Linux 多线程编程

linux 多线程设计是指基于 Linux 操作系统下的多线程设计，包括多任务程序的设计，并发程序设计,网络程序设计，数据共享等。
Linux 系统下的多线程遵循 POSIX 线程接口，称为 pthread。

一. 什么是线程？
线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

二. 什么时候使用多线程？
当多个任务可以并行执行时，可以为每个任务启动一个线程。

三. 线程的创建及等待：
（1）使用 pthread_create 函数创建。

```
#include<pthread.h>
int pthread_create (pthread_t *__restrict __newthread,//新创建的线程ID
                    __const pthread_attr_t *__restrict __attr,//线程属性
                    void *(*__start_routine) (void *),//新创建的线程从start_routine开始执行
                    void *__restrict __arg)//执行函数的参数
```

返回值：成功-0，失败-返回错误编号，可以用 strerror(errno)函数得到错误信息。
（2）使用 pthread_join 函数等待指定线程结束
int pthread_join(pthread_t thread, void **value_ptr);
参数:
thread 一个有效的线程 id；

value_ptr 接收线程返回值的指针；

注：调用此函数的线程在指定的线程退出前将处于挂起状态或出现错误而直接返回，如果 value_ptr 非 NULL 则 value_ptr 指向线程返回值的指针，函数成功后指定的线程使用的资源将被释放。

四. 线程的终止：三种方式

1) 线程从启动例程中返回，返回值是线程的退出码；
2) 线程调用了 pthread_exit 函数；这里不是调用 exit，因为线程调用 exit 函数，会导致线程所在的进程退出。
3) 线程可以被同一进程中的其他线程取消。

一个小例子：

启动两个线程，一个线程对全局变量 num 执行加 1 操作，执行五百次；一个线程对全局变量执行减 1 操作，同样执行五百次。

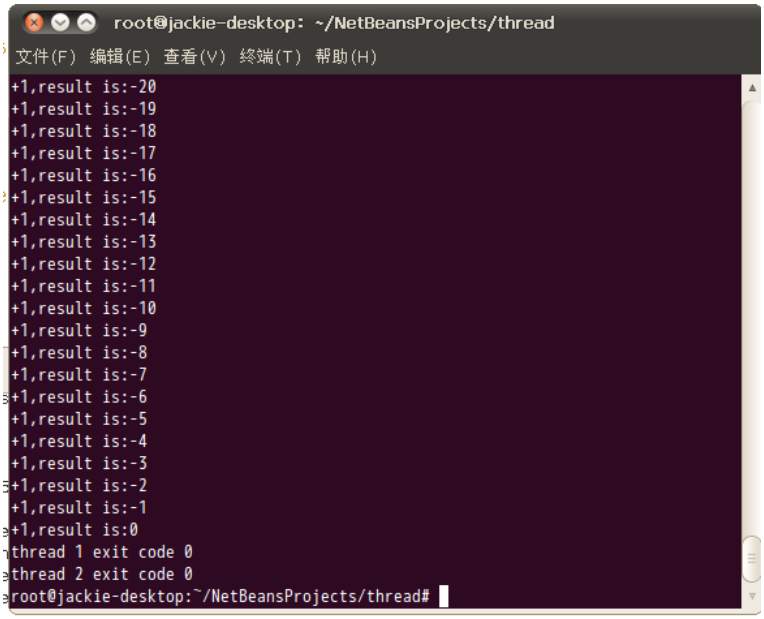
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

int num=0;
void *add(void *arg) {//线程执行函数，执行 500 次加法
   int i = 0,tmp;
   for (; i <500; i++)
   {
      tmp=num+1;
      num=tmp;
      printf("add+1,result is:%d\n",num);
```

```
    }
    return ((void *)0);
}

void *sub(void *arg)//线程执行函数，执行 500 次减法
{
    int i=0,tmp;
    for(;i<500;i++)
    {
        tmp=num-1;
        num=tmp;
        printf("sub-1,result is:%d\n",num);
    }
    return ((void *)0);
}

int main(int argc, char** argv) {

    pthread_t tid1,tid2;
    int err;
    void *tret;
    err=pthread_create(&tid1,NULL,add,NULL);//创建线程
    if(err!=0)
    {
        printf("pthread_create error:%s\n",strerror(err));
        exit(-1);
    }
    err=pthread_create(&tid2,NULL,sub,NULL);
    if(err!=0)
```

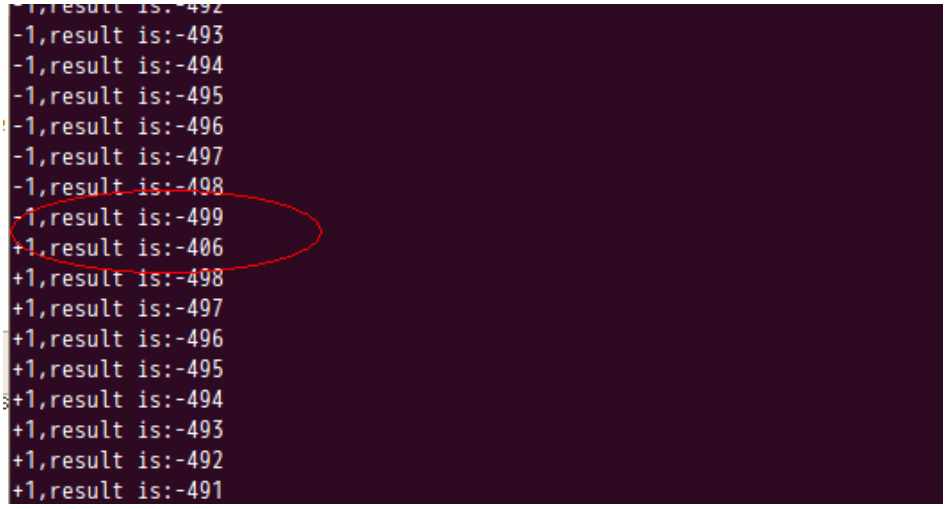
```
    {
        printf("pthread_create error:%s\n",strerror(err));
         exit(-1);
    }
    err=pthread_join(tid1,&tret);//阻塞等待线程 id 为 tid1 的线
程，直到该线程退出
    if(err!=0)
    {
        printf("can not join with thread1:%s\n",strerror(err));
        exit(-1);
    }
    printf("thread 1 exit code %d\n",(int)tret);
    err=pthread_join(tid2,&tret);
    if(err!=0)
    {
        printf("can not join with thread1:%s\n",strerror(err));
        exit(-1);
    }
    printf("thread 2 exit code %d\n",(int)tret);
    return 0;
}
```

使用 gcc 编译该文件（gcc main.c -o main -lpthread)。

执行结果：



乍一看，结果是对的，加 500 次，减 500 次，最后结果为 0。但是仔细看所有的输出，你会发现有异样的东西。



导致这个不和谐出现的原因是，两个线程可以对同一变量进行修改。假如线程 1 执行 tmp=50+1 后，被系统中断，此时线程 2 对 num=50 执行了减一操作，当线程 1 恢复，在执行 num=tmp=51。而正确结果应为 50。所以当多个线程对共享区域进行修改时，应该采用同步的方式。

五. 线程同步线程同步的三种方式：

（1）互斥量　互斥量用 pthread_mutex_t 数据类型来表示。

两种方式初始化，第一种：赋值为常量 PTHREAD_MUTEX_INITIALIZER；第二种，当互斥量为动态分配时，使用 pthread_mutex_init 函数进行初始化，使用 pthread_mutex_destroy 函数销毁。

```
#include<pthread.h>
int pthread_mutex_init (pthread_mutex_t *__mutex,
                        __const pthread_mutexattr_t *__mutexattr);
int pthread_mutex_destroy (pthread_mutex_t *__mutex);
```

返回值：成功-0，失败-错误编号 。

加解锁：加锁调用 pthread_mutex_lock，解锁调用 pthread_mutex_unlock。

```
#include<pthread.h>
int pthread_mutex_lock (pthread_mutex_t *__mutex);
int pthread_mutex_unlock (pthread_mutex_t *__mutex);
```

使用互斥量修改上一个程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

int num=0;
pthread_mutex_t mylock=PTHREAD_MUTEX_INITIALIZER;

void *add(void *arg) {
   int i = 0,tmp;
   for (; i <500; i++)
```

```
    {
        pthread_mutex_lock(&mylock);
        tmp=num+1;
        num=tmp;
        printf("+1,result is:%d\n",num);
        pthread_mutex_unlock(&mylock);
    }
    return ((void *)0);
}

void *sub(void *arg)
{
    int i=0,tmp;
    for(;i<500;i++)
    {
        pthread_mutex_lock(&mylock);
        tmp=num-1;
        num=tmp;
        printf("-1,result is:%d\n",num);
        pthread_mutex_unlock(&mylock);
    }
    return ((void *)0);
}

int main(int argc, char** argv) {

    pthread_t tid1,tid2;
    int err;
    void *tret;
```

```
    err=pthread_create(&tid1,NULL,add,NULL);//创建线程
    if(err!=0)
    {
        printf("pthread_create error:%s\n",strerror(err));
        exit(-1);
    }
    err=pthread_create(&tid2,NULL,sub,NULL);
    if(err!=0)
    {
        printf("pthread_create error:%s\n",strerror(err));
         exit(-1);
    }
    err=pthread_join(tid1,&tret);//阻塞等待线程 id 为 tid1 的线
程，直到该线程退出
    if(err!=0)
    {
        printf("can not join with thread1:%s\n",strerror(err));
        exit(-1);
    }
    printf("thread 1 exit code %d\n",(int)tret);
    err=pthread_join(tid2,&tret);
    if(err!=0)
    {
        printf("can not join with thread1:%s\n",strerror(err));
        exit(-1);
    }
    printf("thread 2 exit code %d\n",(int)tret);
    return 0;
}
```

（2）读写锁　允许多个线程同时读，只能有一个线程同时写。适用于读的次数远大于写的情况。 读写锁初始化：

```
#include<pthread.h>
int pthread_rwlock_init (pthread_rwlock_t *__restrict __rwlock,
                         __const pthread_rwlockattr_t *__restrict
                         __attr);
int pthread_rwlock_destroy (pthread_rwlock_t *__rwlock);
```

返回值：成功--0，失败-错误编号。

加锁，这里分为读加锁和写加锁。
读加锁：
int pthread_rwlock_rdlock (pthread_rwlock_t *__rwlock)
写加锁：
int pthread_rwlock_wrlock (pthread_rwlock_t *__rwlock)
解锁用同一个函数：
int pthread_rwlock_unlock (pthread_rwlock_t *__rwlock)

（3）条件变量条件变量用 pthread_cond_t 数据类型表示。条件变量本身由互斥量保护，所以在改变条件状态前必须锁住互斥量。
条件变量初始化：
第一种，赋值常量 PTHREAD_COND_INITIALIZER；第二种，使用 pthread_cond_init 函数
int pthread_cond_init (pthread_cond_t *__restrict __cond,
__const pthread_condattr_t *__restrict
__cond_attr);int pthread_cond_destroy (pthread_cond_t *__cond);

条件等待

使用 pthread_cond_wait 等待条件为真。

```
pthread_cond_wait (pthread_cond_t *__restrict __cond,
    pthread_mutex_t *__restrict __mutex)
```

这里需要注意的是，调用 pthread_cond_wait 传递的互斥量已锁定，pthread_cond_wait 将调用线程放入等待条件的线程列表，然后释放互斥量，在 pthread_cond_wait 返回时，再次锁定互斥量。

唤醒线程

pthread_cond_signal 唤醒等待该条件的某个线程，

pthread_cond_broadcast 唤醒等待该条件的所有线程。

```
int pthread_cond_signal (pthread_cond_t *__cond);
int pthread_cond_broadcast (pthread_cond_t *__cond)
```

再来一个例子，主线程启动 4 个线程，每个线程有一个参数 i（i=生成顺序），无论线程的启动顺序如何，执行顺序只能为：线程 0、线程 1、线程 2、线程 3。

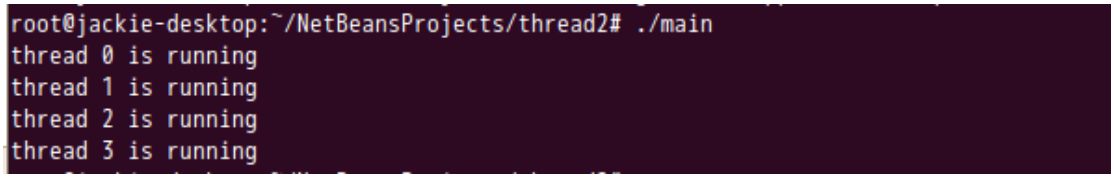
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#define DEBUG 0
//#define DEBUG 1   //控制是否进行调试的开关。

int num=0;
pthread_mutex_t mylock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t qready=PTHREAD_COND_INITIALIZER;
```

```
void * thread_func(void *arg)
{
   int i=(int)arg;
   int ret;
   sleep(5-i);//线程睡眠,然最先生成的线程，最后苏醒
   pthread_mutex_lock(&mylock);//调用 pthread_cond_wait
前，必须获得互斥锁
   while(i!=num)
   {
#ifdef DEBUG
      printf("thread %d waiting\n",i);
#endif
      ret=pthread_cond_wait(&qready,&mylock);//该函数把线程
放入等待条件的线程列表，然后对互斥锁进行解锁，这两部都是原
子操作。并且在 pthread_cond_wait 返回时，互斥量再次锁住。
      if(ret==0)
      {
#ifdef DEBUG
         printf("thread %d wait success\n",i);
#endif
      }else
      {
#ifdef DEBUG
         printf("thread %d wait failed:%s\n",i,strerror(ret));
#endif
      }
   }
   printf("thread %d is running \n",i);
   num++;
```

```
    pthread_mutex_unlock(&mylock);//解锁
    pthread_cond_broadcast(&qready);//唤醒等待该条件的所有
线程
    return (void *)0;
}
int main(int argc, char** argv) {

    int i=0,err;
    pthread_t tid[4];
    void *tret;
    for(;i<4;i++)
    {
        err=pthread_create(&tid[i],NULL,thread_func,(void *)i);
        if(err!=0)
        {
            printf("thread_create error:%s\n",strerror(err));
            exit(-1);
        }
    }
    for (i = 0; i < 4; i++)
    {
        err = pthread_join(tid[i], &tret);
        if (err != 0)
        {
            printf("can not join with thread %d:%s\n",
i,strerror(err));
            exit(-1);
        }
    }
```

```
    return 0;
}
```

在非 DEBUG 模式，执行结果如图所示：



在 DEBUG 模式，执行结果如图所示:

在 DEBUG 模式可以看出，线程 3 先被唤醒，然后执行 pthread_cond_wait（输出 thread 3 waiting)，此时在 pthread_cond_wait 中先解锁互斥量，然后进入等待状态。这是 thread 2 加锁互斥量成功，进入 pthread_cond_wait(输出 thread 2 waiting） ,同样解锁互斥量，然后进入等待状态。直到线程 0，全局变量与线程参数 i 一致，满足条件，不进入条件等待，输出 thread 0 is running。全局变量 num 执行加 1 操作，解锁互斥量，然后唤醒所有等待该条件的线程。thread 3 被唤醒，输出 thread 3 wait success。但是不满足条件，再次执行 pthread_cond_wait。如此执行下去，满足条件的线程执行，不满足条件的线程等待。

六. 作业题：

主函数通过创建两个线程来实现对一个数的递加（从 0 到 10），达到类似于以下的输出（其实就是两个线程交替的递增一个共享变量 ^-^ ）。