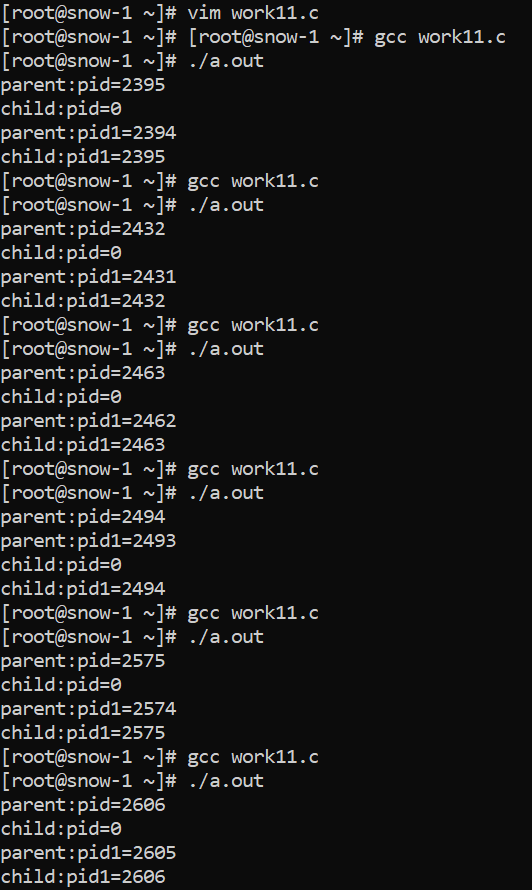
# 华为云上openEuler系统环境实验

1. 进程相关编程实验
2. **完成操作系统原理课程教材P103作业3.7的运行验证，多运行几次查看结果；**

多次运行图3-32代码：

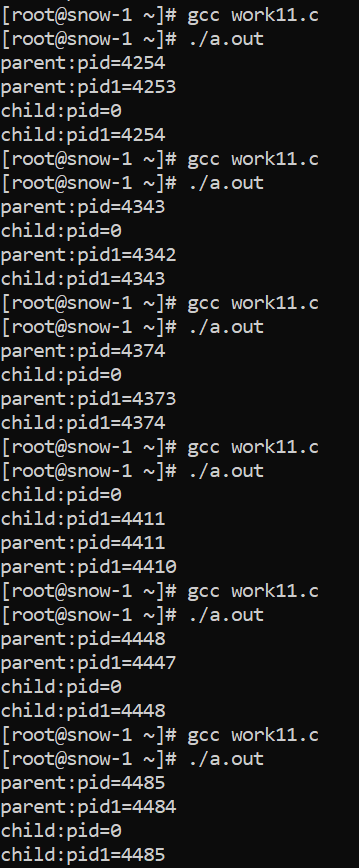


使用fork()函数后，父进程中返回的值是子进程的进程号，在子进程中返回的是0；每次输出不一样是因为系统给父子进程分的进程号一直在变化

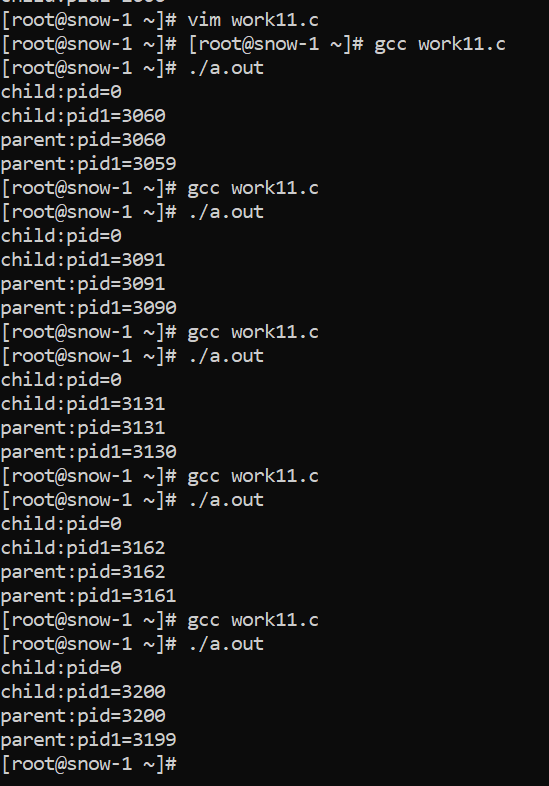
出现的问题：程序显示wait()函数无法识别

解决：加上wait()头文件

去掉结尾处的wait()后再运行：



发现去掉和没去掉没有明显区别，故在父进程开头加入wait(NULL),结果如下：



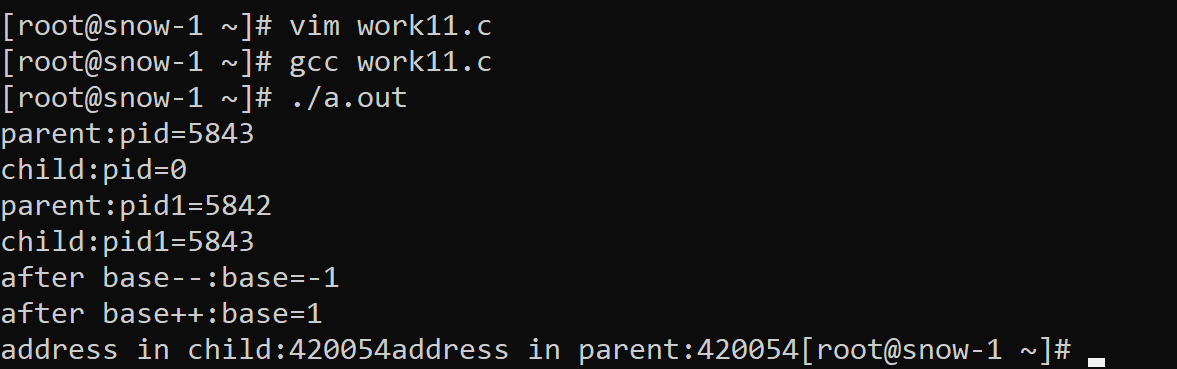
将wait(NULL)加到父线程开头发现，父进程会自我阻塞，直到子线程运行完毕。

理论分析：

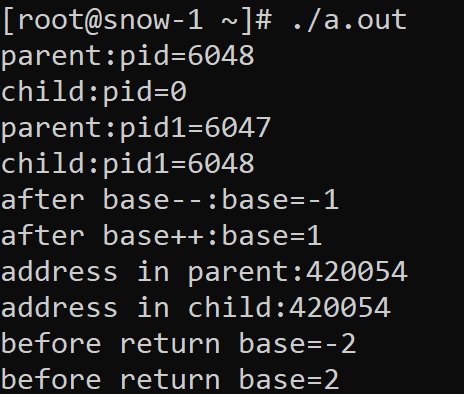
通过程序运行结果可以看出，fork()函数被调用后，在当前进程中创建了一个子进程与父进程同步运行，因此返回了两次结果。pid值为0时，表明子进程在运行，getpid()返回子进程的pid赋给pid1，为子进程的地址；pid值大于0时，表明父进程在运行，pid值为子进程的地址， getpid()返回父进程的pid赋给pid1，为父进程的地址。由于子进程为父进程的拷贝，地址为父进程地址+1。综上所述，parent:pid值与child:pid1值相同，都为子进程的地址，parent:pid1的值为父进程的地址，child:pid的值为0。

fork()函数并不确定父子进程的执行顺序，不同版本的操作系统，不同系统环境、系统负载下的执行顺序都不尽相同，这也导致了上述多次执行结果顺序的不同，由操作系统的调度算法决定。

加入全局变量后的输出：



在return前加上全局变量的操作和输出：

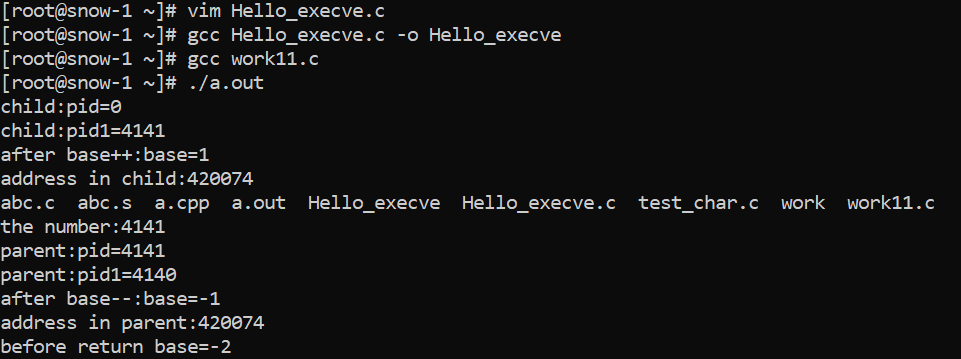


结论：1.父子进程不能共享全局变量。

2.但是如果父子进程只是对全局变量做读操作，则父子进程共享同一份全局变量。

3.如果父子进程中的任何一个对变量做修改操作，会在内存中拷贝一份副本，然后在这个副本上进行修改，修改完成后再通过MMU映射回去。

在子程序中使用system函数和execve()函数输出子进程的进程识别码：



分析：

调用exec族函数可以把当前进程替换为一个拥有相同pid的新进程。在本例中调用execve()函数作用为调用可执行文件Hello\_execve替换当前进程，根据输出可以发现替换前后的pid并未发生改变，但execve()函数后的语句由于进程被替换不再运行。在子进程中打开的程序的进程识别码与子进程相同，同样也会执行系统命令。

代码如下：

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int base=0;

int main(int argc,char \*argv[],char\*\* environ)

{

pid\_t pid,pid1;

pid=fork();

if(pid<0){

fprintf(stderr,"Fork Failed");return 1;

}

else if (pid==0){

pid1=getpid();

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

printf("child:pid1=%d\n",pid1);/\*B\*/

base++;

printf("after base++:base=%d\n",base);

printf("address in child:%x\n",&base);

system("ls");

execve("./Hello\_execve",argv,environ);

printf("not!!!\n");

}

else{

wait(NULL);

pid1=getpid();

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

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

base--;

printf("after base--:base=%d\n",base);

printf("address in parent:%x\n",&base);

}

base=base\*2;

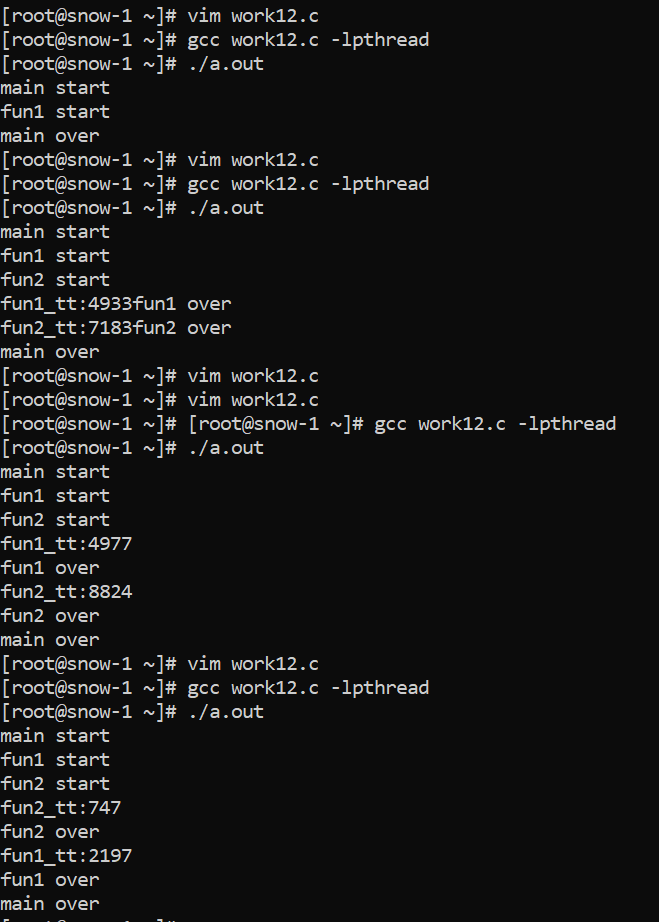
printf("before return base=%d\n",base);

return 0;

}

二、

在进程中创建两个线程并初始化一个变量，两个线程分别操作5000次并输出结果：



发现每次输出结果都不同，说明由于两个操作并不互斥，所以产生了混乱的修改结果。

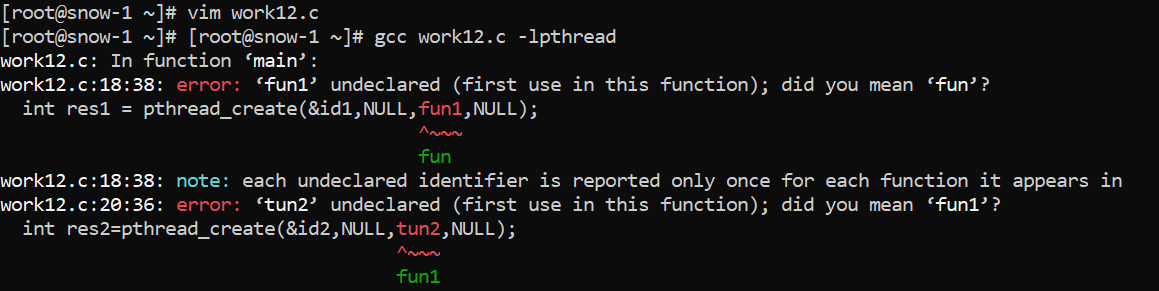
Ps:这个过程中出现的问题：

1. 问题：在linux中，gcc含有线程创建的文件时报错

解决：再使用gcc命令时，加上”-lpthread”

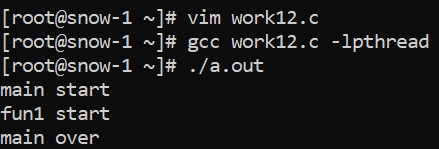
方式：Bing

1. 问题：在work12.c文件编译时报错：



解决：要在主函数之前声明主函数内用到的函数

1. 问题：主程序中运行完第一个线程后自动结束，不再运行第二个线程，如图：



解决：在主线程中运行sleep(2)，防止主线程运行后死亡，不给子线程运行机会

第一次修改，给变量tt加上锁，源码如下：

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

int tt;

//声明一个线程函数

void \*fun1(void \*);

void \*fun2(void \*);

pthread\_mutex\_t mutex;//定义互斥变量

int main()

{

printf("main start\n");

tt=0;

pthread\_mutex\_init(&mutex, NULL);//互斥变量初始化

pthread\_t id1;

pthread\_t id2;

//创建函数线程，并且指定函数线程要执行的函数

int res1 = pthread\_create(&id1,NULL,fun1,NULL);

assert(res1 == 0);

int res2=pthread\_create(&id2,NULL,fun2,NULL);

assert(res2==0);

sleep(2);

printf("main over\n");

pthread\_mutex\_destroy(&mutex);//销毁锁

exit(0);

}

//定义线程函数

void\* fun1(void \*arg)

{

printf("fun1 start\n");

pthread\_mutex\_lock(&mutex);//加锁

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

{

tt++;

}

pthread\_mutex\_unlock(&mutex);//解锁

printf("fun1\_tt:%d\n",tt);

printf("fun1 over\n");

}

void\* fun2(void \*arg)

{

printf("fun2 start\n");

pthread\_mutex\_lock(&mutex);//加锁

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

{

tt--;

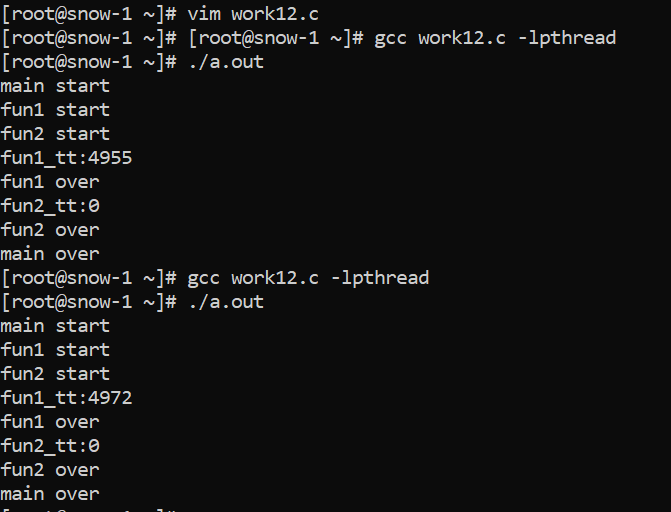
}

pthread\_mutex\_unlock(&mutex);//解锁

printf("fun2\_tt:%d\n",tt);

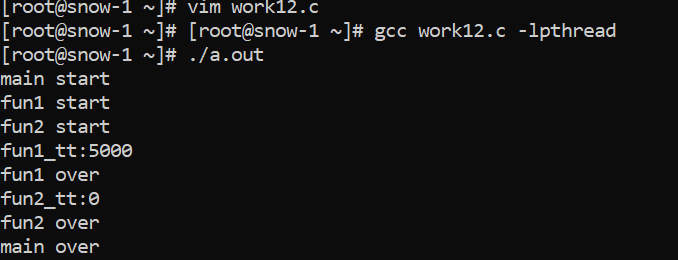
printf("fun2 over\n");

}

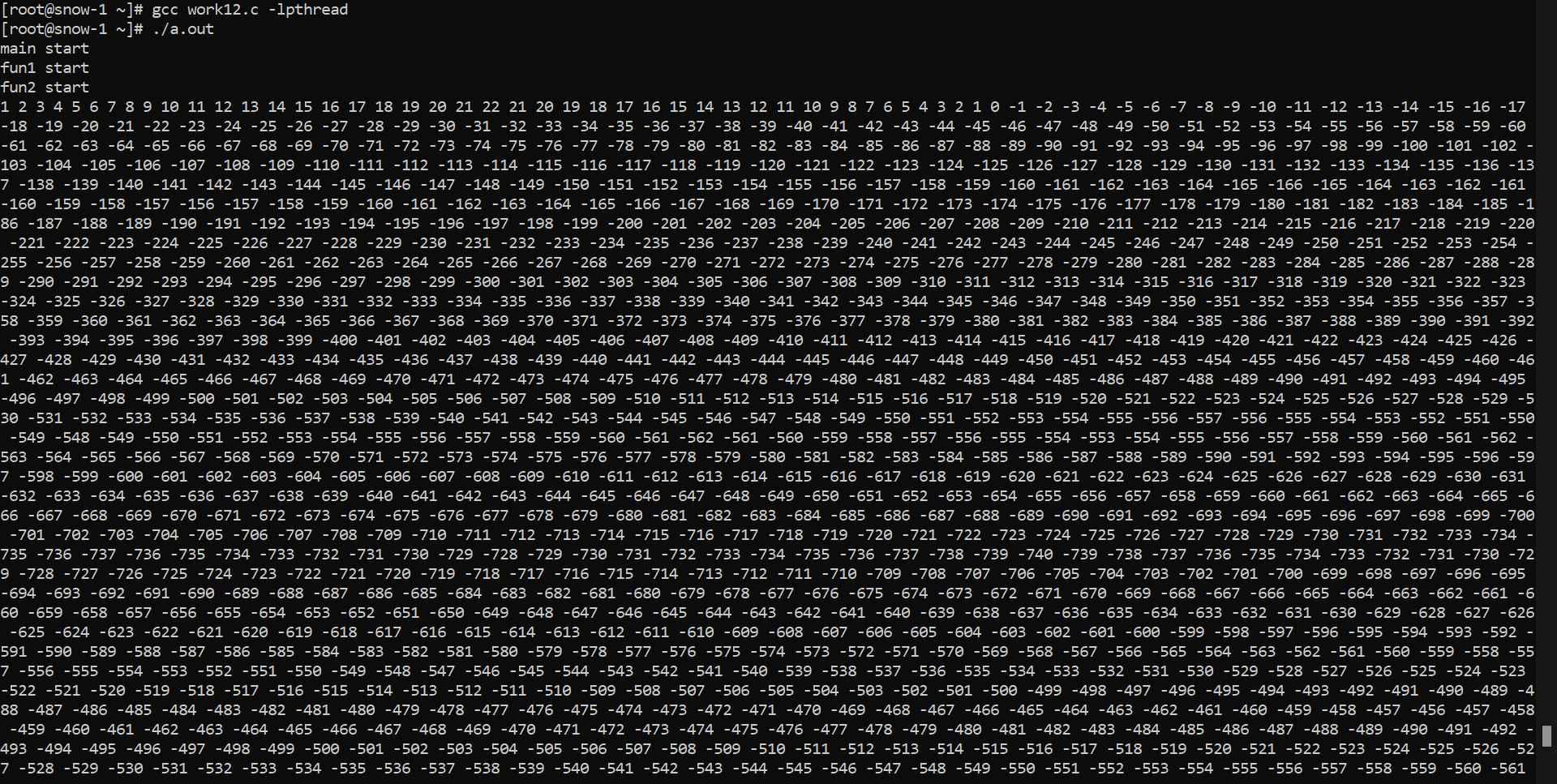


可以发现线程一输出总为一个小于5000的正数，线程二输出为0。

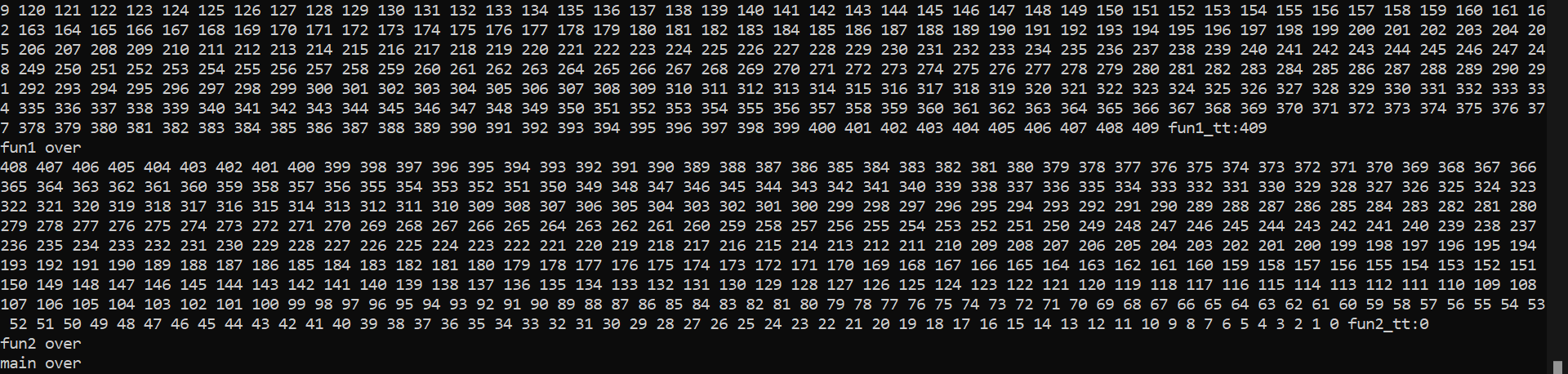
修改程序后，实现了给变量tt加锁：



加锁后，实现了互斥，得到了正确结果。



And so on



#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<string.h>

#include<unistd.h>

#include<pthread.h>

int tt;

//声明一个线程函数

void \*fun1(void \*);

void \*fun2(void \*);

pthread\_mutex\_t mutex;//定义互斥变量

int main()

{

printf("main start\n");

tt=0;

pthread\_mutex\_init(&mutex, NULL);//互斥变量初始化

pthread\_t id1;

pthread\_t id2;

//创建函数线程，并且指定函数线程要执行的函数

int res1 = pthread\_create(&id1,NULL,fun1,NULL);

int res2=pthread\_create(&id2,NULL,fun2,NULL);

//assert(res2==0);

sleep(2);

printf("main over\n");

pthread\_mutex\_destroy(&mutex);//销毁锁

exit(0);

}

//定义线程函数

void\* fun1(void \*arg)

{

printf("fun1 start\n");

//pthread\_mutex\_lock(&mutex);//加锁

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

{

pthread\_mutex\_lock(&mutex);//加锁

tt++;

//printf("fun1running\n");

pthread\_mutex\_unlock(&mutex);//解锁

}

//pthread\_mutex\_unlock(&mutex);//解锁

printf("fun1\_tt:%d\n",tt);

printf("fun1 over\n");

}

void\* fun2(void \*arg)

{

printf("fun2 start\n");

//pthread\_mutex\_lock(&mutex);//加锁

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

{

pthread\_mutex\_lock(&mutex);//加锁

tt--;

//printf("fun2running\n");

pthread\_mutex\_unlock(&mutex);//解锁

}

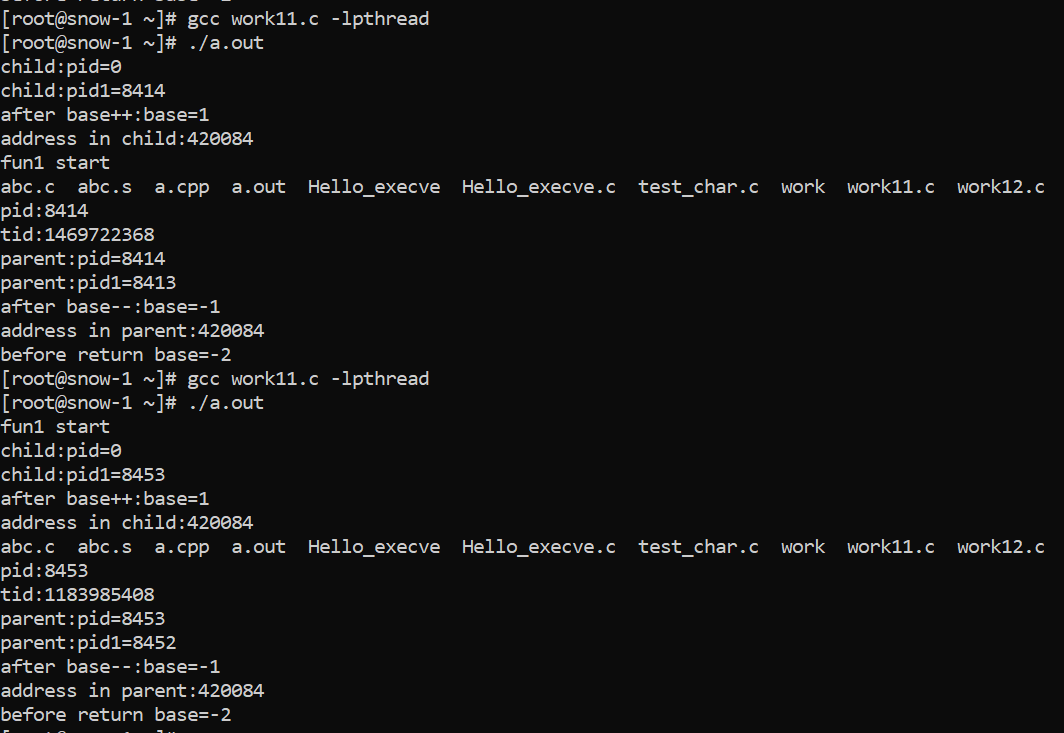
//pthread\_mutex\_unlock(&mutex);//解锁

printf("fun2\_tt:%d\n",tt);

printf("fun2 over\n");

}

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：



分析：

调用getpid()函数得到的是进程的pid，而在内核中，每个线程都拥有自己单独的pid，要得到线程的pid，必须用syscall(SYS\_gettid)函数，而调用pthread\_self()获取的是线程ID，其仅在同一个进程中保证唯一。在main()函数中，syscall(SYS\_gettid)的值和getpid()得到的值相同，在子线程中却不相同。

关于的system()函数和exec族函数的分析与进程中相同，调用exec族函数可以把当前线程替换为一个拥有相同pid的新线程。

进程pid：

进程pid（进程ID）,每个进程在系统中都有一个唯一·的非负整数表示的进程ID，用getpid() 获取进程ID。

线程tid：

线程tid（线程ID），每个线程在所属进程中都有一个唯一的线程ID，用pthread\_self() 获取自身现成ID。有多个进程时，可能会出现多个线程ID相同的线程，故线程tid只在其所属的进程上下文中有意义，不能作为系统中某个线程的唯一标识符。

线程pid：

线程pid，每个线程在系统中都有一个唯一的pid标识符，用系统调用sys\_call(SYS\_gettid()) 获取自身线程pid。主线程pid与所在进程pid相同。