job`. Per la sfortunata coincidenza, adesso abbiamo due thread che eseguono lo stesso lavoro.

Per rendere la situazione peggiore, un thread scollegherà l'oggetto lavoro dalla coda, lasciando `job_queue` contenente `null`. Quando l'altro thread valuta `job_queue->next`, ne risulta un errore di segmentazione.

Questo è un esempio di una condizione di competizione. In circostanze "fortunate", questo particolare schedulamento dei due thread può non verificarsi mai o la condizione di competizione può non mostrarsi mai. Solo in diverse circostanze, probabilmente quando eseguito in un sistema molto caricato (o in un nuovo server multiprocessore di un cliente importante) il bug si può mostrare.

Per eliminare le condizioni di competizione è necessario un modo per fare operazioni *atomiche* (*atomic*). Un'operazione atomica è indivisibile e non interrompibile; una volta che l'operazione si avvia non può essere messa in pausa o interrotta finché non è stata completa, e nessun'altra operazione avrà luogo nel frattempo. In questo particolare esempio verificherai `job_queue`; se esso non è vuoto, rimuovi il primo lavoro, tutto come una singola operazione atomica.

4.4.2 Mutex – Mutua esclusione

La soluzione al problema della coda di lavoro in condizione di competizione è di permettere solo ad un thread per volta di accedere alla coda di lavoro. Una volta che un thread inizia a guardare la coda nessun altro thread dovrebbe essere in grado di accedervi finché un thread non ha deciso se processare un lavoro, e, se così, ha rimosso il lavoro dalla lista.

L'implementazione richiede il supporto da parte del sistema operativo. GNU/Linux fornisce i *mutexes*, abbreviazione di *MUTual EXclusion locks*. Un mutex è una speciale chiusura che un solo thread per volta ha il permesso di chiudere. Se un thread chiude un mutex e quindi anche un secondo thread cerca di chiudere lo stesso mutex, il secondo thread viene

bloccato, o messo in attesa. Solo quando il primo thread sblocca il mutex il secondo thread viene *sbloccato* – gli viene permesso di riprendere l'esecuzione. GNU/Linux garantisce che le condizioni di competizione non si verifichino tra thread che cercano di bloccare un mutex; solo un thread avrà sempre la possibilità di ottenerne la chiusura e tutti gli altri thread verranno bloccati.

Immagina un mutex come la porta del bagno. Chiunque vi arriva prima, entra e blocca la porta. Se qualcun altro cerca di entrare nel bagno mentre è occupato, quella persona troverà la porta bloccata e sarà costretto ad aspettare fuori finché chi l'ha occupato non esce.

Per creare un mutex, crea una variabile di tipo `pthread_mutex_t` e passa un puntatore ad essa a `pthread_mutex_init`. Il secondo argomento a `pthread_mutex_init` è un puntatore ad un oggetto attributo mutex, che specifica gli attributi del mutex. Come con `pthread_create`, se l'attributo puntatore è null, si assumono gli attributi di default. La variabile mutex dovrebbe essere inizializzata una sola volta. Questo frammento di codice dimostra la dichiarazione ed inizializzazione di una variabile mutex.

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Un altro semplice modo di creare un mutex con gli attributi di default è quello di inizializzarlo con lo speciale valore `PTHREAD_MUTEX_INITIALIZER`. Non è necessaria nessuna ulteriore chiamata a `pthread_mutex_init`. Ciò è particolarmente conveniente per variabili globali (e, in C++, membri di dati statici). Il precedente frammento di codice può essere scritto equivalentemente come questo:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un thread può cercare di bloccare un mutex chiamando su questo `pthread_mutex_lock`. Se il mutex non era bloccato, esso diventa bloccato e la funzione ritorna immediatamente. Se il mutex era bloccato da un altro thread, `pthread_mutex_lokc` blocca l'esecuzione e ritorna solo eventualmente quando il mutex non è bloccato dall'altro thread. Più di un thread per volta può essere bloccato su un mutex chiuso. Quando il mutex è sbloccato, solo uno dei thread bloccati (scelto in modi imprevedibile) viene sbloccato e gli è permesso di chiudere il mutex; gli altri thread stanno bloccati.

Una chiamata a `pthread_mutex_unlock` sblocca un mutex. Questa funzione dovrebbe essere sempre chiamata dallo stesso thread che ha bloccato il mutex. Il listato 4.11 mostra un'altra versione dell'esempio della coda di lavoro. Adesso la coda è protetta da un mutex. Prima di accedere alla coda (sia in scrittura che in lettura), ogni thread blocca prima un mutex. Solo quando l'intera sequenza di verifica della coda e rimozione del lavoro è completa il mutex viene sbloccato. Ciò previene la condizione di competizione precedentemente descritta.

Listato 4.11: (***job-queue2.c***) – **Funzione Thread della coda di lavoro, protetta da un Mutex**

```
1 #include <malloc.h>
2 #include <pthread.h>
3
4 struct job {
```

```
5     /* Link field for linked list.  */
6     struct job* next;
7
8     /* Other fields describing work to be done... */
9   };
10
11  /* A linked list of pending jobs.  */
12  struct job* job_queue;
13
14  extern void process_job (struct job*);
15
16  /* A mutex protecting job_queue.  */
17  pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
18
19  /* Process queued jobs until the queue is empty.  */
20
21  void* thread_function (void* arg)
22  {
23    while (1) {
24      struct job* next_job;
25
26      /* Lock the mutex on the job queue.  */
27      pthread_mutex_lock (&job_queue_mutex);
28      /* Now it's safe to check if the queue is empty.  */
29      if (job_queue == NULL)
30        next_job = NULL;
31      else {
32        /* Get the next available job.  */
33        next_job = job_queue;
34        /* Remove this job from the list.  */
35        job_queue = job_queue->next;
36      }
37      /* Unlock the mutex on the job queue, since we're done with the
38         queue for now.  */
39      pthread_mutex_unlock (&job_queue_mutex);
40
41      /* Was the queue empty?  If so, end the thread.  */
42      if (next_job == NULL)
43        break;
44
45      /* Carry out the work.  */
46      process_job (next_job);
47      /* Clean up.  */
48      free (next_job);
49    }
50    return NULL;
51  }
```

Tutti accedono a `job_queue`, il puntatore ai dati condiviso, viene tra la chiamata a `pthread_mutex_lock`

e la chiamata a `pthread_mutex_unlock`. Ad un oggetto lavoro, memorizzato in `next_job`, si accede al di fuori di questa regione solo dopo che quell'oggetto è stato rimosso dalla coda ed è quindi inaccessibile ad altri thread.

Nota che se la coda è vuota (`job_queue` è null), non usciamo immediatamente dal loop perché ciò lascerebbe il mutex permanentemente bloccato ed eviterebbe ad ogni altro thread di accedere nuovamente alla coda di lavoro. Piuttosto, ricordiamo questo fatto impostando `next_job` a null ed uscendo solo dopo aver sbloccato il mutex.

L'uso del mutex per bloccare `job_queue` non è automatico; ti permette di aggiungere il codice per bloccare il mutex prima di accedere a quella variabile e quindi sbloccarlo in seguito. Per esempio, una funzione per aggiungere un lavoro alla coda di lavoro potrebbe somigliare a questa:

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
  new_job->next = job_queue;
  job_queue = new_job;
  pthread_mutex_unlock (&job_queue_mutex);
}
```

4.4.3 Stallo del Mutex - Deadlocks

I mutex forniscono un meccanismo per permettere ad un thread di bloccare l'esecuzione di un altro. Ciò apre la possibilità di una nuova classe di bug, chiamati *deadlocks*. Uno stallo si verifica quando uno o più thread sono fermi in attesa di qualcosa che non si verifica mai. Un semplice tipo di stallo può verificarsi quando lo stesso thread cerca di bloccare un mutex due volte in una riga. Il comportamento in questo caso dipende da quale tipo di mutex viene usato. Esistono tre tipi di mutex:

- Bloccare un *mutex veloce* (*fast mutex*) (il tipo di default) causerà il verificarsi di uno stallo. Un tentativo di chiudere un mutex bloccherà finché il mutex non sarà sbloccato. Ma poiché il thread che ha chiuso il mutex è bloccato nello stesso mutex, la chiusura non può mai essere rilasciata.

- Bloccare un *mutex ricorsivo* non causa lo stallo. Un mutex ricorsivo può essere chiuso diverse volte senza problemi dallo stesso thread. Il mutex ricorda quante volte `pthread_mutex_lock` è stato chiamato su di esso dal thread che tiene la chiusura; quel thread deve fare lo stesso numero di chiamate a `pthread_mutex_unlock` prima che il mutex venga attualmente sbloccato ed un altro threadabbia il permesso di bloccarlo.

- GNU/Linux troverà e segnerà una doppia chiusura su un *errore di verifica del mutex* (*error-checking mutex*) che potrebbe altrimente causare uno stallo. La seconda chiamata consecutiva a `pthread_mutex_lock` restituisce il codice d'errore `EDEADLK`.

Per default, un mutex GNU/Linux è del tipo veloce. Per creare un mutex di uno degli altri due tipi, prima si crea un oggetto attributo mutex dichiarando una variabile `pthread_mutexattr_t`

e chiamando `pthread_mutexattr_init` su un puntatore ad essa. Quindi impostare il tipo di mutex chiamando `pthread_mutexattr_setkind_np`; il primo argomento è un puntatore all'oggetto attributo mutex, il secondo è `PTHREAD_MUTEX_RECURSIVE_NP` per un mutex ricorsivo, o `PTHREAD_MUTEX_ERRORCHECK_NP` per un mutex con controllo d'errore. Passare un puntatorea questo oggetto attributo a `pthread_mutex_destroy`.

La sequenza di codice illustra la creazione di un mutex con controllo d'errore, per esempio:

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;
pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

Come suggerito dal suffisso np, i tipi di mutex ricorsivo e con controllo d'errore sono specifici per GNU/Linux e non sono portabili. Quindi, generalmente non è consigliato usarli nei programmi (comunque, i mutex con controllo d'errore possono essere utili per debugging).

4.4.4 Verifiche non bloccanti dei Mutex

Occasionalmente, è utile verificare se un mutex è attualmente chiuso senza bloccarsi su di esso. Per esempio, un thread può aver bisogno di chiudere un mutex ma può avere altro lavoro da fare invece di bloccarsi se il mutex è già chiuso. Poiché `pthread_mutex_lock` non ritornerà finché il mutex non diventa sbloccato è necessaria qualche altra funzione.

GNU/Linux fornisce `pthread_mutex_trylock` per questo scopo. Se chiami `pthread_mutex_trylock` su un mutex non chiuso, chiuderai il mutex come se avessi chiamato `pthread_mutex_lock` e `pthread_mutex_trylock` ritornerà zero. Comunque, se il mutex è già chiuso da un altro thread, `pthread_mutex_trylock` non si bloccherà. Piuttosto, esso ritornerà immediatamente con il codice d'errore `EBUSY`. La chiusurà del mutex tenuta dall'altro thread non sarà influenzata. Potrai provare nuovamente dopo a chiudere il mutex.

4.4.5 Semafori per i Thread

Nell'esempio precedente, nel quale molti thread processano i lavori da una coda, la funzione thread principale dei thread tira fuori il prossimo lavoro finché non ci sono più lavori rimasti e quindi esce dal thread. Questo schema funziona se tutti i lavori sono accodati in anticipo o se i nuovi lavori sono accodati al più tanto velocemente quanto vengono processati dai thread. Comunque, se i thread lavorano troppo velocemente, la coda dei lavori verrà svuotata e i thread usciranno. Se successivamente vengono accodati nuovi lavori, non può rimanere nessun thread per precessarli. A noi piacerebbe piuttosto un meccanismo per bloccare i thread quando la coda è vuota, finché non tornano disponibili nuovi lavori.

Un *semaforo* fornisce un metodo conveniente per far ciò. Un semaforo è un contatore che può essere utilizzato per sincronizzare thread multipli. Come con un mutex, GNU/Linux garantisce che verificare o modificare il valore di un semaforo possa essere fatto senza problemi, senza creare condizioni di competizione.

Ogni semaforo ha un valore contatore, che è un intero non negativo. Un semaforo supporta due operazioni di base:

- un'operazione (*wait*) decrementa il valore del semaforo di 1. Se il valore è già ero, l'operazione si blocca finché il valore del semaforo non diventa positivo (dovuto ad un'azione di qualche altro thread). Quando il valore del semaforo diventa positivo, esso è decrementato di 1 e l'operazione wait ritorna.

- un'operazione *post* incrementa il valore del semaforo di 1. Se il semaforo era precedentemente zero e altri thread sono bloccati in un'operazione wait su quel semaforo, uno di questi thread viene sbloccato e la sua operazione wait si completa (ciò fa tornare il valore del semaforo nuovamente a zero)

Nota che GNU/Linux fornisce due implemntazioni di semaforo leggermente diverse. Quella che noi descriviamo qui è l'implementazione del semaforo standard POSIX. Usa questi semafori nella comunicazione tra thread. L'altra implementazione, usata per la comunicazione tra i processi, è descritta nella **??**, "**??**" . Se usi i semafori, inlcudi `<semaphore.h>`.

Sezione 5.2, Processare Semafori

Un semaforo è rappresentato da una variabile `sem_t`. Prima di usarla, devi inizializzarla usado la funzione `sem_init`, passando un puntatore alla variabile `sem_t`. Il secondo parametro dovrebbe essere zero [2], e il terzo parametro è il valore iniziale del semaforo. Se non hai più bisogno di un semaforo, è bene deallocarlo con `sem_destroy`.

Per aspettare su un semaforo, usa `sem_wait`. Per postare su un semaforo, usa `sem_post`. Viene fornita anche una funzione wait non bloccante, `sem_trywait`, è simile a `pthread_mutex_trylock` – se l'attesa potrebbe essere stata bloccata poiché il valore del semaforo era zero la funzione ritorna immediatamente, con un valore d'errore `EAGAIN`, piuttosto che bloccarsi.

GNU/Linux fornisce anche una funzione per ottenere il valore corrente di un semaforo, `sem_getvalue`, che mette il valore nella variabile `int` puntata dal suo secondo argomento. Non dovresti comunque usare il valore di un semaforo che hai ottenuto da questa funzione per prendere una decisione sul postare a aspettare in un semaforo. Far ciò potrebbe portare ad una condizione di competizione: un altro thread potrebbe cambiare il valore del semaforo tra la chiamata a `sem_getvalue` e la chiamata ad un'altra funzione semaforo. Usa piuttosto le funzioni atomiche post e wait.

Tornando al nostro esempio della coda di lavoro, possiamo usare un semaforo per contare il numero di lavori in attesa sulla coda. Il listato 4.12 controlla la coda con un semaforo. La funzione `enqueue_job` aggiunge un nuovo lavoro alla coda.

Listato 4.12: (***job-queue3.c***) – **Coda di lavoro controllata da un semaforo**

```
1 #include <malloc.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4
5 struct job {
6   /* Link field for linked list.  */
```

[2] Un valore diverso da zero indicherebbe un semaforo che può essere condiviso dai processi, che non è supportato da GNU/Linux per questo tipo di semaforo

```
7   struct job* next;
8
9   /* Other fields describing work to be done... */
10 };
11
12 /* A linked list of pending jobs. */
13 struct job* job_queue;
14
15 extern void process_job (struct job*);
16
17 /* A mutex protecting job_queue. */
18 pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
19
20 /* A semaphore counting the number of jobs in the queue. */
21 sem_t job_queue_count;
22
23 /* Perform one-time initialization of the job queue. */
24
25 void initialize_job_queue ()
26 {
27   /* The queue is initially empty. */
28   job_queue = NULL;
29   /* Initialize the semaphore which counts jobs in the queue. Its
30      initial value should be zero. */
31   sem_init (&job_queue_count, 0, 0);
32 }
33
34 /* Process queued jobs until the queue is empty. */
35
36 void* thread_function (void* arg)
37 {
38   while (1) {
39     struct job* next_job;
40
41     /* Wait on the job queue semaphore. If its value is positive,
42        indicating that the queue is not empty, decrement the count by
43        one. If the queue is empty, block until a new job is enqueued. */
44     sem_wait (&job_queue_count);
45
46     /* Lock the mutex on the job queue. */
47     pthread_mutex_lock (&job_queue_mutex);
48     /* Because of the semaphore, we know the queue is not empty. Get
49        the next available job. */
50     next_job = job_queue;
51     /* Remove this job from the list. */
52     job_queue = job_queue->next;
53     /* Unlock the mutex on the job queue, since we're done with the
54        queue for now. */
55     pthread_mutex_unlock (&job_queue_mutex);
```

```
56
57      /* Carry out the work.  */
58      process_job (next_job);
59      /* Clean up.  */
60      free (next_job);
61    }
62    return NULL;
63  }
64
65  /* Add a new job to the front of the job queue.  */
66
67  void enqueue_job (/* Pass job-specific data here...  */)
68  {
69    struct job* new_job;
70
71    /* Allocate a new job object.  */
72    new_job = (struct job*) malloc (sizeof (struct job));
73    /* Set the other fields of the job struct here...  */
74
75    /* Lock the mutex on the job queue before accessing it.  */
76    pthread_mutex_lock (&job_queue_mutex);
77    /* Place the new job at the head of the queue.  */
78    new_job->next = job_queue;
79    job_queue = new_job;
80
81    /* Post to the semaphore to indicate another job is available.  If
82       threads are blocked, waiting on the semaphore, one will become
83       unblocked so it can process the job.  */
84    sem_post (&job_queue_count);
85
86    /* Unlock the job queue mutex.  */
87    pthread_mutex_unlock (&job_queue_mutex);
88  }
```

Prima di prendere un lavoro dall'inizio della coda, ogni thread prima aspetterà al semaforo. Se il valore del semaforo è zero, che indica che la coda è vuota, il thread semplicemente si bloccherà finché il valore del semaforo non diventa positivo, indicando che un lavoro è stato aggiunto alla coda.

La funzione `enqueue_job` aggiunge un lavoro alla coda. Proprio come `thread_function`, esso ha bisogno di vedere il mutex coda prima di modificare la coda . Dopo aver aggiunto un lavoro alla coda, esso posta al semaforo, indicando che è disponibile un nuovo lavoro. Nella versione mostrata nel listato 4.12, i thread che processano i lavori non escono mai; se non ci sono lavori disponibili per un po', tutti i thread semplicemente si bloccano in `sem_wait`.

verificare traduzione di vedere il mutex coda

4.4.6 Variabili condizione

Abbiamo mostrato come usare un mutex per proteggere una variabile da accessi simultanei da due thread e come usare i semafori per implementare un contatore condiviso. Una *variabile*

condizione è un terzo dispositivo di sincronizzazione che GNU/Linux fornisce; con esso, puoi implementare condizioni più complesse sotto le quali eseguire i thread.

Supponi di aver scritto una funzione thread che esegue un loop all'infinito, facendo qualche lavoro ad ogni iterazione. Il loop thread, comunque, ha bisogno di essere controllato da un flag: il loop gira solo quando il flag è settato; quando il flag non è settato, il loop va in pausa.

Il listato 4.13 mostra come puoi implementare ciò girando in un loop. Durante ogni iterazione del loop, la funzione thread verifica che il flag sia impostato. Poiché al flag accedono più thread, esso è protetto da un mutex. Questa implementazione può essere corretta, ma non è efficiente. La funzione thread spenderà molta CPU ogni volta che il flag non è impostato, verificando e riverificando il flag, ogni volta chiudendo e aprendo il mutex. Ciò che realmente vuoi è un modo per far dormire il thread quando il flag non è impostato, finché il cambiamento di qualche circostanza non fa diventare il flag impostato.

Listato 4.13: **(*spin-condvar.c*) – Una semplice implementazione della variabile condizione**

```
1  #include <pthread.h>
2
3  extern void do_work ();
4
5  int thread_flag;
6  pthread_mutex_t thread_flag_mutex;
7
8  void initialize_flag ()
9  {
10   pthread_mutex_init (&thread_flag_mutex, NULL);
11   thread_flag = 0;
12 }
13
14 /* Calls do_work repeatedly while the thread flag is set; otherwise
15    spins.  */
16
17 void* thread_function (void* thread_arg)
18 {
19   while (1) {
20     int flag_is_set;
21
22     /* Protect the flag with a mutex lock.  */
23     pthread_mutex_lock (&thread_flag_mutex);
24     flag_is_set = thread_flag;
25     pthread_mutex_unlock (&thread_flag_mutex);
26
27     if (flag_is_set)
28       do_work ();
29     /* Else don't do anything.  Just loop again.  */
30   }
31   return NULL;
32 }
```

```
33
34 /* Sets the value of the thread flag to FLAG_VALUE.  */
35
36 void set_thread_flag (int flag_value)
37 {
38   /* Protect the flag with a mutex lock.  */
39   pthread_mutex_lock (&thread_flag_mutex);
40   thread_flag = flag_value;
41   pthread_mutex_unlock (&thread_flag_mutex);
42 }
```

Una variabile condizione ti permette di implementare una condizione sotto la quale un thread si esegue e, al contrario, la condizione sotto la quale il thread è bloccato. Finché ogni thread che potenzialmente cambia il senso della condizione usa la variabile condizione in maniera appropriata, Linux garantisce che i thread bloccati nella condizione verranno sbloccati quando la condizione cambia.

Come con un semaforo, un thread può *aspettare* su una variabile condizione. Se il thread A aspetta su una variabile condizione, esso è bloccato finché un altro thread, thread B, segnala la stessa variabile codizione. Diversamente da un semaforo, una variabile condizione non ha un contatore o memoria; il thread A deve restare in attesa sulla variabile condizione *prima* che il thread B la segnali. Se il thread B segnala la variabile condizione prima che il thread A aspetti su essa, il segnale è perso, e il thread A si blocca finché qualche altro thread non segnala la variabile condizione nuovamente.

verificare traduzione

Ciò è come dovresti usare una variabile condizione per rendere l'esempio precedente più efficiente:

- Il loop nella `thread_function` verifica il flag. Se il flag non è impostato, il thread aspetta sulla variabile condizione.

- La funzione `set_thread_flag` segnala la variabile condizione dopo aver cambiato il valore del flag. In questo modo, se `thread_function` è bloccata sulla variabile condizione, esso verrà sbloccato e la condizione verificata nuovamente.

In questo c'è un problema: c'è una condizione di competizione tra il verificare il valore del flag e segnalare o aspettare sulla variabile condizione. Supponi che `thread_function` abbia verificato il flag e visto che non era impostato. In quel momento, lo scheduler di Linux abbia messo in pausa quel thread e ripreso quello principale. Per qualche coincidenza, il thread principale è in `set_thread_flag`. Esso imposta il flag e quindi segnala la variabile condizione. Poiché nessun thread sta aspettando sulla variabile condizione in quel momnento (ricorda che `thread_function` era stata messa in pausa prima che potesse aspettare sulla variabile condizione), il segnale è perso. Adesso, quando Linux rischedula l'altro thread, inizia aspettando sulla variabile condizione e può finire per restare bloccato per sempre.

Per risolvere questo problema, abbiamo bisogno di un modo per bloccare il flag e la variabile condizione assieme con un singolo mutex. Fortunatamente, GNU/Linux fornisce esattamente questo meccanismo. Ogni variabile condizione deve essere usata in congiunzione

con un mutex, per evitare questo tipo di condizione di competizione. Usando questo schema, la funzione thread segue questi passi:

1. Il loop nella `thread_function` chiude il mutex e legge il valore del flag.
2. Se il flag è impostato, esso sblocca il mutex ed esegue la funzione lavoro.
3. Se il flag non è impostato, esso automaticamnete sblocca il mutex ed aspetta sulla variabile condizione

La caratteristica criticq qui è nel passo 3, nel quale GNU/Linux ti permette di sbloccare il mutex e aspettare sulla variabile condizione automaticamente, senza la possibilità che un altro thread intervenga. Questo elimina la possibilità che un altro thread possa cambiare il valore del flag e segnalare la variabile condizione tra la verifica del valore del flag di `thread_function` e l'attesa sulla variabile condizione.

Una variabile condizione è rappresentata da un'istanza di `pthread_cond_t`. Ricorda che ogni variabile condizione dovrebbe essere accompagnata da un mutex. Queste sono le funzioni che gestiscono le variabili condizione:

- `pthread_cond_init` inizializza una variabile condizione. Il primo argomento è un puntatore a un'istanza di `pthread_cond_t`. Il secondo argomento, un puntatore ad un oggetto attributo di una variabile condizione, è ignorato sotto GNU/Linux.
  Il mutex deve essere inizializzato separatamente, come descritto nella sottosezione 4.4.2, "Mutex – Mutua esclusione".

- `pthread_cond_signal` segnala una variabile condizione. Un singolo thread che è stato bloccato sulla variabile condizione verrà sbloccato. Se nessun altro thread è bloccato sulla variabile condizione, il segnale è ignorato. L'argomento è un puntatore all'istanza di `pthread_cond_t`
  Una chiamata simile, `pthread_cond_broadcast`, sblocca *tutti* i thread che sono bloccati sulla variabile condizione, invece di uno solo.

- `pthread_cond_wait` blocca il thread chiamante finché la variabile condizione non è segnalata. L'argomento è un puntatore all'istanza di `pthread_cond_t`. Il secondo argomento è un puntatore all'istanza di `pthread_mutex_t`.
  Quando `pthread_cond_wait` è chiamato, il mutex deve essere già chiuso dal thread chiamante. La funzione apre automaticamente il mutex e blocca sulla variabile condizione. Quando la variabile condizione è segnalata e il thread chiamante sbloccato, `pthread_cond_wait` riacquisisce automaticamente la chiusura sul mutex.

Ogni volta che il tuo programma esegue un'azione che può cambiare il senso della condizione che stai proteggendo con la variabile condizione, esso dovrebbe eseguire questi passi. (Nel nostro esempio, la condizione è lo stato del flag thread, così questi passi devono essere fatti ogni volta che il flag è cambiato).

1. Chiudere il mutex assieme alla variabile condizione.

2. Eseguire l'azione che può cambiare il senso della condizione (nel nostro esempio, impostare il flag).

3. segnalare o broadcast la variabile condizione, dipendentemente dal comportamento desiderato.

4. Aprire il mutex assieme alla variabile condizione.

Il listato 4.14 mostre nuovamente l'esempio precedente, adesso usando una variabile condizione per proteggere il flag thread. Nota che nella `thread_function`, viene tenuta una chiusura sul mutex prima di verificare il valore di `thread_flag`. Questa chiusura è rilasciata automaticamente da `pthread_cond_wait` prima di bloccare ed è automaticamente riacquisita dopo. Nota anche che `set_thread_flag` chiude il mutex prima di ipostare il valore di `thread_flag` e segnalare il mutex.

Listato 4.14: ***(condvar.c)*** **– Controllare un thread usando una variabile condizione**

```
1  #include <pthread.h>
2
3  extern void do_work ();
4
5  int thread_flag;
6  pthread_cond_t thread_flag_cv;
7  pthread_mutex_t thread_flag_mutex;
8
9  void initialize_flag ()
10 {
11   /* Initialize the mutex and condition variable.  */
12   pthread_mutex_init (&thread_flag_mutex, NULL);
13   pthread_cond_init (&thread_flag_cv, NULL);
14   /* Initialize the flag value.  */
15   thread_flag = 0;
16 }
17
18 /* Calls do_work repeatedly while the thread flag is set; blocks if
19    the flag is clear.  */
20
21 void* thread_function (void* thread_arg)
22 {
23   /* Loop infinitely.  */
24   while (1) {
25     /* Lock the mutex before accessing the flag value.  */
26     pthread_mutex_lock (&thread_flag_mutex);
27     while (!thread_flag)
28       /* The flag is clear.  Wait for a signal on the condition
29          variable, indicating the flag value has changed.  When the
30          signal arrives and this thread unblocks, loop and check the
31          flag again.  */
32       pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
```

```
33      /* When we've gotten here, we know the flag must be set.  Unlock
34         the mutex.  */
35      pthread_mutex_unlock (&thread_flag_mutex);
36      /* Do some work.  */
37      do_work ();
38    }
39    return NULL;
40  }
41
42  /* Sets the value of the thread flag to FLAG_VALUE.  */
43
44  void set_thread_flag (int flag_value)
45  {
46    /* Lock the mutex before accessing the flag value.  */
47    pthread_mutex_lock (&thread_flag_mutex);
48    /* Set the flag value, and then signal in case thread_function is
49       blocked, waiting for the flag to become set.  However,
50       thread_function can't actually check the flag until the mutex is
51       unlocked.  */
52    thread_flag = flag_value;
53    pthread_cond_signal (&thread_flag_cv);
54    /* Unlock the mutex.  */
55    pthread_mutex_unlock (&thread_flag_mutex);
56  }
```

La condizione protetta da una variabile condizione può essere arbitrariamente complessa. Comunque, prima di eseguire ogni operazione che può cambiare il senso della condizione, dovrebbe essere richiesta una chiusura del mutex, e la variabile condizione dovrebbe essere segnalata successivamente.

Una variabile condizione può anche essere usata senza una condizione, semplicemente come un meccanismo per bloccare un thread finché un altro tread non "lo sveglia". Può anche essere usato un semaforo per questo scopo. La differenza principale è che un semaforo "ricorda" la chiamata wake-up (risveglio) anche se nessun thread era bloccato su di esso in quel momento, mentre una variabile condizione scarta la chiamata wake-up finché qualche thread è attualmente bloccato su di esso al momento. Inoltre, un semaforo smista solo un singolo wake-up per post; con `pthread_cond_broadcast`, può essere risvegliato un numero sconosciuto e arbitrario di thread allo stesso tempo.

4.4.7 Deadlocks (stalli) con due o più thread

Gli stalli possono verificarsi quando due o più thread sono bloccati, nell'attesa che si verifichi una condizione che solo l'altro può causare. Per esempio, se il thread A è bloccato su una variabile condizione in attesa che il thread B la segnali, e il thread B è bloccato su una variabile condizione in attesa che il thread A la segnali, si è verificato un deaklock perché nessun thread segnalerà mai l'altro. Dovresti fare attenzione ed evitare la possibilità di situazioni del genere perché sono un po' difficili da individuare.

Un errore comune che può causare un deadlock riguarda un problema nel quale più di un thread cerca di bloccare lo stesso insieme di oggetti. Per esempio, considera un programma nel quale due diversi thread, che eseguono due diverse funzioni thread, hanno bisogno di chiudere gli stessi due mutex. Supponi che il thread A chiuda il mutex 1 e quindi il mutex 2, e che sia successo che il thread B abbia chiuso il mutex 2 prima del mutex 1. In uno scenario di scheduling sufficientemente sfortunato, Linux può schedulare A abbastanza a lungo per chiudere il mutex 1 e quindi schedulare il thread B che prontamente chiude il mutex 2. Adesso nessun thread può andare avanti erché ognuno è bloccato in un mutex che l'altro thread tiene chiuso.

Questo è un esempio di un problema di deadlock più generale, che può includere non solo la sincronizzazione di oggetti come mutex, ma anche altre risorse, come blocco di files o dispositivi. Il problema si verifica quando thread multipli cercano di chiudere lo stesso insieme di risorse in ordini diversi. La soluzione è quella di assicurarsi che tutti i thread che chiudono più di una risorsa le chiudano nello stesso ordine.

4.5 Implementazione dei Thread in GNU/Linux

L'implementazione dei thread POSIX su GNU/Linux differisce dall'implementazione dei thread su molti altri sistemi UNIX-like in maniera importante: su GNU/Linux, i thread sono implementati come processi. Ogni volta che chiami `pthread_create` per creare un nuovo thread, Linux crea un nuovo processo che esegue quel thread. Comunque, questo processo non è lo stesso di un processo che creeresti con `fork`; in particolare, esso condivide lo stesso spazio di indirizzi e le stesse risorse come il processo originale piuttosto che riceverne delle copie.

Il programma `thread-pid` mostrato nel listato 4.15 lo dimostra. Il programma crea un threa; sia il thread originale che quello nuovo chiamano la funzione `getpid` e stampano i loro rispettivi ID di processo e quindi girano all'infinito.

Listato 4.15: ***(thread-pid)*** **– Stampa l'ID di processo per i Thread**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  void* thread_function (void* arg)
6  {
7    fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
8    /* Spin forever.  */
9    while (1);
10   return NULL;
11 }
12
13 int main ()
14 {
15   pthread_t thread;
16   fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
17   pthread_create (&thread, NULL, &thread_function, NULL);
```

```
18    /* Spin forever.  */
19    while (1);
20    return 0;
21  }
```

Esegui il programma in background e invoca quindi `ps x` per visualizzare i tuoi processi in esecuzione. Non dimenticare di uccidere successivamente il programma `thread-pid` – esso consuma molta CPU e non fa nulla. Ecco a cosa dovrebbe somigliare l'output

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
  PID  TTY      STAT TIME COMMAND
14042 pts/9    S    0:00 bash
14608 pts/9    R    0:01 ./thread-pid
14609 pts/9    S    0:00 ./thread-pid
14610 pts/9    R    0:01 ./thread-pid
14611 pts/9    R    0:00 ps x
% kill 14608
[1]+ Terminated ./thread-pid
```

Notifica del controllo dei lavori nella Shell

Le righe che iniziano con `[1]` sono della shelle. Quando esegui un programma in background, la shell gli assegna un numero di lavoro – in questo caso, 1 – e stampa il pid del programma. Se un lavoro in background termina, la shell riporta questo fatto la prossima volta che invochi un comando.

Nota che ci sono tre processi che stanno eseguendo il programma `thread-pid`. Il primo di questi, con pid 14608, è il thread principale del programma; il terzo, con pid 14610, è il thread che abbiamo creato per eseguire `thread_function`.

Riguardo al secondo thread, con pid 14609? Questo è il "thread manager", che è parte dell'implementazione interna dei thread di GNU/Linux. Il thread manager è creato la prima volta che un programma chiama `pthread_create` per creare un nuovo thread.

4.5.1 Gestione dei segnali

Supponi che un programma multithread riceva un segnale. In quale thread è invocato il gestore del segnale? Il comportamento dell'interazione tra segnali e trhead varia da un sistema UNIX-like a un'altro. In GNU/Linux, il comportamento è dettato dal fatto che i thread sono implementati come processi.

Poiché ogni thread è un processo separato e poiché un segnale è inviato ad un particolare processo, non c'è ambiguità su quale thread riceve il segnale. Tipicamente, i segnali inviati dall'esterno del programma sono inviati al processo corrispondente al thread principale del

programma. Per esempio, se un programma fa il fork ed il processo figlio esegue un programma multithread, il processo padre terrà l'id di processo del thread principale del programma del processo figlio e userà quell'id di processo per inviare segnali al suo figlio. Questa è generalmente una buona convenzione che dovresti seguire tu stesso quando invii segnali ad un programma multithread.

Nota che quest'aspetto dell'implementazione GNU/Linux de pthread è una variante con il thread standard POSIX. Non fare affidamento a questo comportamento in programmi che si intendono essere portabili.

All'interno di un programma multithread, è possibile per un thread inviare un segnale specificatamente ad un altro thread. Usa la funzione `pthread_kill` per farlo. Il suo primo parametro è un ID di thread e il suo secondo parametro è un numero di segnale.

4.5.2 La chiamata di sistema *clone*

Sebbene i thread GNU/Linux creati nello stesso programma sono implementati come processi separati, essi condividono il loro spazio virtuale di memoria ed altre risorse. Un processo figlio creato con `fork`, comunque, ottiene copie di questi elementi. Com'è creato il tipo di forma dei processi? La chiamata di sistema Linux `clone` è una forma generalizzata di `fork` e `pthread_create` che permette al chiamante di specificare quali risorse sono condivise tra il processo chiamante e il nuovo processo creato. Inoltre, `clone` richiede che venga specificata l'area di memoria per l'esecuzione dello stack che i nuovi processi useranno. Comunque, menzioniamo `clone` qui per soddisfare la curiosità del lettore. Questa chiamata di sistema non dovrebbe essere ordinariamente usata nei programmi. Usa `fork` per creare nuovi processi o `pthread_create` per creare thread.

4.6 Processi Vs. Thread

Per alcuni programmi che traggono benefici dalla concorrenza, la decisione se usare i processi o i thread può essere difficile. Qui ci sono alcune linee guida per aiutarti a decidere quale modello di concorrenza si addice meglio al tuo programma:

- Tutti i thread in un programma devono eseguire lo stesso eseguibile. Un processo figlio, d'altro canto, può eseguire un diverso eseguibile chiamando la funzione `exec`.

- Un thread che sbaglia può danneggiare altri thread nello stesso processo poiché i thread condividono lo stesso spazio di memoria ed altre risorse. Per esempio, una scrittura in un'area di memoria tramite un puntatore non inizializzato in un thread può corrompere la memoria visibile ad un altro thread.
  Un processo che sbaglia, d'altro canto, non può farlo perché ogni processo ha una copia dello spazio di memoria del programma.

- Copiare la memoria per un nuovo processo aggiunge sovraccarico addizionale alle prestazioni, relativo al creare un nuovo thread. Comunque, la compia è fatta solo quando la memoria viene modificata, così la penalità è minima se il processo figlio soltanto legge la memoria.

- I thread dovrebbero essere usati per programmi che hanno bisogno di parallelismo *a grana fine*. Per esempio, se un problema può essere rotto in più thread con compiti pressoché identici, potrebbe essere una buona scelta. I processi dovrebbero essere usati per programmi che hanno bisogno di un parallelismo meno preciso.
- Condividere i dati tra i thread è inutile perché i thread condividono la stessa memoria. (Comunque, si deve dare più attenzione per evitare le condizioni di competizione, come descritto precedentemente). Condividere i dati tra i processi richiede l'uso di meccanismi IPC, come descritto nel capitolo 5. Questo può essere più lento ma permette a più processi di soffrire di meno del bug della concorrenza.

Capitolo 5

Comunicazione tra processi

Il Capitolo 3, "Processi", ha parlato della creazione dei processi e mostrato come un processo può ottenere lo stato di uscita di un processo figlio. Questa è la forma più semplice di comunicazione tra due processi, ma non è la più potente. Il meccanismo del capitolo 3 non fornisce nessun modo al padre di comunicare con il figlio, tranne che per mezzo degli argomenti della linea di comando e le variabili d'ambiente, nessun altro modo per il figlio di comunicare con il padre, ad eccezione dello stato di uscita del figlio. Nessuno di questi meccanismi fornisce una maniera per comunicare con il processo figlio mentre questo è in esecuzione, questi meccanismi non permettono neppure la comunicazione con un processo al di fuori della relazione padre-figlio.

Questo capitolo descrive il significato della comunicazione tra processi che aggira queste limitazioni. Presenteremo vari modi per comunicare tra padri e figli, tra processi "non collegati" e anche tra processi su macchine diverse.

La comunicazione tra processi – *Interprocess communication (IPC)* – è il trasferimento di dati tra processi. Per esempio, un browser web può richiedere una pagina web da un server web, che quindi invia i dati HTML. Questo trasferimento di dati di solito usa i socket in una connessione telefonica. In un altro esempio, puòi voler stampare i nomi dei files di una directory usando un comando come `ls | lpr`. La shell crea un processo `ls` ed un processo separato `lpr`, collegando i due con una pipe, rappresentata dal simbolo "`|`". Una pipe permette una via di comunicazione tra due processi correlati. Il processo `ls` scrive dati nella pipe e il processo `lpr` legge i dati dalla pipe.

In questo capitolo, discutiamo cinque tipi di comunicazione tra processi:

- La memoria condivisa permette ai processi di comunicare semplicemente leggendo e scrivendo su una specifica locazione di memoria.
- La memoria mappata è simile alla memoria condivisa, ad eccezione che essa è associata ad un file nel filesystem.
- Le pipe permettono la comunicazione sequenziale da un processo ad un processo correlato.

- Le FIFO sono simili alle pipe, ad eccezione che i processi non correlati possono comunicare perché alla pipe viene dato un nome nel file system.

- I socket supportano la comunicazioni tra processi non correlati anche in computer diversi.

Questi tipi di IPC differiscono per i seguenti criteri:

- Se essi estendono la comunicazione a processi correlati (processi con un antenato comune), a processi non correlati condividendo lo stesso filesystem o a ogni computer connesso ad una rete.

- Se un processo comunicante è limitato solamente a scrivere dati o a leggere dati.

- Il numero di processi a cui è permesso di comunicare.

- se i processi comunicanti sono sincronizzati da IPC – per esempio, un processo che legge si ferma finché non sono disponibili dati da leggere.

In questo capitolo, omettiamo la discussione sugli IPC che permettono la comunicazione solo un numero limitato di volte, come la comunicazione tramite il valore di uscita del figlio.

5.1 Memoria condivisa

Uno dei metodi più semplici per la comunicazione tra processi è usando la memoria condivisa. La memoria condivisa permette a due o più processi di accedere alla stessa memoria come se avessero tutti chiamato `malloc` e ottenuto puntatori alla stessa memoria attuale. Quando un processo cambia la memoria, tutti gli altri processi vedono la modifica.

5.1.1 Comunicazione locale veloce

La memoria condivisa è la forma più veloce di comunicazione tra processi perché tutti i processi condividono lo stesso pezzo di memoria. Accedere a questa memoria condivisa è tanto veloce quanto accedere alla memoria non condivisa del processo, e non richiede chiamate di sistema o voci del khernel. Evita anche di copiare dati non necessariamente.

Poiché il kernel non sincronizza gli accessi alla memoria condivisa, devi fornire la tua sincronizzazione. Per esempio, un processo non dovrebbe leggere dalla memoria fin dopo che i dati non vi siano stati scritti, e due processi non dovrebbero scrivere sulla stessa locazione di memoria allo stesso tempo. Una strategia comune per evitare queste condizioni di competizione è quella di usare i semafori, che sono discussi nella prossima sezione. I nostri programmi illustrativi, comunque, mostrano giusto un singolo processo che accede alla memoria, per focalizzare sui meccanismi della memoria condivisa ed evitrare di confondere il codice di esempio con la logica di sincronizzazione.

5.1.2 Il modello della memoria

Per usare un segmento di memoria condiviso, un processo deve allocare il segmento. Quindi ogni processo che desidera accedere al segmento deve attaccarsi al segmento. Dopo aver finito il loro uso del segmento, ogni processo si stacca dal segmento. A questo punto, un processo deve deallocare il segmento.

Comprendere il modello della memoria di Linux aiuta a spiegare l'allocazione e l'attaccamento dei processi. Su Linux, ogni memoria virtuale del processo è divisa in pagine. Ogni processo mantiene una mappa dai suoi indirizzi di memoria a queste pagine di memoria virtuali, che attualmente contengono i dati. Anche se ogni processo ha i suoi indirizzi propri, la mappatura di più processi può puntare alla stessa pagina, permettendo di condividere la memoria. Delle pagine di memoria si parla nella **??**, "**??**" del **??**, "**??**". Allocare un nuovo segmento di memoria condivisa causa la creazione di una pagina di memoria virtuale. Poiché tutti i processi desiderano accedere allo stesso segmento di memoria condivisa, solo un processo dovrebbe allocare un nuovo segmento condiviso. Allocare un segmento esistente non crea nuove pagine, ma restituisce un identificatore per la pagina esistente. Per permettere ad un processo di usare un segmento di memoria condivisa, un processo lo collega, che aggiunge voci di mappatura dalla sua memoria virtuale alla pagina del segmento condiviso. Quando finito con il segmento, queste voci di mappatura vengono rimosse. Quando non ci sono più processi che vogliono accedere a questi segmenti di memoria condivisa, esattamente un processo deve deallocare le pagine di memoria virtuale. Tutti i segmenti di memoria condivisa sono allocati come multipli interi della *dimensione di pagina* del sistema, che è il numero di byte in una pagina di memoria. Su sistemi Linux, la dimensione della pagina è di 4KB, ma si può ottenere questo valore chiamando la funzione `getpagesize`

Sezione 8.8, titolo... del capitolo 8, titolo...

verificare traduzione

5.1.3 Allocazione

Un processo alloca un segmento di memoria condivisa usando `shmget` ("SHared Memory GET"). Il suo primo parametro è una chiave intera che specifica quale segmento creare. Processi non correlati possono accedere allo stesso segmento condiviso specificando lo stesso valore chiave. Sfortunatamente, altri processi possono anche avere scelto la stessa chiave fissata, che può andare in conflitto. Usando la speciale costante `IPC_PRIVATE` come valore chiave garantische che venga creato un nuovo segmento di memoria.

Il suo secondo parametro specifica il numero di byte nel segmento. Poiché i segmenti sono allocati usando le pagine, il numero di byte attualmente allocati è arrotondato superiormente all'intero multiplo della dimensione della pagina. Il terzo parametro è l'or bit a bit dei valori flag che specificano le opzioni di `shmget`. I valori flag includono questi:

- `IPC_CREAT` – Questo flag indica che dovrebbe essere creato un nuovo segmento. Ciò permette di chreare un nuovo segmento quando si specifica un valore chiave.

- `IPC_EXCL` – Questo flag, che è sempre usato con `IPC_CREAT`, causa a `shmget` di fallire se è specificata una chiave di segmento che già esiste. Quindi, questo permette al processo chiamante di avere un segmento "esclusivo". Se questo flag non è stato dato e la chiave

di un segmento esistente è usata, `shmget` ritorna il segmento esistente invece di crearne uno nuovo.

- Flag modo – Questo valore è fatto da 9 biti, indica il permesso dato al proprietario, gruppo e il resto di controllare l'accesso al segmento. I bit di esecuzione sono ignorati. Un modo semplice per specificare i pemessi è di usare le costanti definite in `<sys/stat.h>` e documentati nella sezione `2 stat` delle pagine man.[1]. Per esempio, `S_IRUSR` e `S_IWUSR` specificano permessi di lettura e scrittura per il proprietario del segmento di memoria condivisa, e `S_IROTH` e `S_IWOTH` specificano permessi di lettura e scrittura per gli altri.

Per esempio, questa chiamata di `shmget` crea un nuovo segmento di memoria condiviso (o accede ad uno esistente, se `shm_key` è già usato) che è leggibile e scrivibile per il proprietario ma non per altri utenti.

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR | S_IWUSER);
```

Se la chiamata ha successo, `shmget` ritorna un identificatore di segmento. Se il segmento di memoria esiste già, vengono verificati i permessi di accesso e viene fatta una verifica per assicurarsi che il segmento non sia marcato per la distruzione.

5.1.4 Attacco e distacco

Per rendere il segmento di memoria condivisa disponibile, un processo deve usare `shmat`, "SHared Memory ATtach". Passargli l'identificatore del segmento di memoria condivisa `SHMID` restituito da `shmget`. Il secondo argomento è un puntatore che specifica dove nel tuo spazio di indirizzi del processo vuoi mappare la memoria condivisa; se specifichi NULL, Linux sceglierà un indirizzo disponibile. Il terzo argomento è un flag, che può includere il seguente:

- `SHM_RND` indica che l'indirizzo specificato per il secondo parametro dovrebbe essere arrotondato inferiormente ad un multiplo della dimensione della pagina. Se non specifichi questo flag, dovrai allineare tu stesso il secondo argomento di `shmat` alla dimensione della pagina.

- `SHM_RDONLY` indica che il segmento sarà solo letto, non scritto

Se la chiamata ha successo, essa restituisce l'indirizzo del segmento condiviso attaccato. I figli creati dalla chiamata `fork` ereditano i segmenti attaccati; essi possono staccare il segmento di memoria condivisa, se vogliono.

Quando hai finito con un segmento di memoria condivisa, il segmento dovrebbe essere staccato usando `shmdt` ("SHared Memory DeTach"). Passagli l'indirizzo restituito da `shmat`. Se il segmento è stato deallocato e questo era l'ultimo processo che lo stava usando, viene rimosso. Le chiamate a `exit` ed ogni altra della famiglia `exec` staccano automaticamente i segmenti.

[1] Questi bit di permesso sono gli stessi di quelli usati per i files. Essi sono descritti nella **??**, "**??**"

5.1.5 Controllare e deallocare la memoria condivisa

La chiamata `shmctl` ("SHared Memory ConTroL") restituisce informazioni su un segmento di memoria condiviso e può modificarlo. Il primo parametro è un identificatore di segmento di memoria condivisa.

Per ottenere informazioni su un segmento di memoria condivisa, passa `IPC_STAT` come secondo argomento ed un puntatore ad una `struct shmid_ds`.

Per rimuovere un segmento, passa `IPC_RMID` come secondo argomento, e passa NULL come terzo argomento. Il segmento viene rimosso quando l'ultimo processo che vi è attaccato lo ha infine staccato.

Ogni segmento di memoria condivisa dovrebbe essere deallocato esplicitamente usando `shmclt` quando hai finito con questo, Per evitare la violazione di limiti di tutto il sistema sul numero totale di segmenti di memoria condivisa. Invocando `exit` e `exec` vengono staccati i segmenti di memoria, ma non vengono deallocati.

Vedi la pagina man di `shmctl` per una descrizione di altre operazione che puoi fare sui segmenti di memoria condivisa.

5.1.6 Un programma di esempio

Il programma nel listato 5.1 illustra l'uso della memoria condivisa

Listato 5.1: ***(shm.c)*** **– Esercizio Memoria condivisa**

```
1  #include <stdio.h>
2  #include <sys/shm.h>
3  #include <sys/stat.h>
4
5  int main ()
6  {
7    int segment_id;
8    char* shared_memory;
9    struct shmid_ds shmbuffer;
10   int segment_size;
11   const int shared_segment_size = 0x6400;
12
13   /* Allocate a shared memory segment.  */
14   segment_id = shmget (IPC_PRIVATE, shared_segment_size,
15                        IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
16
17   /* Attach the shared memory segment.  */
18   shared_memory = (char*) shmat (segment_id, 0, 0);
19   printf ("shared memory attached at address %p\n", shared_memory);
20   /* Determine the segment's size.  */
21   shmctl (segment_id, IPC_STAT, &shmbuffer);
22   segment_size = shmbuffer.shm_segsz;
23   printf ("segment size: %d\n", segment_size);
24   /* Write a string to the shared memory segment.  */
25   sprintf (shared_memory, "Hello, world.");
```

```
26    /* Deatch the shared memory segment. */
27    shmdt (shared_memory);
28
29    /* Reattach the shared memory segment, at a different address. */
30    shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
31    printf ("shared memory reattached at address %p\n", shared_memory);
32    /* Print out the string from shared memory. */
33    printf ("%s\n", shared_memory);
34    /* Detach the shared memory segment. */
35    shmdt (shared_memory);
36
37    /* Deallocate the shared memory segment. */
38    shmctl (segment_id, IPC_RMID, 0);
39
40    return 0;
41  }
```

5.1.7 Debugging

Il comando `ipcs` fornisce informazioni sulle possibilità di comunicazione tra processi, inclusi i segmenti condivisi. Usa il flag `-m` per ottenere informazioni sulla memoria condivisa. Per esempio, questo codice mostra che è in uso un segmento di memoria condivisa, numero 1627649:

```
% ipcs -m
------ Shared Memory Segments --------
key              shmid    owner    perms    bytes    nattch   status
0x00000000       1627649  user     640      25600    0
```

Se un segmento di memoria è stato erroneamente lasciato da un programma, puoi usare il comando `ipcrm` per rimuoverlo.

```
% ipcrm shm 1627649
```

5.1.8 Pro e contro

I segmenti di memoria condivisa permettono una comunicazione bidirezionale veloce tra qualsiasi numero di processi. Ogni utente può sia leggere che scrivere, ma un programma deve stabilire e seguire qualche protocollo per prevenire le condizioni di competizione come la sovrascrittura di informazioni prima che queste vengano lette. Sfortunatamente, Linux non garantisce strettamente l'accesso esclusivo anche se crei un nuovo segmento condiviso con `IPC_PRIVATE`.

Inoltre, affinché più processi possano usare un segmento condiviso, essi devono cercare di arrangiarsi ad usare la stessa chiave.