第八章： 线程

目标：
本章旨在向学员介绍linux操作系统下线程的使用：
1）了解linux系统下线程与进程的区别
2）掌握线程相关的编程方法
3）掌握线程间通信同步的机制

时间：3.5 学时

教学方法：讲授PPT、实例练习

8.1 什么是线程？

定义

一个程序中的多个执行路线就叫线程(thread)
线程是一个进程内部的控制序列,进程至少有一个执行线程

不同

调用fork创建的进程拥有自己的变量和PID，时间调度也独立，进程中创建线程时，新的线程拥有自己的栈，与它的创建者共享全局变量、文件描述符、信号句柄等资源

特点

线程执行开销小，但不利于资源的管理和保护

8.2 线程

- pthread_create()函数

创建一个新线程，类似于创建新进程的fork函数

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

参数thread：线程创建时，这个指针指向变量中被写入一个标识符，标识符来引用新线程

参数attr：用于设置线程的属性

参数start_routine：指定线程将要执行的函数

参数arg：要执行函数传递的参数

8.2 线程

- 线程终止函数pthread_exit

```
#include <pthread.h>

void pthread_exit(void *retval);
```

8.2 线程

- 收集线程函数pthread_join

作用等价于进程中用来收集子进程信息的wait函数。

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

参数th：指定将要等待的线程

参数thread_return：指向线程的返回值

8.2 线程

实验：简单的线程程序

```
#include  <pthread.h>
void *thread_function(void *arg);
char message[] = “Hello World”;
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL,
thread_function,(void*)message);
    if (res != 0) {
        perror(“Thread creation failed”);
        exit(EXIT_FAILURE);
    }
    printf(“Waiting for thread to finish…\n”);
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
perror(“Thread join filed”);
        exit(EXIT_FAILURE);
    }
    printf(“Thread joined, it returned %s\n”, (char
*)thread_result);
    printf(“Message is now %s\n”, message);
    exit(EXIT_SUCCESS);
  }
  void *thread_function(void *arg) {
    printf(“thread_function is running. Argument
was %s\n”, (char *)arg);
    sleep(3);
    strcpy(message, “Bye!”);
    pthread_exit(“Thank you for the CPU time”);
}
```

8.2 线程

- 练习：编写一个程序，至少创建2个线程，两个线程都循环执行，一个线程每隔1秒输出我是线程1，另一个线程每隔1秒输出我是线程2

8.3 线程的同步机制

- 信号量
  与进程间通信机制类似，但仅用于线程间同步操作过程中

- 互斥量
  信号量的另一种应用，线程同步机制的一种，某个线程先取得资源后，后访问资源的线程会被阻塞

8.3.1 线程信号量

- 信号量创建函数sem_init

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

参数sem：初始化的信号量对象
参数pshared：控制信号量的类型(0:线程之间共享.>0:进程之间共享)
参数value ： 指定信号量的初始值

8.3.1 线程信号量

- 信号量控制函数sem_wait和sem_post

```
#include <semaphore.h>

int sem_wait(sem_t * sem); 从信号量的值减去一个“1”
int sem_post(sem_t * sem);
```

sem_wait():从信号量的值减去一个“1”，但它永远会先等待该信号量为一个非零值才开始做减法。

sem_post():给信号量的值加上一个“1”

参数sem：指向sem_init初始化的信号量的指针参数

8.3.1 线程信号量

- 信号量清理函数sem_destroy

```
#include <semaphore.h>

int sem_destroy(sem_t * sem);
```

8.3.1 线程信号量

练习：线程信号量

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
sem_t sem;
void *thread_function(void *arg);
void sendsem(){
    sem_post(&sem);
}
char message[] = "Hello World";
int main(){
   int res;
   pthread_t a_thread;
   void *thread_result;
   sem_init(&sem ,0,0);
   signal(SIGINT,sendsem);
   res = pthread_create(&a_thread, NULL,
          thread_function, (void *)message);
```

```
if (res !=0)    {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);    }
 printf("Waiting for SEM from SIGNAL...\n");
 res = pthread_join(a_thread, &thread_result);
 if (res !=0)
{
    perror( “Thread join failed” );
    exit(EXIT_FAILURE);    }
    printf( “Thread joined\n” );
    exit(EXIT_FAILURE);
}
void *thread_function(void *arg)
{ sem_wait(&sem);
  printf( “thread_function is running. Argument
was %s\n” , (char *)arg);
  sleep(1);
  pthread_exit(NULL);
}
```

8.3.2 线程互斥量

- 初始化互斥量

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
```

互斥函数使用的方法与信号量类似

8.3.2 线程互斥量

- 控制互斥量函数pthread_mutex_lock和pthread_mutex_unlock

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

8.3.2 线程互斥量

- 互斥量清理函数pthread_mutex_destroy

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

8.3.2 线程互斥量

练习：线程互斥量

```
int count=0;
pthread_mutex_t count_lock =
PTHREAD_MUTEX_INITIALIZER;
void *thread_function(void *arg);
char *message1="I'm thread 1";
char *message2="I'm thread 2";
int main()  {
   int res;
   pthread_t a_thread;
   pthread_t b_thread;
   void *thread_result;
   res = pthread_create(&a_thread, NULL,
                    thread_function,  (void *)message1);
if (res !=0)    {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);    }
    res = pthread_create(&b_thread, NULL,
                     thread_function, (void *)message2);
if (res !=0)    {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);    }
    printf("Waiting for thread to finish...\n");
     res = pthread_join(a_thread, &thread_result);
```

```
if (res !=0)    {
  perror("Thread join failed");
  exit(EXIT_FAILURE);    }
  printf("Waiting for thread to finish...\n");
  res = pthread_join(b_thread,
                                    &thread_result);
  if (res !=0)    {
     perror("Thread join failed");
     exit(EXIT_FAILURE);    }
     printf("Thread joined\n");
     exit(EXIT_FAILURE);
 }
 void *thread_function(void *arg)
 { printf("thread_function is running. Argument
was %s\n", (char *)arg);
  pthread_mutex_lock(&count_lock);
  count++;
  sleep(1);
  printf("count is %d\n",count);
  pthread_mutex_unlock(&count_lock);
  pthread_exit(NULL);
 }
```

8.3.3 线程控制

- 线程初始化属性函数pthread_attr_init

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

8.3.3 线程控制

- 线程属性修改函数，设置线程为独立线程

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- attr参数：输出线程属性，在pthread_create被调用
- detachstate参数：PTHREAD_CREATE_DETACHED，使线程成为独立线程，不需要主线程调用pthread_join进行子线程的资源回收。

8.3.3 线程控制

练习：设置脱离状态属性

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void *thread_function(void *arg);
char message[] = “Hello World”;
int thread_finished = 0;
int main() {
    int res;
    pthread_t a_thread;
    pthread_attr_t thread_attr;
    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
      perror(“Attribute creation failed”);
      exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
    if (res !=0) {
      perror(“Setting detached attribute failed”);
      exit(EXIT_FAILURE);
     }
```

```
res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
    if (res !=0) {
      perror(“Thread creation failed”);
      exit(EXIT_FAILURE);
    }
    pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
      printf(“Waiting for thread to say it’s finished…\n”);
      sleep(1);
   }
   printf(“Other thread finished, bye!\n”);
   exit(EXIT_SUCCESS);
 }
 void *thread_function(void *arg) {
     printf(“thread_function is running. Argument was %s\n”,
(char *)arg);
     sleep(4);
     printf(“Second thread setting finished flag, and exiting
now\n”);
    thread_finished = 1;
    pthread_exit(NULL);
  }
```