Programmazione avanzata su Linux

Traduzione italiana dell'opera
Advanced Unix Programming
di
Mark Mitchell, Jeffrey Oldham, Alex Samuel

Simone Sollena

18 ottobre 2012

Indice

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*. Un deadlock, 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 sarà sospeso 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 effettuare lo stesso numero di chiamate a `pthread_mutex_unlock` affinché che il mutex venga attualmente sbloccato ed un altro thread abbia 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, bisogna prima creare 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 puntatore a questo oggetto attributo a `pthread_mutex_destroy` per creare un mutex di questo tipo, e quindi distruggi l'oggetto attributo con `pthread_mutexattr_destroy`

Questa 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] 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

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