Threads

张海宁

贵州大学

hnzhang1@gzu.edu.cn

May 16, 2018

Overview

Threads

What is a Threads

Threads

我们知道可以通过多个进程来执行多任务，但是考虑到进程的资源占用等问题，本章讲解线程来实现多任务操作。
线程是一个程序的多个不同的执行部分，它们共享全局变量。

线程和进程

1. 进程是操作系统进行资源（包括 cpu、内存、磁盘 IO 等）分配的最小单位。
2. 线程是 cpu 调度的基本单位。
3. 我们打开的微信，浏览器都是一个进程。进程可能有多个子任务，比如微信要接收消息，发送消息，这些子任务就是线程。资源分配给进程，线程共享进程资源。
4. 线程的调度比进程的调度所需要的系统开销要小得多。

How to create threads

线程相关的函数 I

函数原型

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
  void *(*start_routine)(void *), void *arg);
```

1. 第一个参数是事先声明的一个 pthread_t 类型的指针，新创建的线程通过这个指针来引用。
2. 第二个参数用来设置线程的属性，一般情况下可以设置为 NULL。
3. 第三个参数是本线程要执行的具体方法（函数）的指针。
4. 最后一个参数是本线程执行第三个参数指定的方法时，需要使用的参数。
5. 如果创建线程成功会返回 0 。

线程相关的函数 II

函数原型

```
#include <pthread.h>
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);
```

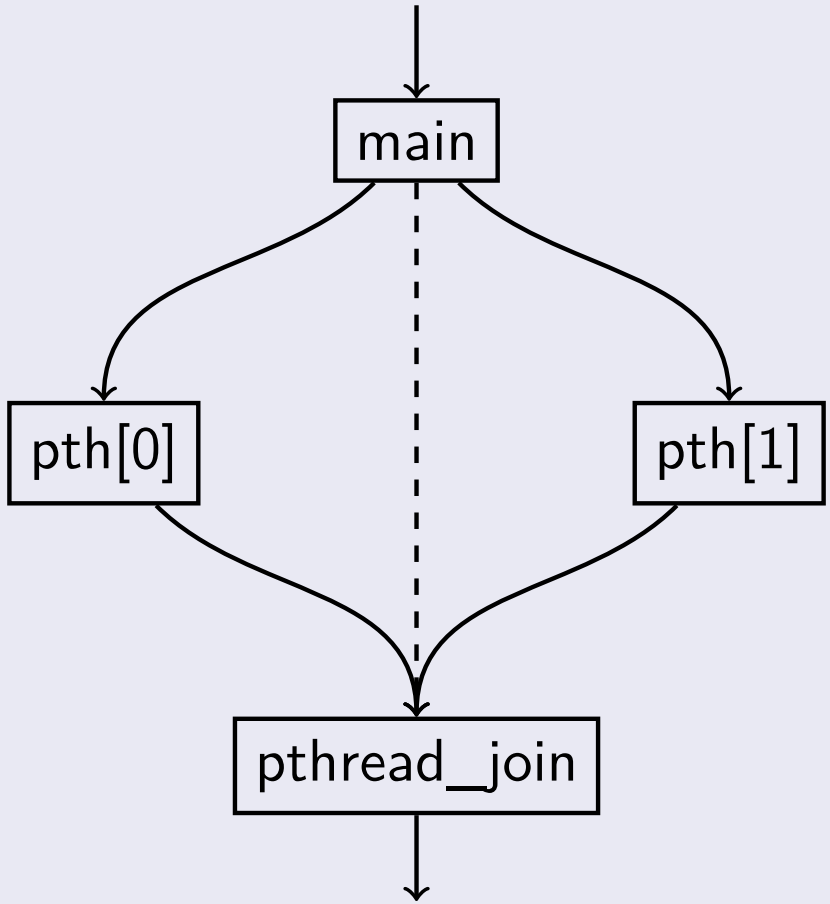
1. pthread_exit 会结束调用它的线程。
2. pthread_join 是主程序用来等待子线程结束的方法，和进程里的 wait 类似。
   1. 第一个参数用来指定是哪个线程。
   2. 第二个参数用来存储线程的返回值。
   3. 如果成功会返回 0 。

多线程例子

```
void *fmt(void * g);
int main(){
 pthread_t  pth[2];  void * rz;
 char  ch='a';  char  chA='A';
 int ra=pthread_create(&pth[0],NULL,fmt,&ch);
 int rA=pthread_create(&pth[1],NULL,fmt,&chA);
 pthread_join(pth[0],&rz);
 pthread_join(pth[1],&rz);
 printf("two threads has finished.\n");
}
void *fmt(void * arg){
 char c=*(char *)arg;  int i=0;
 for(char a=c;i<3;a++,i++){
  printf("%c\n",a); sleep(1);}
 return NULL;
 //pthread_exit("a thread finished.\n");}
```

执行过程

执行示意图



Question: 如果没有 pthread_join 会如何执行？

Synchronization

为什么需要线程同步

上一个多线程的例子是两个线程各自使用自己的参数去执行一段代码。但是如果两个线程去操作一个共同资源的时候采取上一个多线程类似的代码就会出现问题。
这个时候就需要线程同步，以免多个线程去争抢同一个资源，在这里线程同步的概念就是要让线程有个先来后到进行排队，同一个资源同一时刻只能由一个线程进行操作。

Linux 线程同步的方法

线程同步有多种方法，此处介绍两种基本的方法：

1. semaphore
   act as gatekeepers around a piece of code
2. mutex
   act as a mutual exclusion (hence the name mutex) device to protect sections of code

这两种方式都可实现类似的功能，但是也存在哪一种方法更适用的问题。请回答以下情况中，哪种机制更适合：

1. 修改一个共享变量
2. 总共有 5 个资源，比如客服电话线路，一个线程想使用其中一个

semaphore

1. Dijkstra, a Dutch computer scientist, first conceived the concept of semaphores.
2. A semaphore is a special type of variable that can be incremented or decremented, but crucial access to the variable is guaranteed to be atomic, even in a multithreaded program.
   This means that if two (or more) threads in a program attempt to change the value of a semaphore, the system guarantees that all the operations will in fact take place in sequence. With normal variables the result of conflicting operations from different threads within the same program is undefined.

semaphore functions I

The semaphore functions do not start with *pthread_*, as most thread-specific functions do, but with *sem_*.
A semaphore is created with the sem_init function:

create a semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

1. initializes a semaphore object pointed to by sem
2. The pshared parameter controls the type of semaphore. If 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes.
3. the initial integer value of the semaphore
4. returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.

semaphore functions II

A pair of functions controls the value of the semaphore and is declared as follows:

control the value of a semaphore

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

1. The sem_post function atomically increases the value of the semaphore by 1.
2. The sem_wait function atomically decreases the value of the semaphore by 1, but always waits until the semaphore has a nonzero count first.
3. return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and errno is set to indicate the error.

semaphore functions III

This function tidies up the semaphore when you have finished with it. It is declared as follows:

destroy a semaphore

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

1. this function takes a pointer to a semaphore and tidies up any resources that it may have. If you attempt to destroy a semaphore for which some thread is waiting, you will get an error.
2. returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.

生产者消费者 (threadPC.c) 示例 I

生产者消费者函数

```
void * consumer(void * args){
 while(1){
  sem_wait(&sem);
  printf("consumer: the  buf is:%s.\n",buf);
 }
 pthread_exit(NULL);
}
void * productor(void * args){
 while(1){
  printf("productor: input something:\n");
  fgets(buf,100,stdin);
  sem_post(&sem);
 }
 pthread_exit(NULL);
}
```

生产者消费者示例 II

主程序

```
sem_t sem; char * buf;
int main(){
 pthread_t thC;
 pthread_t thP;
 void * th_rz;
 buf = (char *) malloc(100);
 res = sem_init(&sem,0,0);
 res=pthread_create(&thC,NULL,consumer,NULL);
 res=pthread_create(&thP,NULL,productor,NULL);
 pthread_join(thC,&th_rz);
 pthread_join(thP,&th_rz);
 sem_destroy(&sem);
 exit(EXIT_SUCCESS);
}
```

生产者消费者示例 III

不理想和理想的生产者消费者程序

1. 我们的生产者会一直提示用户输入数据（写入 buf），不管消费者是否取走了 buf 里的数据。
2. 理想的生产者应该先查看 buf 里的数据是否被取走或者是 buf 未满，然后再生产。（作业）

mutex

The other way of synchronizing access in multithreaded programs is with mutexes (short for mutual exclusions), which act by allowing the programmer to "lock" an object so that only one thread can access it. To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

mutex functions

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

mutexattr 参数是用来控制 mutex 的属性的，此处先设置为 NULL，有兴趣的同学可以自行学习。这些函数如果操作成功会返回 0 。

多窗口售票 (threadMutex.c) 程序 I

售票函数

```
void * sellTicket(void *args){
 int i= *(int *)args;
 while(1){
  pthread_mutex_lock(&mut);
  if(ticket<1){
   pthread_mutex_unlock(&mut);
   break;
  }
  printf("thread %d is selling ticket %d.\n",i,ticket);
  ticket--;
  pthread_mutex_unlock(&mut);
  sleep(1);
 }
 pthread_exit(NULL);
}
```

多窗口售票 (threadMutex.c) 程序 II

主程序

```
int ticket=10;
int main(){
 int res; pthread_t th[3]; void * th_rz;
 res = pthread_mutex_init(&mut,NULL);
 int arg=0;
 res=pthread_create(&th[0],NULL,sellTicket,&arg);
 int arg1=1;
 res=pthread_create(&th[1],NULL,sellTicket,&arg1);
 int arg2=2;
 res=pthread_create(&th[2],NULL,sellTicket,&arg2);
 pthread_join(th[0],&th_rz);
 pthread_join(th[1],&th_rz); pthread_join(th[2],&th_rz);
 pthread_mutex_destroy(&mut);
 exit(EXIT_SUCCESS);
}
```

多窗口售票 (threadMutex.c) 程序 III

运行结果

```
$ gcc -o threadMutex threadMutex.c -lpthread
$ ./threadMutex
thread 0 is selling ticket 10.
thread 1 is selling ticket 9.
thread 2 is selling ticket 8.
thread 0 is selling ticket 7.
thread 1 is selling ticket 6.
thread 2 is selling ticket 5.
thread 2 is selling ticket 4.
thread 1 is selling ticket 3.
thread 0 is selling ticket 2.
thread 2 is selling ticket 1.
```

作业

编写一个程序，要求：

1. 创建两个线程：一个生产者线程，一个消费者线程
2. 当缓冲区为空时，生产者线程提示用户输入一串字符，然后将用户输入的字符写入缓冲区
3. 当缓冲区非空时，消费者线程从缓冲出取出所有字符
4. 当用户输入 end 时，程序结束

The End

Appendix

本课程相关资源下载

1. ppt
   https://github.com/gmsft/ppt/tree/master/linux
2. 实验指导书
   https://github.com/gmsft/ppt/tree/master/book/linux

about man page

The manual is generally split into eight numbered sections, organized as follows (on Research Unix, BSD, macOS and Linux):

section	description
1	General commands
2	System calls
3	Library function(C standard library)
4	Special files(devices) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

Table: man page

在终端中运行 man read 与 man 2 read ， 观察其输出的区别。

detached threads

前面的多线程程序中，主线程都调用了 pthread_join 方法来等待线程结束。在主线程需要子线程返回结果的时候，这种方法是必要的。但是在主线程不需要子线程的返回结果时，就没必要这样做了，只需要让子线程自己去运行，然后自行结束就可以了，这样的线程就被称为 *detached threads*。

在任何一个时间点上，线程是可结合的（joinable）或者是分离的（detached）。一个可结合的线程能够被其他线程收回其资源和杀死。在被其他线程回收之前，它的存储器资源（例如栈）是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的，它的存储器资源在它终止时由系统自动释放。

detached threads 示例

```
pthread_attr_t att;
pthread_attr_init(&att);
pthread_attr_setdetachstate(&att,PTHREAD_CREATE_DETACHED);
char  ch='a';
int ra=pthread_create(&pth[0],&att,fmt,&ch);
```