1.1 进程

步骤一：

```
[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33700parent: pid =33700parent: pid1 =33699[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33702parent: pid =33702parent: pid1 =33701[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33704parent: pid =33704parent: pid1 =33703[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33706parent: pid =33706parent: pid1 =33705[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33708parent: pid =33708parent: pid1 =33707[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33710parent: pid =33710parent: pid1 =33709[root@kp-test01 src]#
```

步骤二：

去掉 wait 后：

```
[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33894parent: pid =33894parent: pid1 =33893[root@kp-test01 src]# ./test
parent: pid =33897parent: pid1 =33896child: pid =0child: pid1 =33897[root@kp-test01 src]# ./test
parent: pid =33899parent: pid1 =33898child: pid =0child: pid1 =33899[root@kp-test01 src]# ./test
parent: pid =33901parent: pid1 =33900child: pid =0child: pid1 =33901[root@kp-test01 src]# ./test
parent: pid =33903parent: pid1 =33902child: pid =0child: pid1 =33903[root@kp-test01 src]# ./test
parent: pid =33905parent: pid1 =33904child: pid =0child: pid1 =33905[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33907parent: pid =33907parent: pid1 =33906[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33909parent: pid =33909parent: pid1 =33908[root@kp-test01 src]# ./test
parent: pid =33911parent: pid1 =33910child: pid =0child: pid1 =33911[root@kp-test01 src]# ./test
parent: pid =33913parent: pid1 =33912child: pid =0child: pid1 =33913[root@kp-test01 src]# ./test
parent: pid =33915parent: pid1 =33914child: pid =0child: pid1 =33915[root@kp-test01 src]# ./test
parent: pid =33917parent: pid1 =33916child: pid =0child: pid1 =33917[root@kp-test01 src]# ./test
parent: pid =33919parent: pid1 =33918child: pid =0child: pid1 =33919[root@kp-test01 src]# ./test
parent: pid =33921parent: pid1 =33920child: pid =0child: pid1 =33921[root@kp-test01 src]# ./test
parent: pid =33923parent: pid1 =33922child: pid =0child: pid1 =33923[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33925parent: pid =33925parent: pid1 =33924[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33927parent: pid =33927parent: pid1 =33926[root@kp-test01 src]# ./test
```

分析：父进程创建子进程后，父子进程进入 if 语句执行各自的代码，父进程和子进程的执行顺序没有先后之分，完全随机，所以在去掉 wait 后的多次运行结果中 child 和 parent 出现的先后顺序是随机的。

步骤三：

增加全局变量 value：

```
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
[root@kp-test01 src]#
```

在 if 语句的 else 部分设置 value--,在 else if(pid==0)部分设置 value++,在这两部分的 printf 中都输出 value 的值和地址，随后运行程序，child 中的 value 为 1，parent 中的 value 为-1,且 child 中 value 的地址和 parent 中 value 的地址相同，这是因为父进程在创建子进程时子进程完全复制了一份父进程的数据，输出 value 的地址时输出的其实是 value 在进程中的虚拟地址，因为是复制的缘故，两个 value 在父进程和子进程中的虚拟地址是一样的，但是映射到内存中的物理地址是不同的，所以在一个进程中修改 value 不会影响到另一个进程中的 value

步骤四：

```
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
 before return value=4,*value=0x420054
 before return value=6,*value=0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
 before return value=6,*value=0x420054
 before return value=4,*value=0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
 before return value=6,*value=0x420054
 before return value=4,*value=0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
 before return value=4,*value=0x420054
 before return value=6,*value=0x420054
```

在 return 语句之前、if 语句之后添加一段对全局变量 value 的操作代码，父进程和子进程在执行完 if 语句之后都会执行这一段代码，实现对 value 的操作后打印出各自的 value 值，因为父进程和子进程的执行顺序是完全随机的，所以 child 和 parent 出现的先后顺序也无法确定，如截图中第一次执行是 chlid 在前，第二次执行是 parent 在前，第三次执行是 chlid 在前

步骤五：

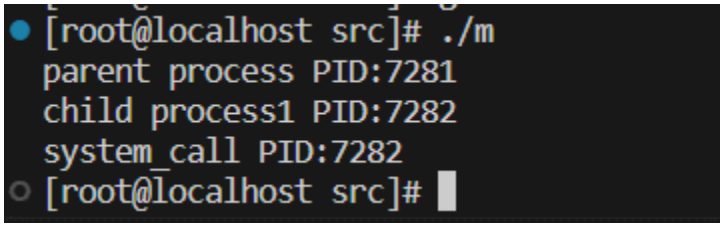
子进程调用system函数

```
[root@localhost src]# ./m
parent process PID:7316
child process1 PID:7317
system_call PID:7318
child process PID:7317
[root@localhost src]#
```

子进程调用 system 函数时执行了 system_call 文件，可以发现在执

行 system_call 文件时进程的 pid 和子进程的 pid 并不一样，这是因为 system 函数是在原进程上创建了一个新的进程。system 函数执行完之后会返回原进程继续执行后续代码。

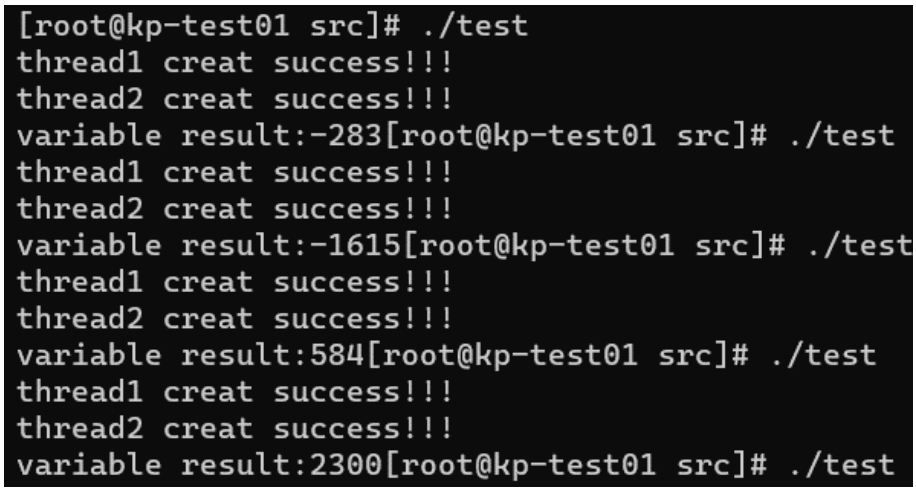
子进程调用 execl 函数



子进程调用 execl 函数时执行了 system_call 文件，可以发现在执行 system_call 文件时进程的 pid 和子进程的 pid 是一样的，并且执行完 excel 函数之后并没有后续输出，说明子进程在执行 excel 函数时并没有创建新进程，并且执行 excel 函数后并没有执行后续的 printf 代码，这是因为 exec 是用新进程(命令)覆盖了原有的进程

1.2 线程

步骤一：



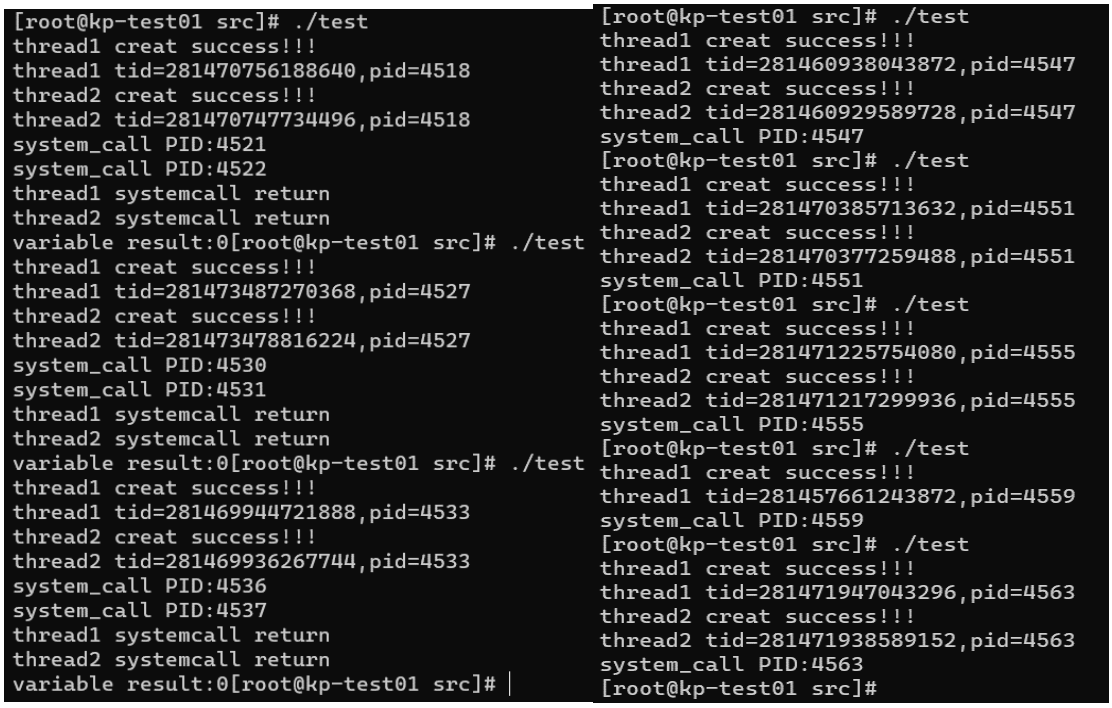
由于线程并发执行，存在竞态条件，即两个线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。

步骤二：

```
[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]#
```

通过设置信号量以及对信号量的 PV 操作使得任意一个时刻最多只有一个线程能够对 value 进行修改，避免了多线程之间的竞争，不管重复多少次，value 最终的值都是恒定为 0

步骤三：



左图是有 system 函数的线程运行的结果，有图是有 exec 函数的线程运行的结果，可以发现线程执行 system 函数后会返回原线程并继续执行后续代码，而线程执行 exec 函数后直接结束，并不会继续执行后续代码，这是因为 system 函数是在原进程上开辟了一个新的进程，而 exec 函数是用新进程(命令)覆盖了原有的进程

1.3 自旋锁

```
[root@kp-test01 src]# ./spinlock
shared_value:0
thread1 create success!
thread2 create success!
shared_value:10000
[root@kp-test01 src]# ./spinlock
shared_value:0
thread1 create success!
thread2 create success!
shared_value:10000
[root@kp-test01 src]# ./spinlock
shared_value:0
thread1 create success!
thread2 create success!
shared_value:10000
[root@kp-test01 src]# ./spinlock
shared_value:0
thread1 create success!
thread2 create success!
shared_value:10000
[root@kp-test01 src]# ./spinlock
shared_value:0
thread1 create success!
thread2 create success!
shared_value:10000
```

在线程调用全局变量 shared_value 的前后分别使用了自旋锁的获取函数和释放函数，且在调用这两个函数的时候不会被中断，以此来保证在任意一个时刻 shared_value 最多只能被一个线程所修改，这就消除了多线程的竞争状态，实现了多线程的同步，最终 shared_value 的值跟预期一样是 10000.

2.困难：

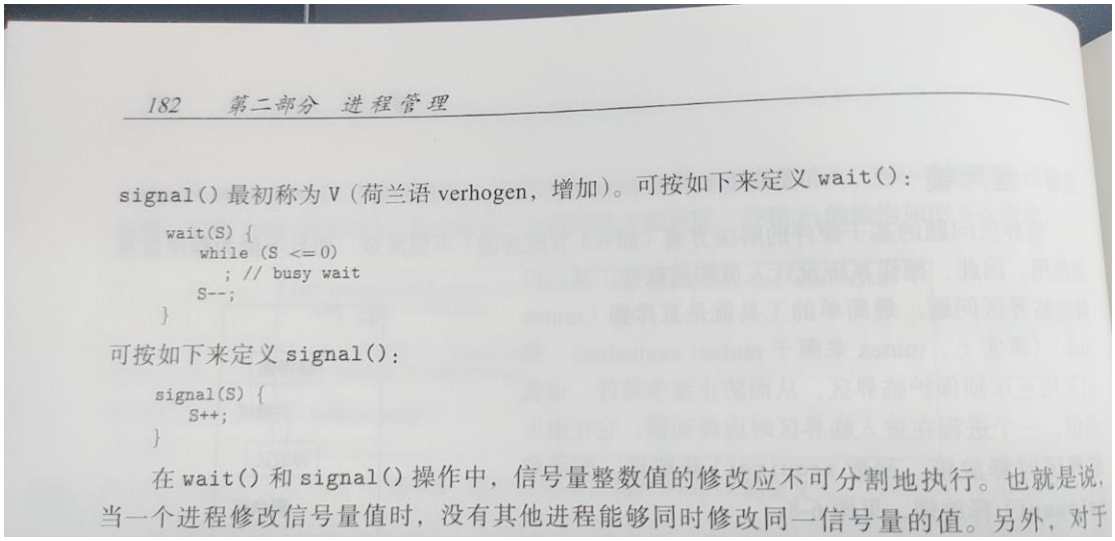
①在遇到 system 和 exce 函数时一头雾水，在看到 system 的介绍 int system(const char *__command)时一直没弄明白这个 command 命令具体指代的是什么，后面看了具体实例后发现是需要执行文件所对应的目录

Linux 下使用 system（）和 execv（）实现对外部程序的调用_linux 采用 system 调用另一个进程-CSDN 博客

②在完成实验一步骤三的过程中发现，父进程和子进程中全局变量 value 的地址是一样的，可是对应的 value 的值却不相同，当时感到很不解，因为在以前一直都认为地址相同对应的是同一个量，后面发现 printf(“%p”,&value);打印的只是 value 在进程中的虚拟地址，而相同的虚拟地址在不同的进程中会映射到不同的物理地址上面，子进程的创建是复制了父进程的虚拟地址空间的

父子进程能否共享全局变量_牛客博客 (nowcoder.net)

③



182　第二部分　进程管理

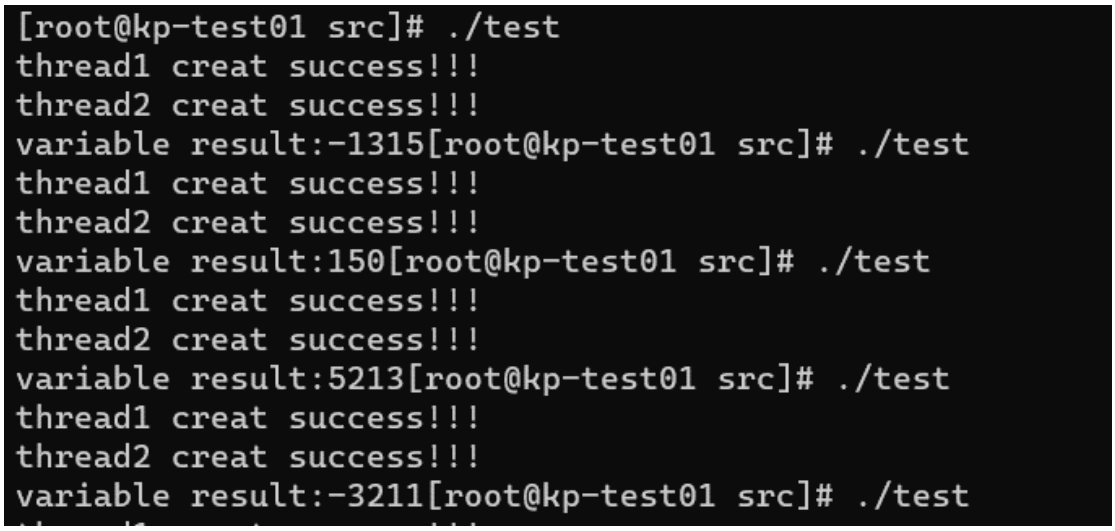
signal() 最初称为 V（荷兰语 verhogen，增加）。可按如下来定义 wait()：

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

可按如下来定义 signal()：

```
signal(S) {
    S++;
}
```

在 wait() 和 signal() 操作中，信号量整数值的修改应不可分割地执行。也就是说，当一个进程修改信号量值时，没有其他进程能够同时修改同一信号量的值。另外，对于



在给多线程程序实现互斥锁的时候，我一开始想到的是课本 6.6 节有关互斥锁实现的内容，简单实现之后运行程序，发现结果还是和没有互斥锁时的结果一样，这代表我的互斥锁是不合格的，在查阅一些资料后发现我实现的这个互斥锁有可能在运行过程中被中断，于是我选用了教材 6.9.4 节 Pthreads 同步章节所介绍的 unnamed semaphore 作为信号量，最终成功解决问题。