

**软件学院学生实验报告**

**实验课程名称：** 计算机操作系统 **教师：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **进程和线程** | | | **实验成绩** |  |
| **学生姓名** |  | **学号** |  | **年级专业班级** |  |
| **小组成员** |  | | | **实验日期** |  |

1. **实验目的**

了解进程及线程的创建、销毁和运行过程中操作系统的活动过程；

**二、实验软硬件环境**

安装有Ubuntu操作系统的计算机一台；

**三、实验内容及要求**

1、实现进程和线程的创建、销毁和运行；

2、熟悉进程及线程的创建、销毁和运行过程中操作系统的活动过程

1. **实验记录**
2. **进程**

1.1进程的定义：

狭义定义：进程就是程序的执行过程。进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。——《百度百科》

进程是一个正在执行的程序实例。——《操作系统精髓与设计原理》

1.2 实验代码：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{ pid\_t pid;

pid = fork();

if (pid == 0){

printf("Child Process pid:%d\n",getpid());

execlp("/usr/bin/top","top", NULL);

}

else{

wait(NULL);

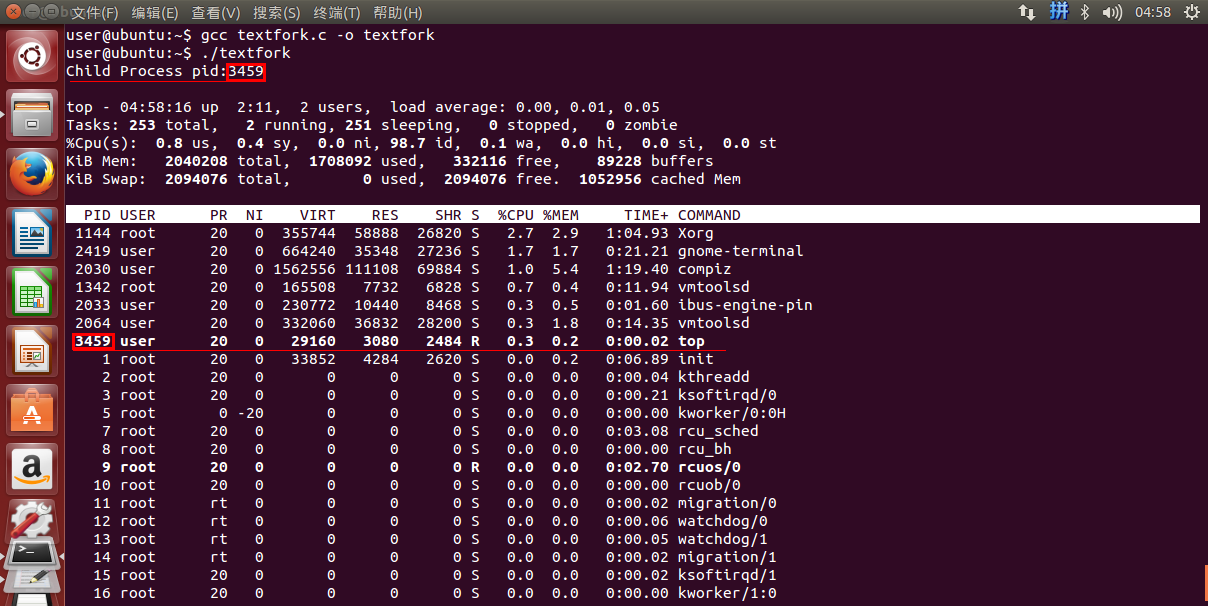
printf("Parent Process Completed.\n");

}

}

1.3 实验思考：

运行实验代码后如图1：



图一

利用fork()函数创建子进程，并使用execlp函数①执行top命令，在Linux操作系统中，top命令经常用来监控linux的系统状况，使用top命令，我们可以看到正在运行或者睡眠中的进程，而属性名代表关系如下表:

|  |  |
| --- | --- |
| PID | 进程id |
| USER | 进程所有者 |
| PR | 进程优先级 |
| NI | nice值。负值表示高优先级，正值表示低优先级 |
| VIRT | 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES |
| RES | 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA |
| SHR | 共享内存大小，单位kb |
| S | 进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程 |
| %CPU | 上次更新到现在的CPU时间占用百分比 |
| %MEM | 进程使用的物理内存百分比 |
| TIME+ | 进程使用的CPU时间总计，单位1/100秒 |
| COMMAND | 进程名称（命令名/命令行） |

在图1中可见子进程进程id为3459，而且在进程列表中查询PID，可以查询到PID3459,这个进程就是程序创建的子进程；输入命令“p”退出top命令，得图2：

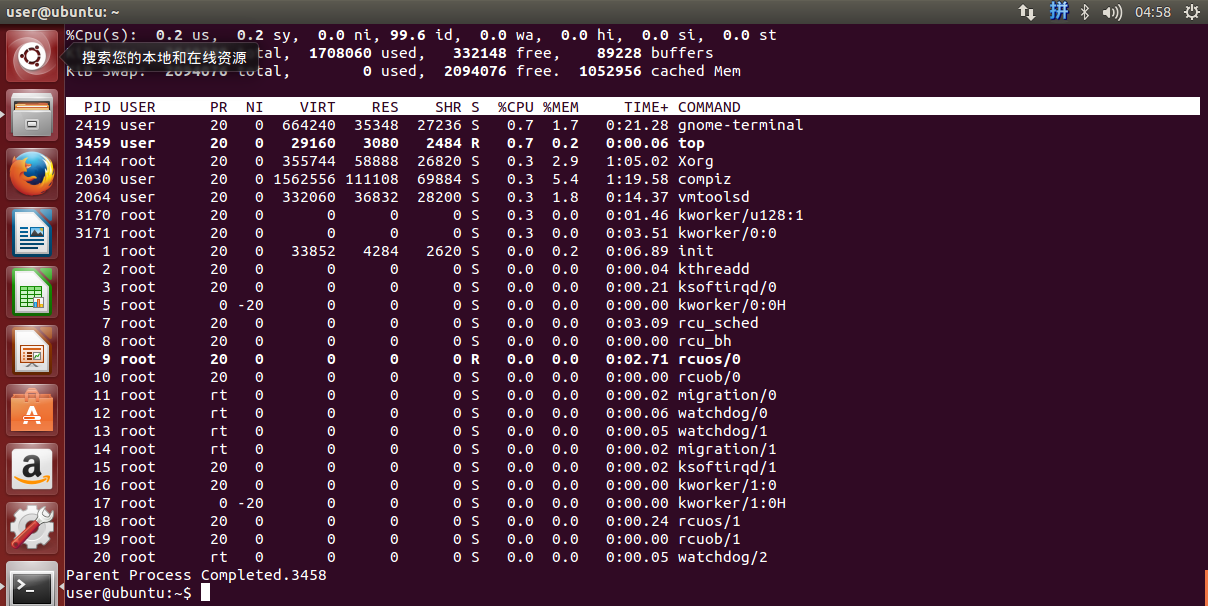


图2

此时得父进程进程ID为3458，但是为什么if()函数中的if 与else 都执行了呢？查询资料得知fork()函数的返回值为父进程以及子进程的进程ID，fork()有两个返回值，是因为由于在复制时复制了[父进程](https://baike.so.com/doc/3943920-4138829.html)的[堆栈段](https://baike.so.com/doc/7558237-7832330.html)，所以两个进程都停留在fork函数中，等待返回。因此fork函数会返回两次，一次是在父进程中返回，另一次是在子进程中返回，这两次的返回值是不一样的。过程如图2：

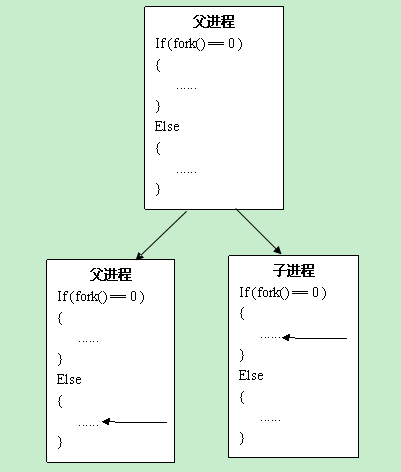


图3（引用自百度百科）

而子进程和父进程在程序结束时都被销毁了，在终端中输入命令ps -A得到当前进程列表如图4：

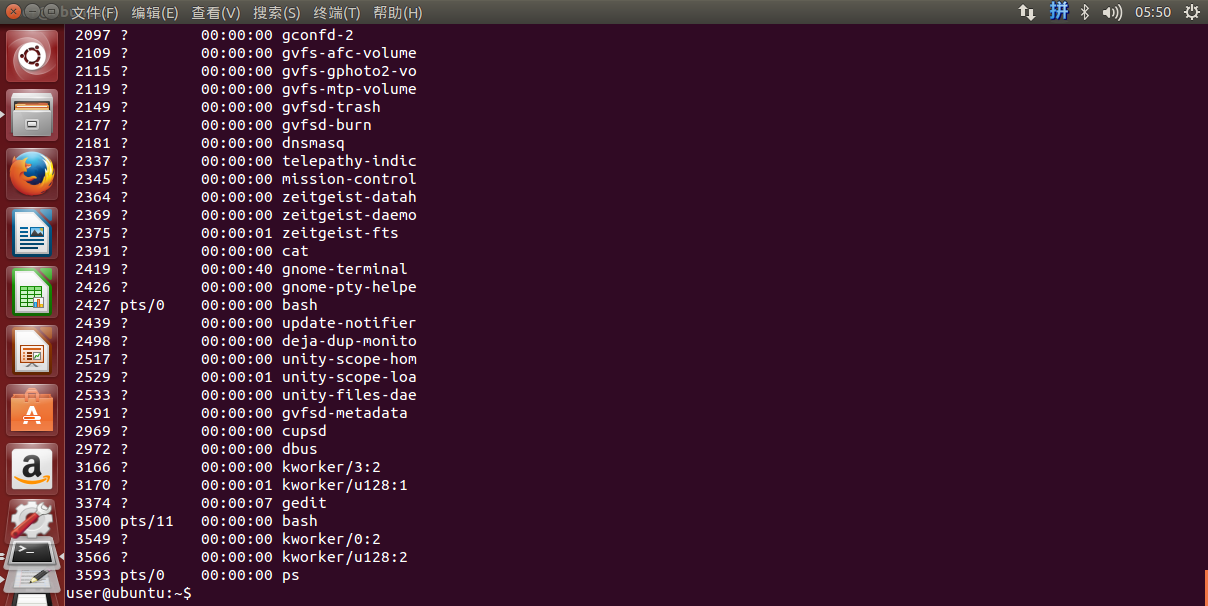


图4

PID3458与3459已经不在正在运行进程列表之中，上述的父进程与子进程在程序运行结束时一同被销毁了，它们之前所占用的系统资源、进程id等都被系统回收。

1. **线程**

2.1线程的概念：

线程，被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。

线程从处理方式区分拥有单线程与多线程两种，单线程在程序执行时，所走的程序路径按照连续顺序排下来，前面的必须处理好，后面的才会执行。多线程是为了同步完成多项任务，不是为了提高运行效率，而是为了提高资源使用效率来提高系统的效率。线程是在同一时间需要完成多项任务的时候实现的。

多线程与单线程区别如图5（左为单线程，右为多线程）：

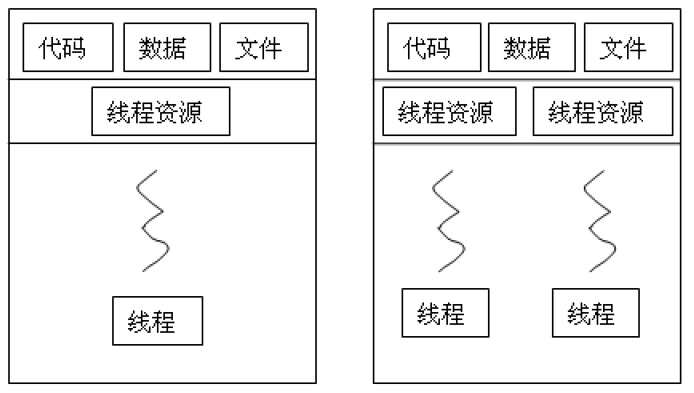


图5

而从实现方式上划分，线程有两种类型:"用户级线程"和"内核级线程"(kernel-level treads)。用户线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。这种线程甚至在象 DOS 这样的操作系统中也可实现，但线程的调度需要用户程序完成，这有些类似 Windows 3.x 的协作式多任务。

内核线程(kernel-level threads)指需要内核的参与，由内核完成线程的调度。其依赖于操作系统核心，由内核的内部需求进行创建和撤销，这两种模型各有其好处和缺点。用户线程不需要额外的内核开支，并且用户态线程的实现方式可以被定制或修改以适应特殊应用的要求，但是当一个线程因 I/O 而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会;而内核线程则没有各个限制，有利于发挥多处理器的并发优势，但却占用了更多的系统开支。

用户级线程与内核级线程实现方式的区别如图6：

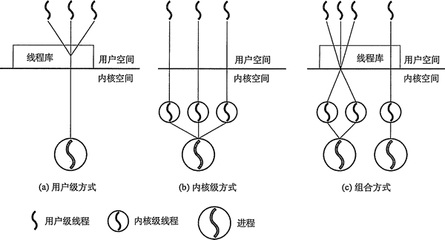


图6

2.2 实验代码：

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#define NUM\_THREADS 8

void \* perform(void \* argu){

int value\_of\_argu;

value\_of\_argu = \*((int \*)argu);

printf("\*\*\*trd%d\*\*\* Hello, world!\n", value\_of\_argu);

return NULL;

}

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

pthread\_t threads[NUM\_THREADS];

int thread\_args[NUM\_THREADS];

int value\_of\_result = 0;

int index = 0;

for (index = 0; index < NUM\_THREADS; index++){

thread\_args[index] = index;

printf("\*\*\*main\*\*\* Process creates thread %d.\n", index);

value\_of\_result=pthread\_create(&threads[index],NULL,perform, &thread\_args[index]);

assert②(!value\_of\_result);

}

for (index = 0; index < NUM\_THREADS; index++) {

value\_of\_result =pthread\_join(threads[index], NULL);

assert(!value\_of\_result);

printf("\*\*\*main\*\*\* Thread %d exited.\n", index);

}

printf("\*\*\*main\*\*\* All threads has completed successfully.\n");

return 0;

}

2.3实验思考：

上面的示例中，我们使用到了两个函数，pthread\_create和pthread\_join，并声明了一个pthread\_t型的变量数组。

　　查阅得知pthread\_t在头文件/usr/include/bits/pthreadtypes.h③中定义：

　　typedef unsigned long int pthread\_t;

　　它是一个线程的标识符。函数pthread\_create用来创建一个线程，它的原型为：

　　extern int pthread\_create \_\_P ((pthread\_t \*\_\_thread, \_\_const pthread\_attr\_t \*\_\_attr,

　　void \*(\*\_\_start\_routine) (void \*), void \*\_\_arg));

　　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，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。函数thread不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。对线程属性的设定和修改我们将在下一节阐述。当创建线程成功时，函数返回0，若不为0则说明创建线程失败，常见的错误返回代码为EAGAIN和EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

　　函数pthread\_join用来等待一个线程的结束。函数原型为：

　　extern int pthread\_join \_\_P ((pthread\_t \_\_th, void \*\*\_\_thread\_return));

　　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。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收回。一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread\_exit来实现。它的函数原型为：

　　extern void pthread\_exit \_\_P ((void \*\_\_retval)) \_\_attribute\_\_ ((\_\_noreturn\_\_));

唯一的参数是函数的返回代码，只要pthread\_join中的第二个参数thread\_return不是NULL，这个值将被传递给thread\_return。最后要说明的是，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。

第一次运行实验程序，得图7结果：

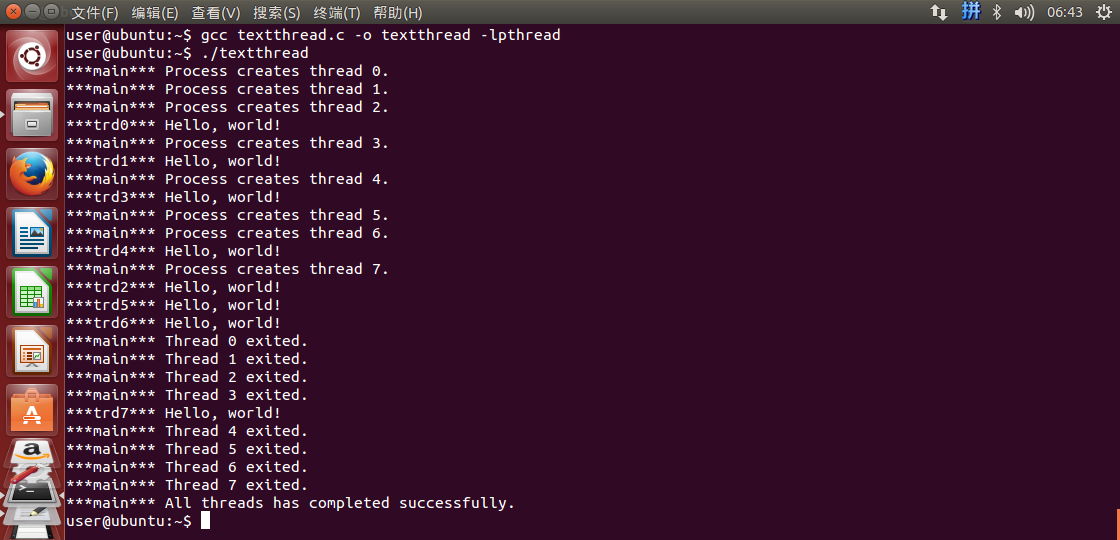


图7

由上图可以发现：代码设定创建8个线程，但是由代码printf("\*\*\*main\*\*\* Process creates thread %d.\n", index);的执行顺序我们可以看到，8个线程并不是在同一段时间创建，而由代码printf("\*\*\*trd%d\*\*\* Hello, world!\n", value\_of\_argu);以及printf("\*\*\*main\*\*\* Thread %d exited.\n", index);的执行顺序我们可以看出线程也不是同一段时间运行以及销毁，有了这个结论，为了消除实验的偶然性，再次执行实验代码，得图8结果：

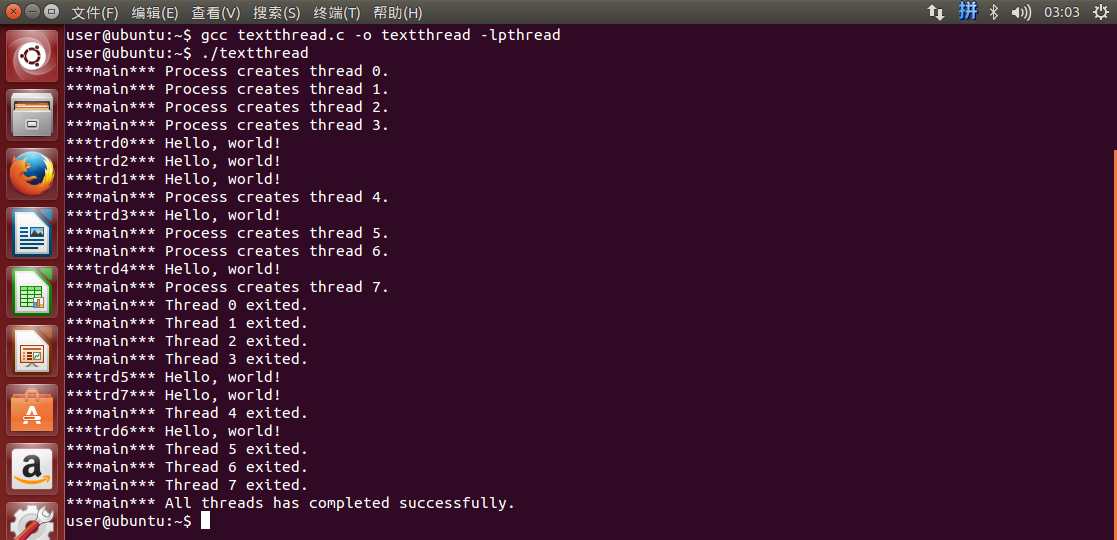


图8

由图8我们得到了与图7不同的结果，前后两次结果不一样，这是两个线程争夺CPU资源的结果。为了更明白这种线程的竞争关系，将函数void \* perform(void \* argu);修改为

void \* perform(void \* argu,int n){

int value\_of\_argu;

value\_of\_argu = \*((int \*)argu);

printf("\*\*\*trd%d\*\*\* n=%d\n", value\_of\_argu,n);

return NULL;

}

并定义一个全局变量n；以此来查看创建的8个进程是否是以顺序修改n的值，运行结果如图9：

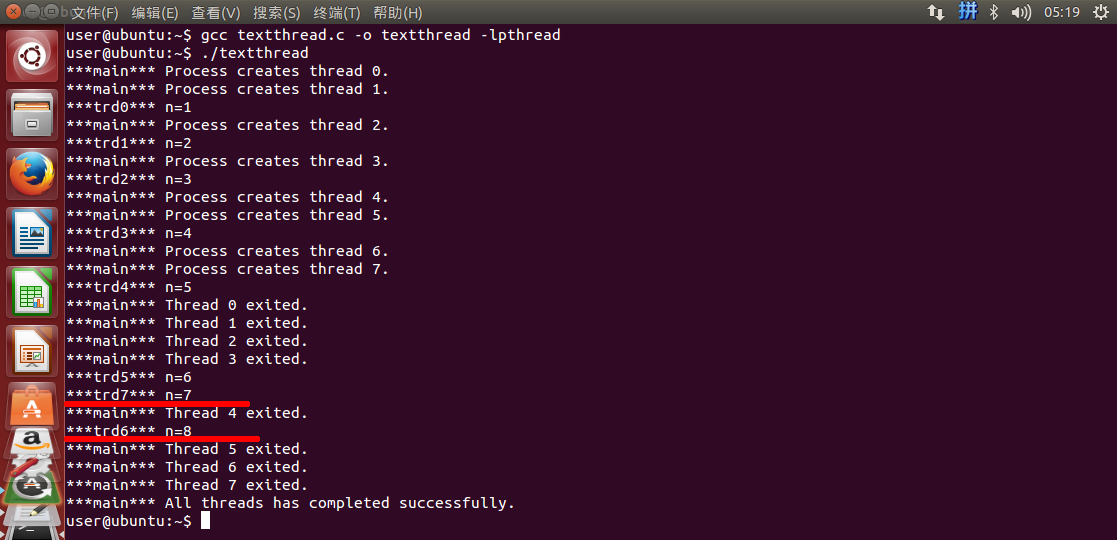


图9

在图中我们可以看到划线两行：进程7比进程6更早地修改了n的值，线程的竞争关系造成了这种情况；

1. **进程与线程的关系与区别：**

3.1关系：

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行。相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。进程与进程关系如图10：

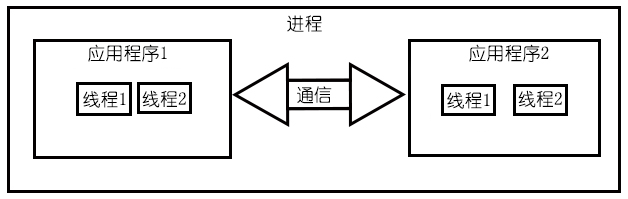


图10

3.2区别：

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

简而言之,一个程序至少有一个进程,一个进程至少有一个线程。b. 线程的划分尺度小于进程，使得多线程程序的并发性高。c.另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。d.线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。e.从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

**五、实验小结（心得体会、遇到的问题及解决方式、未解决/需进一步研讨的问题或建议新实验方法等）**

虽然稍微理解了进程的各种运作方式，但是对于线程的理解还是远远跟不上要求，在做实验报告的途中遇见了许多的问题，虽然通过查阅资料获得了一定的解答，但是还是远远不够。

参考文献：

①execlp()函数会从PATH 环境变量所指的目录中查找符合参数file的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的argv[0]、argv[1]……，最后一个参数必须用空[指针](https://baike.so.com/doc/1043844-1104112.html" \t "https://baike.so.com/doc/_blank)(NULL)作结束。

——百度百科<https://baike.so.com/doc/1269966-1342934.html>

②assert()函数：编写代码时，我们总是会做出一些假设，断言就是用于在代码中捕捉这些假设，可以将断言看作是[异常处理](https://baike.so.com/doc/6283388-6496853.html)的一种高级形式。断言表示为一些[布尔表达式](https://baike.so.com/doc/6755936-6970521.html)，程序员相信在程序中的某个特定点该表达式值为真。可以在任何时候启用和禁用断言验证，因此可以在测试时启用断言，而在部署时禁用断言。同样，程序投入运行后，最终用户在遇到问题时可以重新起用断言。

——360百科<https://baike.so.com/doc/1673469-1769457.html>

③头文件pthreadtypes.h里包含pthread\_join()以及pthread\_create()等操作函数；POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。Windows操作系统也有其移植版pthreads-win32 。

——百度百科<https://baike.baidu.com/item/Pthread/4623312?fr=aladdin>