第20章　高级线程编程

本章从线程的基本概念入手，介绍Linux的高级编程内容。

20.1　线程的概念和用途

线程通常叫做轻型的进程。虽然这个叫法有些简单化，但这有利于了解线程的概念。线程和UNIX系统中的进程十分接近，要了解这两者之间的区别，我们应该看一下 UNIX系统中的进程和Mach的任务和线程之间的关系。在UNIX系统中，一个进程包括一个可执行的程序和一系列的资源，例如文件描述符表和地址空间。在 Mach中，一个任务仅包括一系列的资源；线程处理所有的可执行代码。一个 Mach的任务可以有任意数目的线程和它相关，同时每个线程必须和某个任务相关。和某一个给定的任务相关的所有线程都共享任务的资源。这样，一个线程就是一个程序计数器、一个堆栈和一系列的寄存器。所有需要使用的数据结构都属于任务。一个UNIX系统中的进程在Mach中对应于一个任务和一个单独的线程。

因为线程和进程比起来很小，所以相对来说，线程花费更少的 CPU资源。进程往往需要它们自己的资源，但线程之间可以共享资源，所以线程更加节省内存。 Mach的线程使得程序员可以编写并发运行的程序，而这些程序既可以运行在单处理器的机器上，也可以运行在多处理器的机器中。另外，在单处理器环境中，当应用程序执行容易引起阻塞和延迟的操作时，线程可以提高效率。

20.2　一个简单的例子

用子函数pthread_create创建一个新的线程。它有四个参数：一个用来保存线程的线程变量、一个线程属性、当线程执行时要调用的函数和一个此函数的参数。例如：

```
pthread_t           a_thread;
pthread_attr_t      a_thread_attribute;
void                thread_function(void *argument);
char                *some_argument;

pthread_create( &a_thread, a_thread_attribute, (void *)&thread_function,
        (void *) &some_argument);
```

线程属性只指明了需要使用的最小的堆栈大小。在以后的程序中，线程的属性可以指定其他的值，但现在大部分的程序可以使用缺省值。不像 UNIX系统中使用 fork系统调用创建的进程，它们和它们的父进程使用同一个执行点，线程使用在 pthread_create中的参数指明要开始执行的函数。

现在我们可以编制第一个程序了。我们编制一个多线程的应用程序，在标准输出中打印“ Hello World ”。首先我们需要两个线程变量，一个新线程开始执行时可以调用的函数。我们还需要指明每一个线程应该打印的信息。一个做法是把要打印的字符串分开，给每一个线程一个字符串作为开始的参数。请看下面的代码：

```
void print_message_function( void *ptr );
```

```
main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello";
  char *message2 = "World";

  pthread_create( &thread1, pthread_attr_default,
            (void*)&print_message_function, (void*) message1);
  pthread_create(&thread2, pthread_attr_default,
            (void*)&print_message_function, (void*) message2);

  exit(0);
}

void print_message_function( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s ", message);
}
```

程序通过调用pthread_create创建第一个线程，并将“ Hello ”作为它的启动参数。第二个线程的参数是“ World ”。当第一个线程开始执行时，它使用参数“ Hello ”执行函数print_message_function。它在标准输出中打印“ Hello ”，然后结束对函数的调用。线程当离开它的初始化函数时就将终止，所以第一个线程在打印完“ Hello ”后终止。当第二个线程执行时，它打印“ World ”然后终止。但这个程序有两个主要的缺陷。

首先也是最重要的是线程是同时执行的。这样就无法保证第一个线程先执行打印语句。所以你很可能在屏幕上看到“ World Hello ”，而不是“ Hello World ”。请注意对exit的调用是父线程在主程序中使用的。这样，如果父线程在两个子线程调用打印语句之前调用 exit，那么将不会有任何的打印输出。这是因为 exit函数将会退出进程，同时释放任务，所以结束了所有的线程。任何线程(不论是父线程或者子线程)调用exit 都会终止所有其他线程。如果希望线程分别终止，可以使用pthread_exit函数。

我们可以使用一个办法弥补此缺陷。我们可以在父线程中插入一个延迟程序，给子线程足够的时间完成打印的调用。同样，在调用第二个之前也插入一个延迟程序保证第一个线程在第二个线程执行之前完成任务。

```
void print_message_function( void *ptr );

main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello ";
  char *message2 = "World";

  pthread_create( &thread1, pthread_attr_default,
            (void *) &print_message_function, (void *) message1);
  sleep(10);
  pthread_create(&thread2, pthread_attr_default,
            (void *) &print_message_function, (void *) message2);
```

```
    sleep(10);
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s", message);
    pthread_exit(0);
}
```

这样是否达到了我们的要求了呢？不尽如此，因为依靠时间的延迟执行同步是不可靠的。这里遇到的情形和一个分布程序和共享资源的情形一样。共享的资源是标准的输出设备，分布计算的程序是三个线程。

其实这里还有另外一个错误。函数 sleep和函数exit一样和进程有关。当线程调用 sleep时，整个的进程都处于睡眠状态，也就是说，所有的三个线程都进入睡眠状态。这样我们实际上没有解决任何的问题。希望使一个线程睡眠的函数是 pthread_delay_np。例如让一个线程睡眠2秒钟，用如下程序：

```
    struct timespec delay;
    delay.tv_sec = 2;
    delay.tv_nsec = 0;
    pthread_delay_np( &delay );
```

20.3 线程同步

POSIX提供两种线程同步的方法，mutex和条件变量。mutex是一种简单的加锁的方法来控制对共享资源的存取。我们可以创建一个读 /写程序，它们共用一个共享缓冲区，使用 mutex来控制对缓冲区的存取。

```
void reader_function(void);
void writer_function(void);

 char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;
struct timespec delay;

main()
{
    pthread_t reader;

    delay.tv_sec = 2;
    delay.tv_nsec = 0;

    pthread_mutex_init(&mutex, pthread_mutexattr_default);
    pthread_create( &reader, pthread_attr_default, (void*)&reader_function,
                NULL);
    writer_function();
}
```

```
void writer_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 0 )
      {
          buffer = make_new_item();
          buffer_has_item = 1;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}

void reader_function(void)
{
  while(1)
  {
      pthread_mutex_lock( &mutex );
      if ( buffer_has_item == 1)
      {
          consume_item( buffer );
          buffer_has_item = 0;
      }
      pthread_mutex_unlock( &mutex );
      pthread_delay_np( &delay );
  }
}
```

在上面的程序中，我们假定缓冲区只能保存一条信息，这样缓冲区只有两个状态，有一条信息或者没有信息。使用延迟是为了避免一个线程永远占有 mutex。

但mutex的缺点在于它只有两个状态，锁定和非锁定。 POSIX的条件变量通过允许线程阻塞和等待另一个线程的信号方法，从而弥补了 mutex的不足。当接受到一个信号时，阻塞线程将会被唤起，并试图获得相关的mutex的锁。

20.4 使用信号量协调程序

我们可以使用信号量重新看一下上面的读 /写程序。涉及信号量的操作是 semaphore_up、semaphore_down、semaphore_init、semaphore_destroy和 semaphore_decrement。所有这些操作都只有一个参数，一个指向信号量目标的指针。

```
void reader_function(void);
void writer_function(void);

char buffer;
Semaphore writers_turn;
Semaphore readers_turn;

main()
```

```
{
    pthread_t reader;

    semaphore_init( &readers_turn );
    semaphore_init( &writers_turn );

    /* writer must go first */
    semaphore_down( &readers_turn );

    pthread_create( &reader, pthread_attr_default,
                (void *)&reader_function, NULL);
    writer_function();
}

void writer_function(void)
{
    while(1)
    {
        semaphore_down( &writers_turn );
        buffer = make_new_item();
        semaphore_up( &readers_turn );
    }
}

void reader_function(void)
{
    while(1)
    {
        semaphore_down( &readers_turn );
        consume_item( buffer );
        semaphore_up( &writers_turn );
    }
}
```

这个例子也没有完全地利用一般信号量的所有函数。我们可以使用信号量重新编写“Hello world”的程序：

```
void print_message_function( void *ptr );

Semaphore child_counter;
Semaphore worlds_turn;

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World";

    semaphore_init( &child_counter );
    semaphore_init( &worlds_turn );

    semaphore_down( &worlds_turn ); /* world goes second */
```

```
    semaphore_decrement( &child_counter ); /* value now 0 */
    semaphore_decrement( &child_counter ); /* value now -1 */
    /*
     * child_counter now must be up-ed 2 times for a thread blocked on it
     * to be released
     *
     */

    pthread_create( &thread1, pthread_attr_default,
            (void *) &print_message_function, (void *) message1);

    semaphore_down( &worlds_turn );

    pthread_create(&thread2, pthread_attr_default,
            (void *) &print_message_function, (void *) message2);

    semaphore_down( &child_counter );

    /* not really necessary to destroy since we are exiting anyway */
    semaphore_destroy ( &child_counter );
    semaphore_destroy ( &worlds_turn );
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
    fflush(stdout);
    semaphore_up( &worlds_turn );
    semaphore_up( &child_counter );
    pthread_exit(0);
}
```

信号量child_counter用来强迫父线程阻塞，直到两个子线程执行完 printf语句和其后的semaphore_up(&child_counter)语句才继续执行。

20.5 信号量的实现

20.5.1 Semaphore.h

```
#ifndef SEMAPHORES
#define SEMAPHORES

#include
#include


typedef struct Semaphore
```

```
{
    int         v;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}
Semaphore;


int         semaphore_down (Semaphore * s);
int         semaphore_decrement (Semaphore * s);
int         semaphore_up (Semaphore * s);
void        semaphore_destroy (Semaphore * s);
void        semaphore_init (Semaphore * s);
int         semaphore_value (Semaphore * s);
int         tw_pthread_cond_signal (pthread_cond_t * c);
int         tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m);
int         tw_pthread_mutex_unlock (pthread_mutex_t * m);
int         tw_pthread_mutex_lock (pthread_mutex_t * m);
void        do_error (char *msg);

#endif
```

20.5.2 Semaphore.c

```
#include "semaphore.h"

/*
 * function must be called prior to semaphore use.
 *
 */
void
semaphore_init (Semaphore * s)
{
    s->v = 1;
    if (pthread_mutex_init (&(s->mutex), pthread_mutexattr_default) == -1)
     do_error ("Error setting up semaphore mutex");

    if (pthread_cond_init (&(s->cond), pthread_condattr_default) == -1)
     do_error ("Error setting up semaphore condition signal");
}

/*
 * function should be called when there is no longer a need for
 * the semaphore.
 *
 */
void
semaphore_destroy (Semaphore * s)
{
    if (pthread_mutex_destroy (&(s->mutex)) == -1)
     do_error ("Error destroying semaphore mutex");
```

```
    if (pthread_cond_destroy (&(s->cond)) == -1)
     do_error ("Error destroying semaphore condition signal");
}

/*
 * function increments the semaphore and signals any threads that
 * are blocked waiting a change in the semaphore.
 *
 */
int
semaphore_up (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));

    (s->v)++;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));
    tw_pthread_cond_signal (&(s->cond));

    return (value_after_op);
}

/*
 * function decrements the semaphore and blocks if the semaphore is
 * <= 0 until another thread signals a change.
 *
 */
int
semaphore_down (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    while (s->v <= 0)
    {
     tw_pthread_cond_wait (&(s->cond), &(s->mutex));
    }

    (s->v)--;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/*
```

```
 * function does NOT block but simply decrements the semaphore.
 * should not be used instead of down -- only for programs where
 * multiple threads must up on a semaphore before another thread
 * can go down, i.e., allows programmer to set the semaphore to
 * a negative value prior to using it for synchronization.
 *
 */
int
semaphore_decrement (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    s->v--;
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/*
 * function returns the value of the semaphore at the time the
 * critical section is accessed.  obviously the value is not guarenteed
 * after the function unlocks the critical section.  provided only
 * for casual debugging, a better approach is for the programmar to
 * protect one semaphore with another and then check its value.
 * an alternative is to simply record the value returned by semaphore_up
 * or semaphore_down.
 *
 */
int
semaphore_value (Semaphore * s)
{
    /* not for sync */
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}

/* ------------------------------------------------------------------- */
/* The following functions replace standard library functions in that  */
/* they exit on any error returned from the system calls.  Saves us    */
/* from having to check each and every call above.                     */
/* ------------------------------------------------------------------- */

int
tw_pthread_mutex_unlock (pthread_mutex_t * m)
```

```
{
    int         return_value;
    if ((return_value = pthread_mutex_unlock (m)) == -1)
     do_error ("pthread_mutex_unlock");
    return (return_value);
}
int
tw_pthread_mutex_lock (pthread_mutex_t * m)
{
    int         return_value;
    if ((return_value = pthread_mutex_lock (m)) == -1)
     do_error ("pthread_mutex_lock");
    return (return_value);
}
int
tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m)
{
    int         return_value;
    if ((return_value = pthread_cond_wait (c, m)) == -1)
     do_error ("pthread_cond_wait");
    return (return_value);
}
int
tw_pthread_cond_signal (pthread_cond_t * c)
{
    int         return_value;
    if ((return_value = pthread_cond_signal (c)) == -1)
     do_error ("pthread_cond_signal");
    return (return_value);
}
/*
 * function just prints an error message and exits
 *
 */
void
do_error (char *msg)
{
    perror (msg);
    exit (1);
}
```