Operating System

Lab06 Introduction

目录

线程的基本操作函数1/5

- 四个基本线程函数，头文件均为
  #include<pthread.h>
  - 创建线程函数
  - 等待线程的结束函数
  - 获取自己线程ID函数
  - 终止线程函数

编译程序：**gcc example.c –o example –lpthread**

线程的基本操作函数2/5

- 创建线程函数

int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, (void *)(*start_rtn)(void *), void *arg);

其中，第一个参数为指向线程标识符的指针

第二个参数用来设置线程属性，通常为空

第三个参数是线程运行函数的起始地址

第四个参数是运行函数的参数

返回值：若线程创建成功，则返回0；若线程创建失败，则返回出错编号。

线程的基本操作函数3/5

- 等待线程结束函数

```
int pthread_join(pthread_ttid,void**status);
```

其中，第一个参数为被等待的线程标识符

第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值

说明：这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。

线程的基本操作函数4/5

- 获取自己线程ID函数

```
pthread_t  pthread_self(void);
```

说明：pthread_t的类型为unsigned long int，所以在打印的时候要使用%lu方式，否则显示结果出问题。

线程的基本操作函数5/5

- 终止线程函数

一个线程的结束有两种途径，一种是函数结束，调用它的线程也就结束，另一种方式是通过函数pthread_exit来实现。

void pthread_exit(void *status);

唯一的参数是函数的返回代码

注意：一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread_join的线程则返回错误代码ESRCH。

目录

简单的多线程编程1/3

- 例：example1.c

```
#include <pthread.h>
#include <stdio.h>

void* thread_func(void *arg)
{
    printf("thread id=%lu\n", pthread_self());
    return arg;
}
```

简单的多线程编程2/3

```
int main(void)
{
    pid_t pid;
    pthread_t tid;
    pid = getpid();
    printf("process id=%d\n", pid);
    pthread_create(&tid, NULL, thread_func, NULL);
    pthread_join(tid,NULL);
    return 0;
}
```

简单的多线程编程3/3

- 编译

```
gcc example1.c -o example1 -lpthread
```

- 运行

```
./example1
```

- 运行结果

```
zhenz@zhenz-virtual-machine:~/thread$ ./example1
process id=2679
thread id=3076057920
```

目录

线程应用中的同步互斥问题1/12

- 互斥锁

  互斥锁用来保证一段时间内只有一个线程在执行一段代码。

- 互斥锁的声明

  pthread_mutex_t mutex;

- 互斥锁的初始化

  pthread_mutex_init(&mutex,NULL);

线程应用中的同步互斥问题2/12

- 加锁

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

- 解锁

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

说明：加锁段的代码同一时间只能被一个线程调用执行。当一个线程执行到pthread_mutex_lock处时，如果此时另一个线程使用该锁，则将阻塞此线程，即程序将等待到另一个线程释放此互斥锁。

线程应用中的同步互斥问题 3/12

- 注意：在使用互斥锁的过程中很有可能会出现死锁，即两个线程试图同时占用两个资源，并按不同的次序锁定相应的互斥锁。此时我们可以使用函数pthread_mutex_trylock，它是函数pthread_mutex_lock的非阻塞版本，当它发现死锁不可避免时,它会返回相应的信息，程序员可以针对死锁做出相应的处理。
- 互斥锁销毁

```
pthread_mutex_destroy(phtread_mutex_t *
        lock);
```

线程应用中的同步互斥问题 4/12

- 读写锁

读写锁是从互斥锁中发展下来的，互斥锁会将试图访问我们定义的保护区的所有进程都阻塞掉，但读写锁与此不同，它会将访问中的读操作和写操作区分开来对待，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。

在某些读数据比改数据频繁的应用中，读写锁将会比互斥锁表现出很大的优越性

线程应用中的同步互斥问题 5/12

- 读写锁所遵循的规则：
  - 只要没有线程持有某个给定的读写锁用于写，那么任意数目的线程都可持有该读写锁用于读
  - 仅当没有线程持有某个给定的读写锁用于读或写，才能分配该读写锁用于写。

线程应用中的同步互斥问题 6/12

- 读写锁的声明

```
pthread_rwlock_t  rwlock;
```

- 读写锁的初始化

```
pthread_rwlock_init(pthread_rwlock_t
*rwlock, pthread_rwlockattr_t *attr);
```

例： pthread_rwlock_init (&rwlock,NULL);

线程应用中的同步互斥问题 7/12

- 为读进程获得锁

  pthread_rwlock_rdlock(pthread_rwlock_t *lock);

- 为写进程获得锁

  pthread_rwlock_wrlock(pthread_rwlock_t *lock);

- 读写进程解锁

  pthread_rwlock_unlock(pthread_rwlock_t *lock);

- 销毁读写锁

  pthread_rwlock_destroy(pthread_rwlock_t *lock);

线程应用中的同步互斥问题 8/12

条件变量

- 条件变量是用来等待而不是用来上锁的。它是通过一种能够挂起当前正在执行的进程或放弃当前进程直到在共享数据上的一些条件得以满足。
- 条件变量操作过程是：首先通知条件变量，然后等待，同时挂起当前进程直到有另外一个进程通知该条件变量为止。

互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。

线程应用中的同步互斥问题 9/12

- 工作机制

条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

线程应用中的同步互斥问题 10/12

- 条件变量的声明

  pthread_cond_t condtion;

- 条件变量的初始化

  pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t *attr);

- 等待信号

  pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);

- 发送信号

  pthread_cond_signal(pthread_cond_t *cond);

线程应用中的同步互斥问题 11/12

- 信号量

信号量(semaphore)是另一种加锁操作，与普通加锁不同的是，信号量记录了一个空闲资源数值，信号量的值表示了当前空闲资源的多少。信号量本质上是一个非负的整数计数器，它用来控制对公共资源的访问。

线程应用中的同步互斥问题12/12

- 当公共资源增加时，调用函数sem_post()增加信号量
- 当信号量值大于0时，才能使用公共资源，使用后，函数sem_wait()减少信号量
- 当它变为0时，进程将主动放弃处理器进入等待对列

THE END…

THANK YOU~