西安交通大学

操作系统专题实验报告

班级： 计算机2102

学号： 2215012469

姓名： 向胤兴

2023年 12 月 15 日

目 录

1 openEuler 系统环境实验.................................................................................................................1

1.1 实验目的 ............................................................................................................................... 1

1.2 实验内容 ............................................................................................................................... 1

1.3 实验思想 ............................................................................................................................... 2

1.4 实验步骤 ............................................................................................................................... 2

1.5 测试数据设计....................................... .................................................................................4

1.6 程序运行初值及运行结果分析.............................................................................................9

1.7 实验总结 ............................................................................................................................. 13

1.7.1 实验中的问题与解决过程...................................................................................... 13

1.7.2 实验收获.................................................................................................................. 14

1.7.3 意见与建议.............................................................................................................. 14

1.8 附件 ..................................................................................................................................... 14

1.8.1 附件 1 程序............................................................................................................. 14

1.8.2 附件 2 Readme......................................................................................................... 30

2 进程通信与内存管理..................................................................................................................... 44

2.1 实验目的 ............................................................................................................................. 44

2.2 实验内容 ............................................................................................................................. 45

2.3 实验思想 ............................................................................................................................. 46

2.4 实验步骤 ............................................................................................................................. 47

2.5 测试数据设计.......................................................................................................................48.

2.6 程序运行初值及运行结果分析...........................................................................................50.

2.7 页面置换算法复杂度分析...................................................................................................56

2.8 回答问题 ............................................................................................................................. 57

2.8.1 软中断通信.............................................................................................................. 57

2.8.2 管道通信.................................................................................................................. 58

2.9 实验总结 ............................................................................................................................. 59

2.9.1 实验中的问题与解决过程...................................................................................... 59

2.9.2 实验收获.................................................................................................................. 60

2.9.3 意见与建议.............................................................................................................. 60

2.10 附件 ................................................................................................................................... 61

2.10.1 附件 1 程序........................................................................................................... 61

2.10.2 附件 2 Readme....................................................................................................... 84

3 文件系统 ........................................................................................................................................ 98

3.1 实验目的 ............................................................................................................................. 98

3.2 实验内容 ............................................................................................................................. 98

3.3 实验思想 ............................................................................................................................. 98

3.4 实验步骤 ........................................................................................................................... 103

3.5 程序运行初值及运行结果分析.........................................................................................105.

3.6 实验总结 ........................................................................................................................... 110

3.6.1 实验中的问题与解决过程.................................................................................... 110

3.6.2 实验收获................................................................................................................ 110

I

3.6.3 意见与建议............................................................................................................ 110

3.7 附件 ................................................................................................................................... 111

3.7.1 附件 1 程序........................................................................................................... 111

3.7.2 附件 2 Readme....................................................................................................... 150

# openEuler 系统环境实验

## Readme

本实验的主要实验步骤都依据实验指导书中提供的步骤进行。

**1.1实验目的、1.2 实验内容** 概要简述本次实验要验证的结论或要学习的知识

**1.3实验原理** 介绍了实验所基于的操作系统的原理

**1.4实验步骤** 简要指出本实验分几个步骤，每步要完成的目标

**1.5 测试数据设计**，记录的对实验中关于值的操作的说明。

具体实验步骤和实验结果在**1.6程序运行初值及运行结果分析**中，对实验步骤和运行结果的一些疑惑和验证过程放在**1.7.1 实验中的问题及解决过程**中。

**1.7.2 实验收获** 记录了通过本实验获得的知识和实验过程的心得。

**1.7.3 意见和建议** 是我认为实验设计可以改进的地方。

**1.8.1 附件1程序** 提供了本实验指导书中所有步骤中所有代码。

## 1.1进程相关编程实验

### 1.1.1实验目的

（1）熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看  
系统基本信息以了解系统；  
（2）编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管  
理等机制在操作系统实际运行中的作用

### 1.1.2 实验内容

（1）熟悉操作命令、编辑、编译、运行程序。完成图 1-1 程序的运行验证，  
多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。  
图 1-1 教材中所给代码（p103 作业 3.7）  
（2）扩展图 1-1 的程序：  
a） 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出  
操作结果并解释；  
b） 在 return 前增加对全局变量的操作并输出结果，观察并解释；  
c） 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函  
数；

### 1.1.3 实验原理

（1）进程：进程是计算机科学中的一个重要概念，它是操作系统中的基本执  
行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执  
行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制；

（2）PID：PID 是进程标识符（Process Identifier）的缩写，它是用来唯  
一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分  
配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程；

（3）fork()函数：fork() 是一个在类 Unix 操作系统中常见的系统调用，  
用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进  
程，它与父进程共享很多资源，但也有一些独立的属性。fork() 被用于实现多  
进程编程，常见于操作系统和并发编程中。函数返回一个整数，如果返回值为负  
数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进  
程中。如果返回值大于 0，表示当前正在执行的代码是在父进程中，返回值是子  
进程的 PID。调用 fork() 函数时，操作系统会创建一个新的进程，该进程是调  
用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、  
文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

### 1.1.4 实验步骤

本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，  
进一步加入 wait()函数分析其作用。  
步骤一：编写并多次运行图 1-1 中代码。

步骤二：删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结  
果。

步骤三：修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行  
不同的操作（自行设计），观察并解释所做操作和输出结果。  
  
步骤四：在步骤三基础上，在 return 前增加对全局变量的操作（自行设  
计）并输出结果，观察并解释所做操作和输出结果。  
  
步骤五：修改图 1-1 程序，在子进程中调用 system()与 exec 族函数。编写  
system\_call.c 文件输出进程号 PID，编译后生成 system\_call 可执行文

件。在子进程中调用 system\_call,观察输出结果并分析总结

### 1.1.5 测试数据设计

本实验的五个步骤中，只有步骤3、4需要给全局变量设置初值。

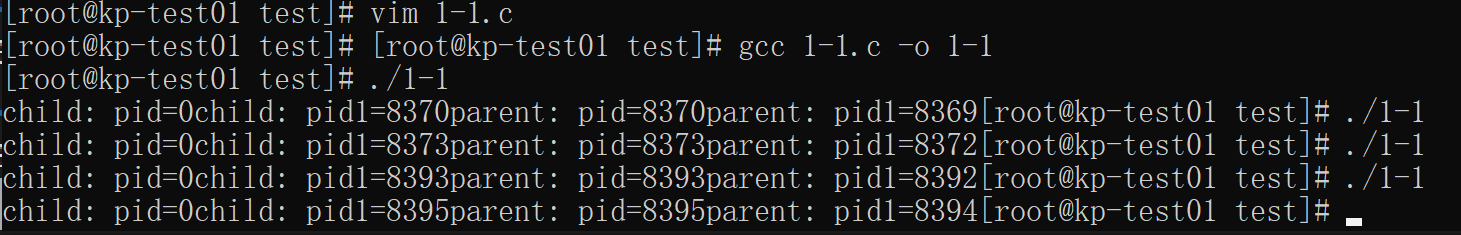
我在该实验中声明了int类型全局变量value，并初始化为0。在子进程中进行减一操作并打印子进程中value的值和其存储地址，在父进程中加一并同样打印其值和地址。

### 1.1.6 程序运行初值及运行结果分析

步骤1:执行原始代码

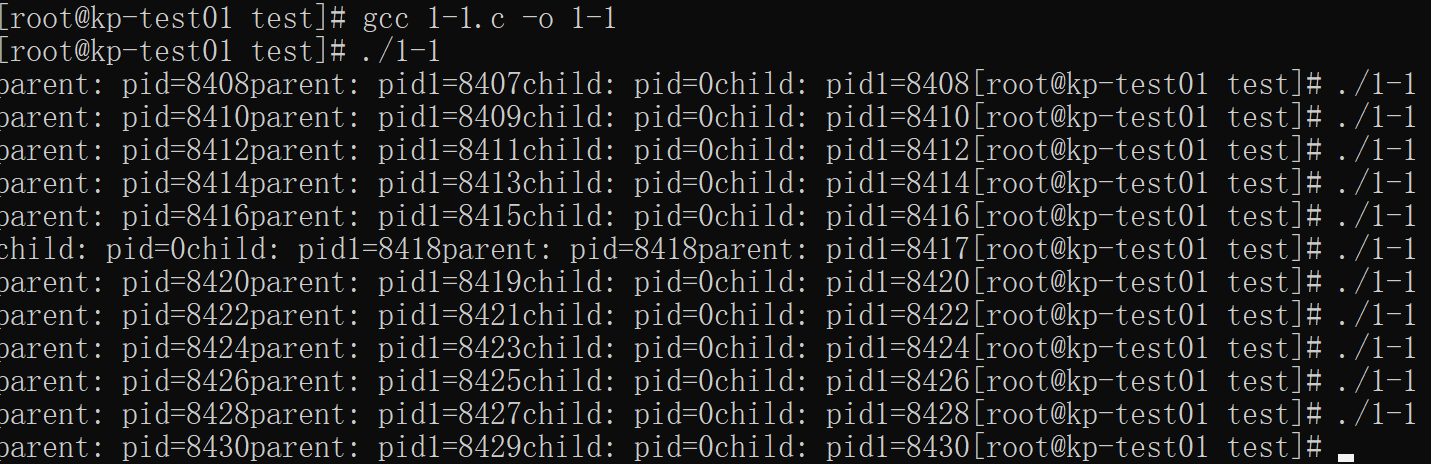
该步骤无需设置初值

运行结果：



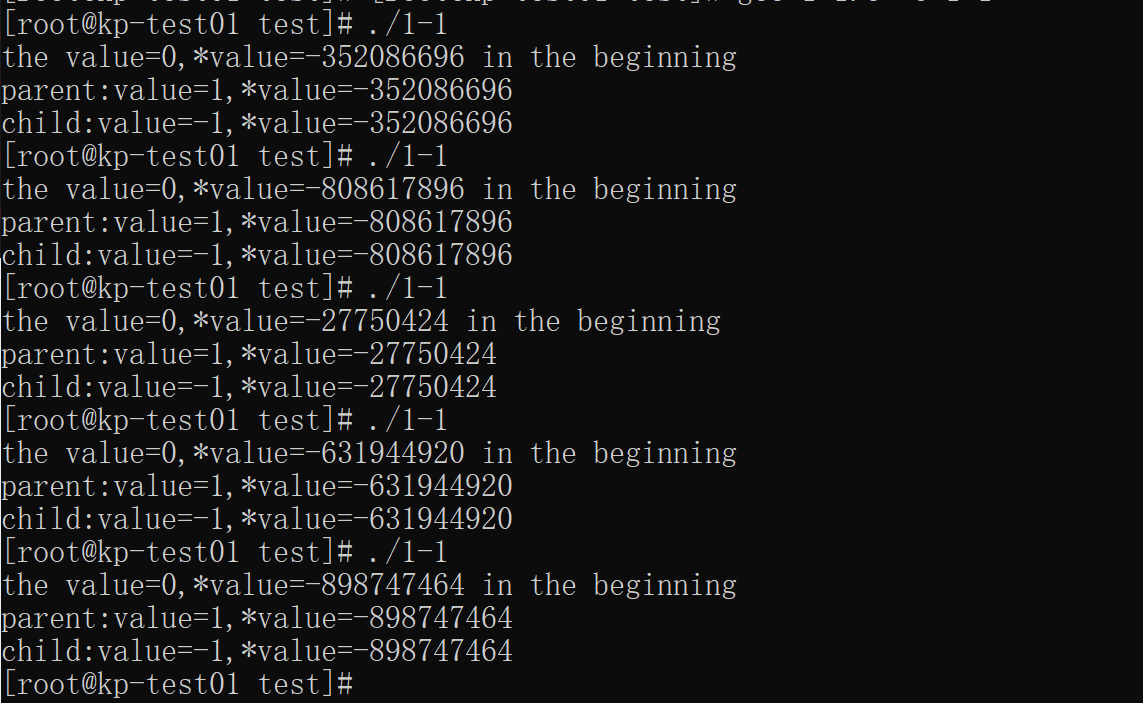
结果分析：pid\_t类型变量是用来标识进程的，每个进程都有它的pid。在父进程中调用pid=fork()分裂产生子进程时，子进程中pid的值会返回为0，而父进程中的pid的值返回为子进程的pid，在父、子进程中分别可以调用getpid()来返回当前进程的pid值。子进程比父进程后创建，对应的pid值比父进程的pid值大。

步骤2：删除父进程中的wait(NULL)

结果：

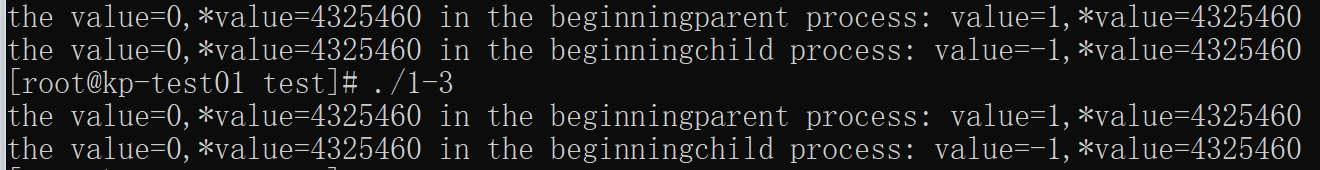
结果分析：多次执行文件，出现了一次先打印child后打印parent的情况，但经过分析，应该与是否删除wait(NULL)无关，因为wait(NULL)语句出现在打印内容之后，wait()不会影响打印顺序。

步骤3：父子进程分别对一全局变量进行不同操作，打印不同进程中全局变量的值和地址

结果：

结果分析：根据程序运行结果可以观察到父进程子进程中的全局变量的地址相同，但操作之间相互影响。

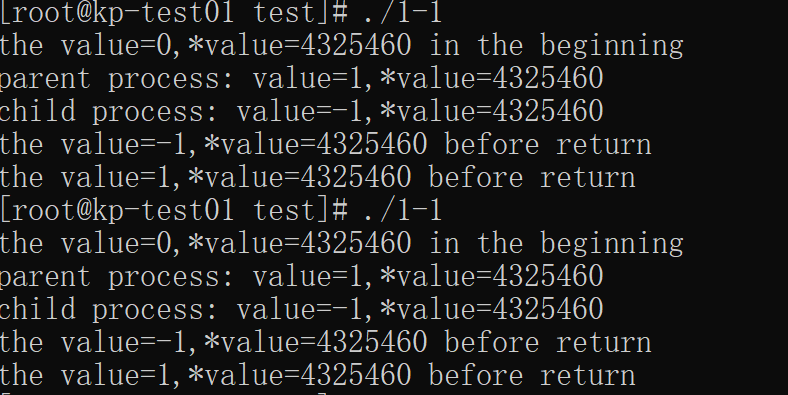
拓展：如果初始打印value时不加入换行符”\n”，fork之前的print内容会被打印两次。



原因是字符”\n”起到刷新缓冲区的作用，如果没有\n，在fork创建子进程时，在构建子进程的缓冲区时会复制一份父进程的缓冲区，会导致在fork之前的原本在父进程种的内容再次被打印一次，而打印内容种”\n”会在父进程打印刷新缓冲区，使得新得到的子进程的缓冲区为空，子进程就不会再打印一次。

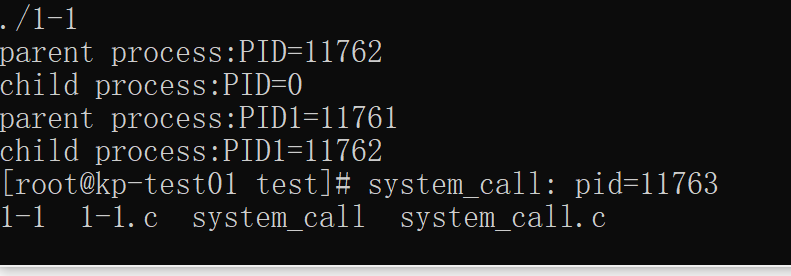
步骤4：在步骤3的基础上，在return之前打印全局变量value的值和地址

结果：



结果分析：由实验结果可知，父子进程都会进行一次return，并且在return前全局变量的值仍然为本进程处理后的值。

步骤5：在子进程中调用 system()与 exec 族函数

结果：

结果分析：在子进程中调用了system()和execlp()函数。根据结果可知，system()可以在进程中调用其他可运行文件，产生新的进程。而execlp()函数可以将指定的命令或代码用来替换本进程中剩余的代码，比如此例中用ls命令代替了剩下的代码，使得在execlp()之后的printf命令无法运行，运行结果中没有出现"this sentences won't be executed\n"。

### 1.1.7 实验总结

#### 1.7.1 实验中的问题与解决过程

在步骤1的实验中，我产生了如下疑惑：

fork()是从原进程中分裂出了子进程（新增一个进程），还是kill原进程，新增两个进程，并使其一个为父进程一个为子进程？

于是我设计代码进行了实验，在调用fork()之前使用getpid()获得当前进程的pid值并且打印，根据此时pid值和fork之后父子进程之间的pid1值来探究它们之间的关系。

代码如下：#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=getpid();

printf("pid=%d before fork\n",pid);

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child pid=%d,pid1=%d\n",pid,pid1);

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent pid=%d,pid1=%d\n",pid,pid1);

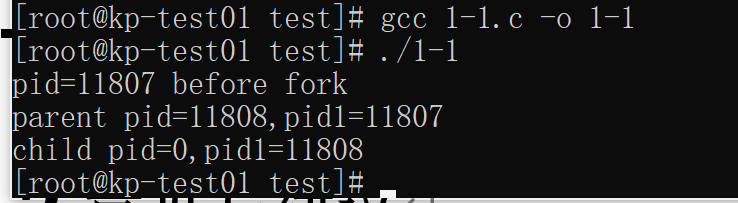
wait(NULL);

}

return 0;

}

实验结果如下：



可以观察到fork之后父进程的pid1与fork之前的原进程的pid值相同，可以得出结论fork之后的父进程即为原进程，只新产生了一个进程，即为子进程。

在步骤2的实验中，我认为是否删除在printf之后的wait(NULL)没有影响，wait(NULL)出现在parent process的printf之前会影响打印顺序，设计了如下代码验证：

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child pid=%d,pid1=%d\n",pid,pid1);

}

else{/\*Parent process\*/

wait(NULL);

pid1=getpid();

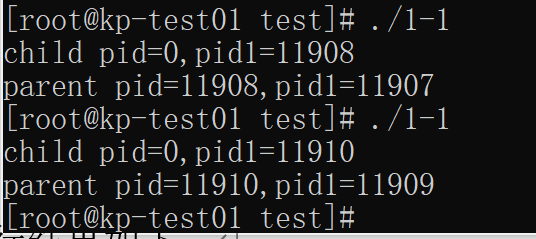
printf("parent pid=%d,pid1=%d\n",pid,pid1);

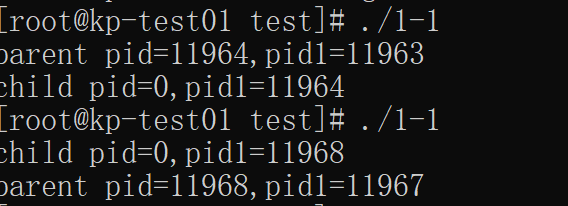
}

return 0;

}

运行结果如下：



删除parent process中的wait(NULL)后，

可能先出现parent也可能先出现child

#### 1.7.2 实验收获

通过本次实验，我更加深入的了解了fork()函数的运行原理，知道了父子进程pid值的关系。对多个进程对内存的管理有了更清晰直观的认知，了解全局变量在不同进程中是在同一个地址空间，但会分开操作保证互斥。

通过本次实验，让我了解了exec族的函数的作用，是用指定命令或代码来替换掉剩余的代码。了解了system()可以用来调用创建新进程其他的可运行文件。

总之，这次实验不光让我对多进程的内容有了更多的了解，还让我对各种对进程管理的函数有了更深入的认识。

#### 1.7.3 意见与建议

可以在指导书里简要说明一下每个步骤的目的是什么，是要验证什么原理，了解什么知识，让学生在进行实验的每一步时不光知道要写什么代码，还知道为什么要这样设计代码。

### 1.1.8 附件

#### 1.8.1 附件 1 程序

**步骤1程序：**

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child: pid=%d",pid);//A

printf("chile: pid1=%d",pid1);//B

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent: pid=%d",pid);//C

printf("parent: pid1=%d",pid1);//D

wait(NULL);

}

return 0;

}

**步骤2程序：**

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child: pid=%d",pid);//A

printf("chile: pid1=%d",pid1);//B

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent: pid=%d",pid);//C

printf("parent: pid1=%d",pid1);//D

}

return 0;

}

**步骤3程序：**

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int value=0;

int main()

{

printf("the value=%d,\*value=%d in the beginning\n",value,&value);

pid\_t pid;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

value--;

printf("child process: value=%d,\*value=%d\n",value,&value);

}

else{/\*Parent process\*/

value++;

printf("parent process: value=%d,\*value=%d\n",value,&value);

wait(NULL);

}

return 0;

}

**步骤4程序：**

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int value=0;

int main()

{

printf("the value=%d,\*value=%d in the beginning\n",value,&value);

pid\_t pid;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

value--;

printf("child process: value=%d,\*value=%d\n",value,&value);

}

else{/\*Parent process\*/

value++;

printf("parent process: value=%d,\*value=%d\n",value,&value);

wait(NULL);

}

printf("the value=%d,\*value=%d before return\n",value,&value);

return 0;

}

**步骤五程序：**

**###1-1.c###**

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child process:PID=%d\n",pid);

printf("child process:PID1=%d\n",pid1);

system("./system\_call");

execlp("ls","ls",NULL);

printf("this sentences won't be executed\n");

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent process:PID=%d\n",pid);

printf("parent process:PID1=%d\n",pid1);

}

return 0;}

### system\_call.c ###

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid;

pid=getpid();

printf("system\_call: pid=%d\n",pid);

return 0;

}

## 1.2线程相关编程实验

### 1.2.1实验目的

探究多线程编程中的线程共享进程信息。在计算机编程中，多线程是一种常  
见的并发编程方式，允许程序在同一进程内创建多个线程，从而实现并发执行。  
由于这些线程共享同一进程的资源，包括内存空间和全局变量，因此可能会出现  
线程共享进程信息的现象。本实验旨在通过创建多个线程并使其共享进程信息，  
以便深入了解线程共享资源时可能出现的问题。

### 1.2.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；  
（2）在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）  
并输出结果；  
（3）多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原  
因并修改程序解决，考虑如何控制互斥和同步；  
（4）将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线  
程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

### 1.2.3 实验原理

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操

作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况  
下，引入互斥和同步机制来确保线程间的正确协同操作。  
（1）线程创建与变量操作： 首先，在一个进程内创建两个线程，并在进程  
内部初始化一个共享的变量。这两个线程将并发地对这个共享变量进行循环操作，  
执行不同的操作。  
（2）竞态条件和不稳定结果： 由于线程并发执行，存在竞态条件，即两个  
线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程  
的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。  
（3）互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）  
来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作  
完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发  
访问带来的不稳定性。  
（4）观察结果与比较： 运行多次实验，观察使用互斥锁后的运行结果。应  
该可以发现，通过互斥锁的保护，不再出现不稳定的结果，每次运行得到的结果  
都是一致的。  
（5）调用系统函数和线程函数的比较： 在任务一中，如果将调用系统函数  
和调用 exec 族函数改成在线程中实现，观察运行结果。可以发现，调用系统函  
数和 exec 族函数时，会输出进程的 PID（Process ID），而在线程中运行时，会  
输出线程的 TID（Thread ID）。这是因为线程是进程的子任务，它们共享进程的  
资源，但有自己的执行流程。

### 1.2.4 实验步骤

步骤1：设计程序，创建两个子线程，两线程分别对同一个共享变量多次操  
作，观察输出结果。

步骤2：修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问  
与互斥。运行程序，观察最终共享变量的值。

步骤3：在第一部分实验了解了 system()与 exec 族函数的基础上，将这两  
个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

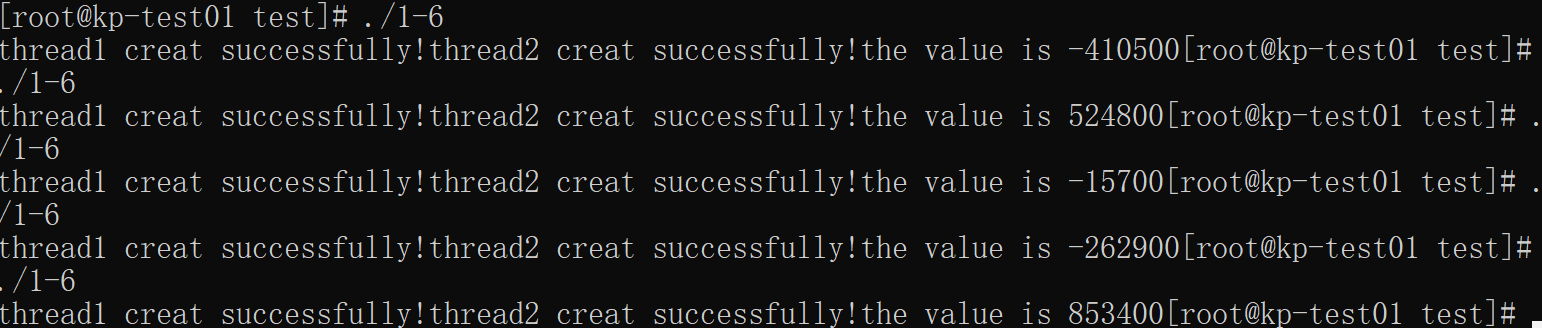
### 1.2.5 测试数据设计

在创建多线程前声明全局变量value并初始化其为0，再在后续多个线程中操作value。

### 1.2.6 程序运行初值及运行结果分析

步骤一：

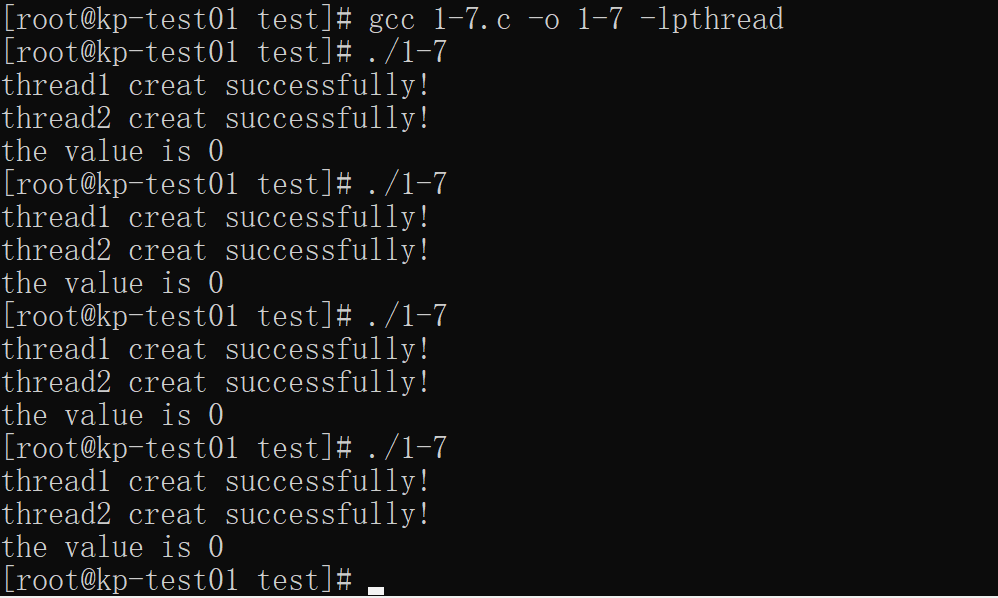
运行初值：value初值设为0，在thread1中进行100000次自增，在thread2中进行100000次自减



发现每次返回的value值不同，原因是两个线程同时对同一个变量进行操作导致value的值异常。

步骤二：

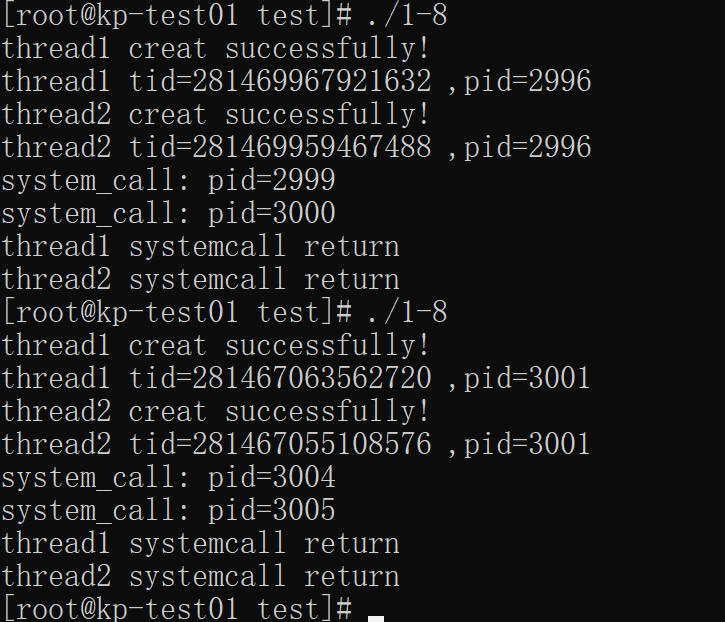
声明set\_t类型变量，使用sem\_init初始化，并使用sem\_wait(&set\_t)和sem\_post(&set\_t)分别作为P操作和V操作。实验结果如下：



根据实验结果，发现引入互斥机制后，value值最终回归正常。

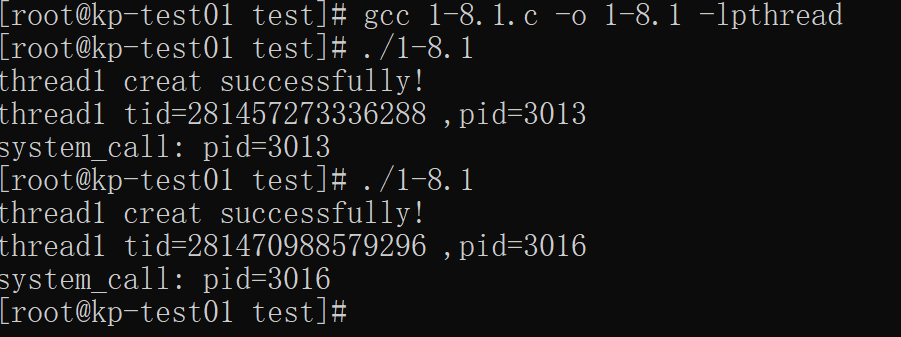
步骤3：

在两个线程中分别调用system(“./system\_call”)，运行结果如下



可以看到的pid值相同而tid值不同，说明此为两个属于相同进程的不同线程。在两个线程中分别调用了./system\_call，产生的进程的pid不同，说明两个线程可以分别创建不同的进程。

使用execlp()函数调用system\_call:



没有观察到thread2的内容，原因是thread1先创建，并且调用execlp()函数把其所在进程的在其之后的代码全部由system\_call文件中的内容覆盖，所以没有观察到thread2。

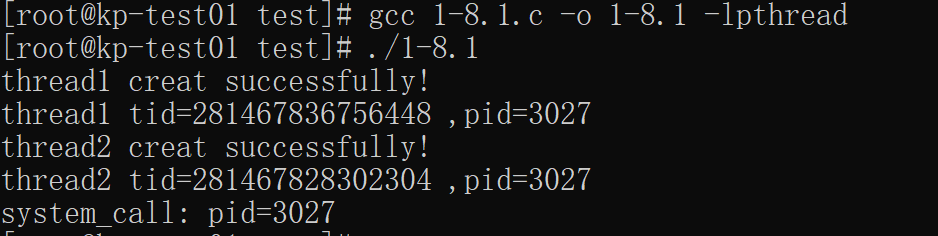
q

### 1.2.7 实验总结

#### 1.7.1 实验中的问题与解决过程

如何证明步骤3中使用execlp()时没有出现thread2的原因是还未来得及执行thread2的内容就被替换？为了验证，我进行了以下实验：

在thread1的execl()前添加sleep(10),实验结果如下图所示：



此时thread2创建成功并且成功打印，可以说明之前没有出现thread2的原因是还未创建就被替换。

#### 1.7.2 实验收获

本次实验中让我学会了怎样创建线程、结束线程。通过本次实验，对课本中的知识进行了验证，让我对进程和线程的共同点和差异性有了更深入和全面的认识。

本次实验让我更加熟练的使用system()和exec族的各种函数，从而在以后的各种实践编程中我能够更加自如的调用这两种函数。

### 1.2.8 附件

#### 1.2.8.1 附件 1 程序

步骤1程序

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

int value=0;

void \*worker1(){

int i;

for(i=0;i<100000;i++)

value++;

pthread\_exit(NULL);

}

void \*worker2(){

int i;

for(i=0;i<100000;i++)

value--;

pthread\_exit(NULL);

}

int main(){

pthread\_t thread1,thread2;

int rc;

rc=pthread\_create(&thread1,NULL,worker1,NULL);

if(rc){

printf("ERROR; return code from creat thread1 is %d\n",rc);

exit(-1);

}

rc=pthread\_create(&thread2,NULL,worker2,NULL);

if(rc){

printf("ERROR; return code from creat thread2 is %d\n",rc);

exit(-1);

}

rc=pthread\_join(thread1,NULL);

if(rc){

printf("ERROR; return code from join thread1 is %d\n",rc);

exit(-1);

}

rc=pthread\_join(thread2,NULL);

if(rc){

printf("ERROR; return code from join thread2 is %d\n",rc);

exit(-1);

}

printf("the value is %d",value);

return 0;

}

**步骤2程序：**

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

int value = 0;

sem\_t sem;

void \*worker1(void \*arg) {

int i;

for (i = 0; i < 100000; i++) {

sem\_wait(&sem); // P操作

value++;

sem\_post(&sem); // V操作

}

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

for (i = 0; i < 100000; i++) {

sem\_wait(&sem); // P操作

value--;

sem\_post(&sem); // V操作

}

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

sem\_init(&sem, 0, 1); // 初始化信号量为1

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

printf("the value is %d\n", value);

sem\_destroy(&sem); // 销毁信号量

return 0;

}

步骤3程序：

使用system()的步骤3代码：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

pid\_t pid;

void \*worker1(void \*arg) {

int i;

printf("thread1 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread1 tid=%d ,pid=%d\n",tid,pid);

system("./system\_call");

printf("thread1 systemcall return\n");

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

printf("thread2 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread2 tid=%d ,pid=%d\n",tid,pid);

system("./system\_call");

printf("thread2 systemcall return\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

return 0;

}

使用execlp()的步骤3

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

pid\_t pid;

void \*worker1(void \*arg) {

int i;

printf("thread1 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread1 tid=%d ,pid=%d\n",tid,pid);

execlp("./system\_call","system\_call",NULL);

printf("thread1 systemcall return\n");

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

printf("thread2 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread2 tid=%d ,pid=%d\n",tid,pid);

execlp("./system\_call","system\_call",NULL);

printf("thread2 systemcall return\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

return 0;

}

## 1.3自旋锁实验

### 1.3.1实验目的

自旋锁作为一种并发控制机制，可以在特定情况下提高多线程程序的性  
能。本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程  
间的同步，从而实现以下目标：  
（1）了解自旋锁的基本概念： 通过研究自旋锁的工作原理和特点，深入  
理解自旋锁相对于其他锁机制的优势和局限性；  
（2）实验自旋锁的应用： 在一个多线程的实验环境中，设计一个竞争资  
源的场景，让多个线程同时竞争对该资源的访问；  
（3）实现自旋锁的同步： 使用自旋锁来保护竞争资源的访问，确保同一  
时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

### 1.3.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；

（2）在两个线程中分别对此变量循环五千次以上做不同的操作（自行设  
计）并输出结果；  
（3）使用自旋锁实现互斥和同步；

### 1.3.3 实验原理

自旋锁是一种基于忙等待（busy-waiting）的同步机制，用于在线程竞争  
共享资源时，不断尝试获取锁，而不是阻塞等待。它的工作原理可以简单地概  
括为以下几个步骤：  
（1）初始化锁： 自旋锁的开始是一个共享的标志变量（flag），最初为未  
锁定状态（0）。这个标志变量用于表示资源是否已被其他线程占用。  
（2）获取锁： 当一个线程尝试获取锁时，它会循环检查标志变量的状  
态。如果发现标志变量是未锁定状态（0），那么该线程将通过原子操作将标志  
变量设置为锁定状态（1），从而成功获取锁。如果标志变量已经是锁定状态，  
线程会一直在循环中等待，直到标志变量变为未锁定状态为止。  
（3）释放锁： 当持有锁的线程完成对共享资源的操作后，它会通过原子  
操作将标志变量设置回未锁定状态（0），从而释放锁，允许其他等待的线程尝  
试获取锁。  
自旋锁的工作原理中关键的部分在于“自旋”这一概念，即等待获取锁的  
线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。这种方  
式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

### 1.3.4 实验步骤

步骤1：根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c

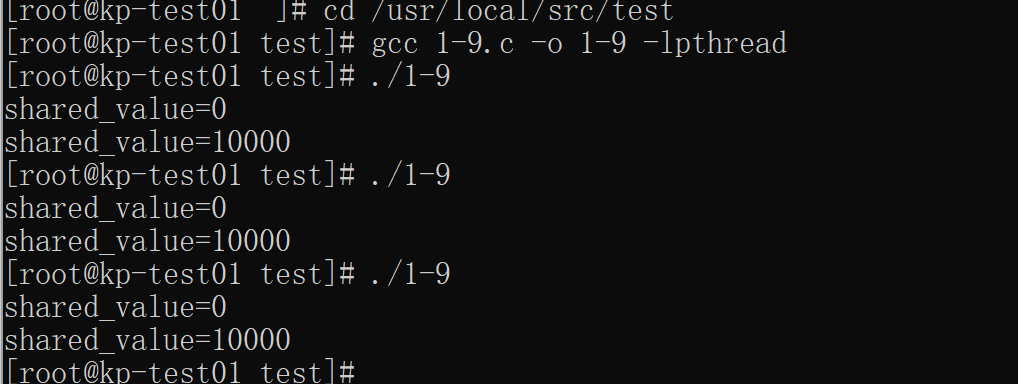
步骤2：运行补充后的文件

### 1.3.5 测试数据设计

初始时将shared\_value设置为0，两个线程分别对其进行500次加1操作。

### **1.3.6 程序运行初值及运行结果分析**

初始时将shared\_value设置为0，两个线程分别对其进行500次加1操作1.3.7 实验总结 ，实验结果如下：



由于两个进程中每次操作value值时，都申请了自旋锁，避免了同时操作数据，使得数据值不会发生异常。

### 1.3.7实验总结

#### 实验收获

通过自行设计自旋锁，让我对自旋锁的结构，以及实现功能的原理有了更深入的了解。即在有一个线程操作多个线程共同申请操作的变量是，其他线程使用while()循环等待，直到占用资源的线程释放锁，某个线程获取访问变量的权限后再次上锁，如此往复。

### 1.3.8 附件

#### 1.3.8.1 附件 1 程序

#include <stdio.h>

#include <pthread.h>

#include<stdlib.h>

// 定义自旋锁结构体

typedef struct {

int flag;

} spinlock\_t;

// 初始化自旋锁

void spinlock\_init(spinlock\_t \*lock) {

lock->flag = 0;

}

// 获取自旋锁

void spinlock\_lock(spinlock\_t \*lock) {

while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {

// 自旋等待

}

}

// 释放自旋锁

void spinlock\_unlock(spinlock\_t \*lock) {

\_\_sync\_lock\_release(&lock->flag);

}

// 共享变量

int shared\_value = 0;

// 线程函数

void \*thread\_function(void \*arg) {

spinlock\_t \*lock = (spinlock\_t \*)arg;

for (int i = 0; i < 5000; ++i) {

spinlock\_lock(lock);

shared\_value++;

spinlock\_unlock(lock);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

spinlock\_t lock;

// 输出共享变量的值

printf("shared\_value=%d\n",shared\_value);

// 初始化自旋锁

spinlock\_init(&lock);

// 创建两个线程

int rc;

rc=pthread\_create(&thread1,NULL,thread\_function,(void \*)&lock);

if(rc){printf("ERROR in create thread1\n");

exit(-1);

}

rc=pthread\_create(&thread2,NULL,thread\_function,(void \*)&lock);

if(rc){printf("ERROR in create thread2\n");

exit(-1);

}

// 等待线程结束

rc=pthread\_join(thread1,NULL);

if(rc){

printf("ERROR in wait thread1\n");

exit(-1);

}

rc=pthread\_join(thread2,NULL);

if(rc){

printf("ERROR in wait thread2\n");

exit(-1);

}

// 输出共享变量的值

printf("shared\_value=%d\n",shared\_value);

return 0;

}