

201220180 李全昊 Oslab4 实验报告

一 . exercise1: 请回答一下, 什么情况下会出现死锁。

如果比如说五个哲学家都同时拿起了自己左边的叉子(也就是手册上的程序每个人都执行到第六行然后进程切换到下一个哲学家), 这样的话, 就会造成没有人可以拿起右边的叉子开始吃面, 也就是我们说的死锁现象。

二 . exercise2: 说一下该方案有什么不足? (答出一点即可)

并发程度过低(效率太差), 因为他这样相当于串行了, 因为有一个互斥信号量的存在, 每个哲学家除了思考之外的所有过程都必须串行进行。(拿叉子、吃面、放叉子)

三 . exercise3: 正确且高效的解法有很多, 请你利用信号量 PV 操作设计一种正确且相对高效(比方案 2 高效)的哲学家吃饭算法。(其实网上一堆答案, 主要是让大家多看看不同的实现。)

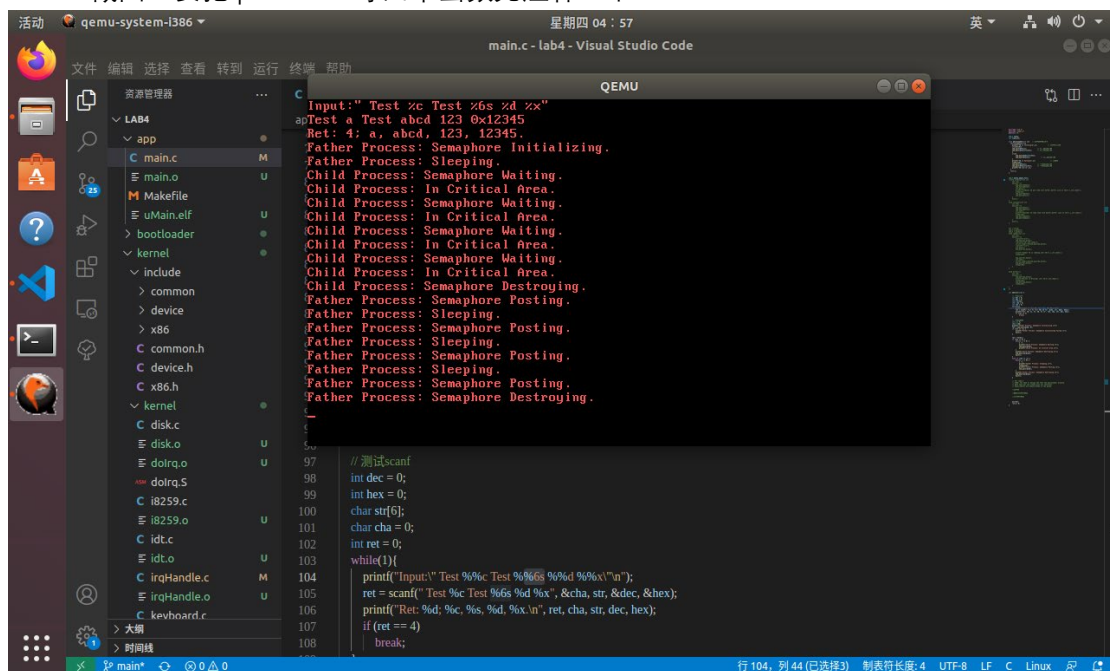
设计方法很简单: 最多只允许四个哲学家进行拿叉子、吃面、放叉子的过程, 也就是把手册代码的 mutex 初值设置为 4 就可以了。

四 . exercise4: 为什么要用两个信号量呢? emptyBuffers 和 fullBuffer 分别有什么直观含义?

emptyBuffers 代表的是缓冲区现在还可以放的元素的个数(空下来的位置数); fullBuffer 代表的是现在我可以从缓冲区取出的元素个数(就是缓冲区内元素个数)

因为生产和消费需要看的分别是 buffer 是否已满和 buffer 是否已空, 是两个含义, 所以需要两个信号量。

Task1 截图: 要把 producer 等四个函数先注释一下



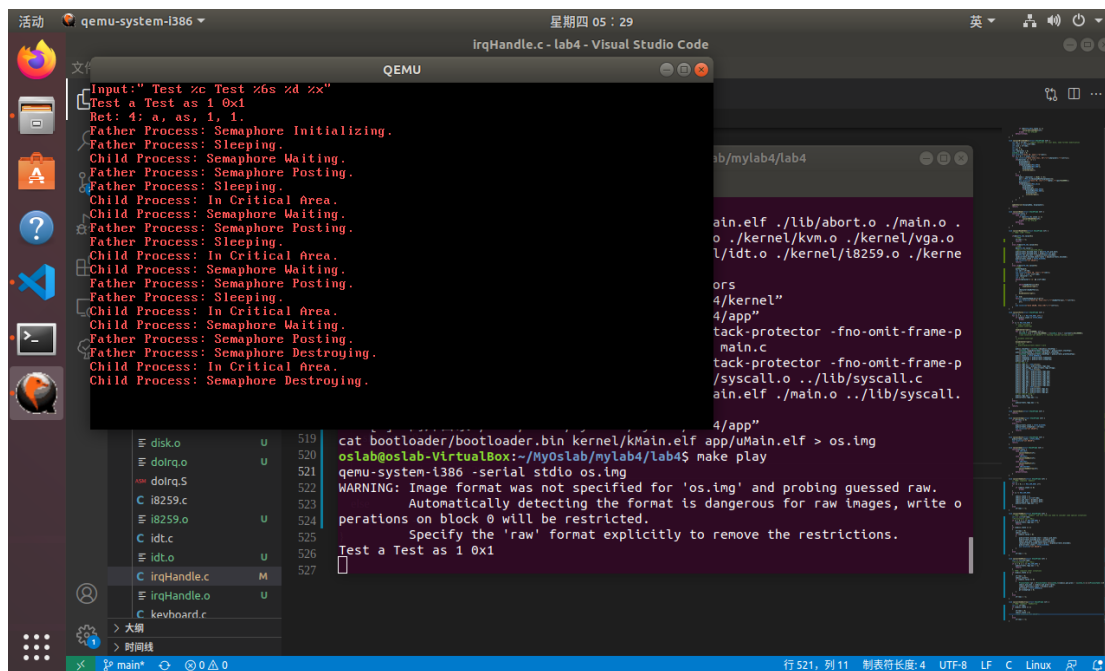
```
Input: " Test %c Test %6s %d %x\n"
apTest a Test abcd 123 0x12345
Ret: 4: a, abcd, 123, 12345.
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.

// 测试scanf
int dec = 0;
int hex = 0;
char str[6];
char cha = 0;
int ret = 0;
while(1){
    printf("Input: " Test %c Test %6s %d %x\n");
    ret = scanf(" Test %c Test %6s %d %x", &cha, str, &dec, &hex);
    printf("Ret: %d: %c, %s, %d, %x\n", ret, cha, str, dec, hex);
    if (ret == 4)
        break;
```

可以看到这里 child process 没等到 parent process 的 V 操作, 就开始进入临界区了, 显然这是不合适的, 于是我们下面实现了信号量 PV 操作有关的函数。

解释一下在 keyboardhandle 函数中 dev[STD_IN].value=1;这句话是为了表明有一个进程占据了 STD_IN 这个设备了, dev 其实起到的是一个互斥信号量的作用

Task2 截图:

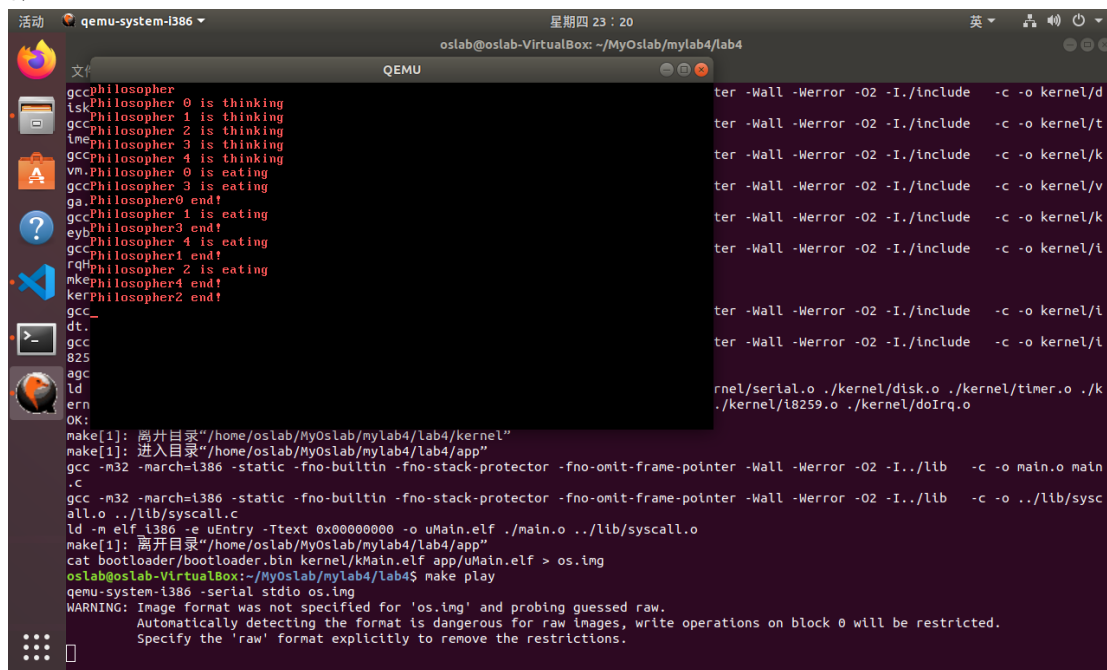


Task3:

(1) 哲学家问题

把每个人的 time 都设置为 1，（这样每个哲学家会 end1 次）然后在原来代码测试信号量的地方，不让父进程 exit，而是让它 sleep(1)之后让子进程 exit，进入哲学家测试阶段。

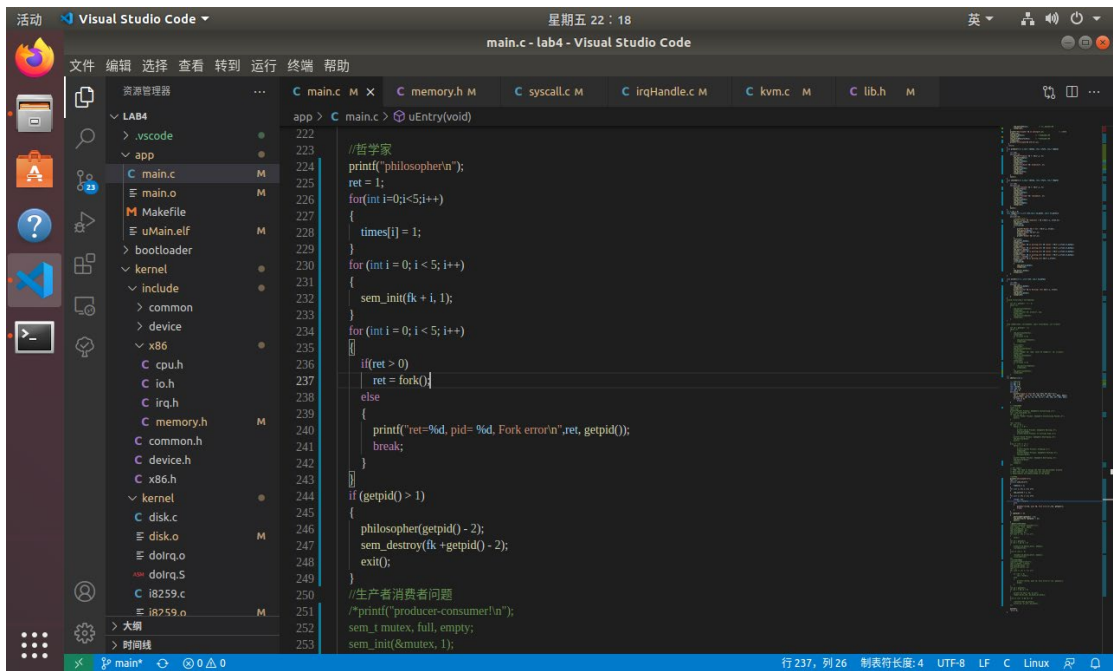
截图如下：



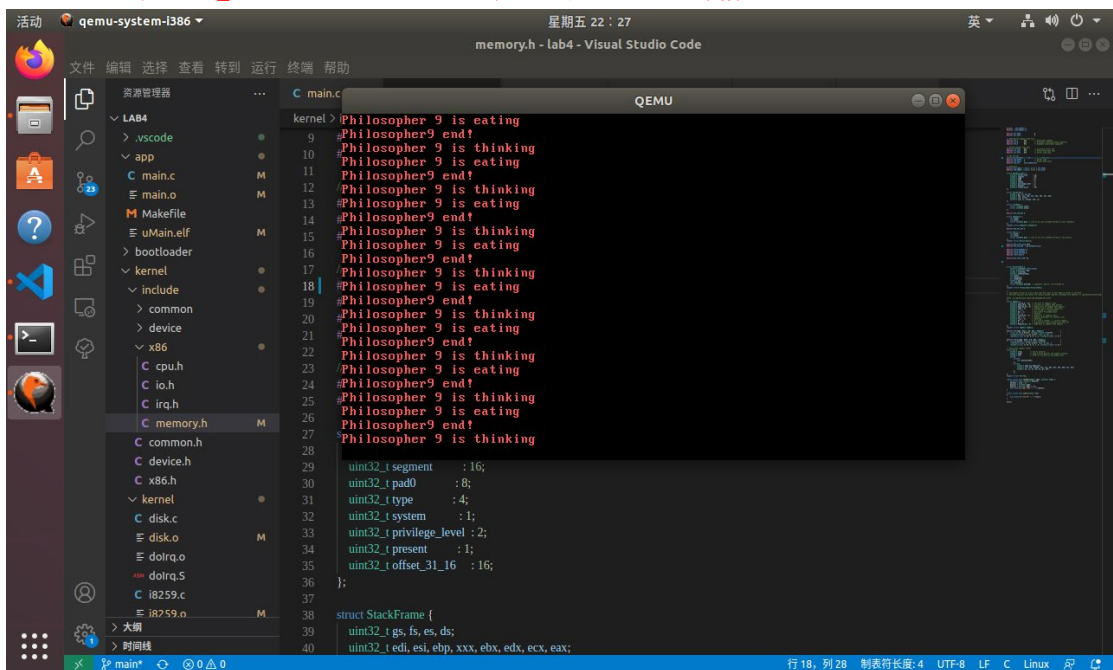
结果符合预期，每个哲学家 end 了 1 次。

问题 1：为啥这里我把 NR_SEGMENTS 改到 20 运行结果会改变呢？？？

答：见下图的第 236、237 行（本来 236 行我是大于等于 0，这里一定要改成大于 0，用来限制只有主进程调用 fork 函数！），我们这边生成的子进程其实也会再次进到这个 for 循环里面执行 fork 系统调用，我们必须限制 MAX_PCB_NUM 在 16（也就是不能多于 2+5 个进程），否则就会生成多于 7 个进程，会出乱子的。



比如我尝试把 NR_SEGMENT 设置成 100，就会出现下面的情形



所以我们一定要把大于等于号改成大于号，这样就不会出乱子了（也就是 fork 出来的子进程不会再去执行 fork 函数）

(2) 生产-消费者问题

```
oslab@oslab-VirtualBox: ~/MyOslab/mylab4/lab4
gccProducer-consumer!
lskProducer 2 t: 4
gccProducer 3 t: 4
gccProducer 4 t: 4
lmeProducer 5 t: 4
gccConsumer 6 t: 4
vmProducer 2: produce
gccProducer 3: produce
gaProducer 2 t: 3
gccProducer 4: produce
eybProducer 3 t: 3
gccProducer 5: produce
rqfProducer 4 t: 3
kerConsumer 6: consume
kerProducer 5 t: 3
kerProducer 2: produce
gccConsumer 6 t: 3
dtProducer 3: produce
gccProducer 2 t: 2
825Consumer 6: consume
gccProducer 3 t: 2
ldConsumer 6 t: 2
erProducer 4: produce
erConsumer 6: consume
OK:
make[1]: 离开目录"/home/oslab/MyOslab/mylab4/lab4/kernel"
make[1]: 进入目录"/home/oslab/MyOslab/mylab4/lab4/app"
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-pointer -Wall -Werror -O2 -I../include -c -o kernel/d
ter -Wall -Werror -O2 -I../include -c -o kernel/t
ter -Wall -Werror -O2 -I../include -c -o kernel/k
ter -Wall -Werror -O2 -I../include -c -o kernel/v
ter -Wall -Werror -O2 -I../include -c -o kernel/k
ter -Wall -Werror -O2 -I../include -c -o kernel/i
ter -Wall -Werror -O2 -I../include -c -o kernel/i
ter -Wall -Werror -O2 -I../include -c -o kernel/i
rnel/serial.o ./kernel/disk.o ./kernel/timer.o ./k
./kernel/i8259.o ./kernel/doIrq.o
make[1]: 离开目录"/home/oslab/MyOslab/mylab4/lab4/app"
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
oslab@oslab-VirtualBox:~/MyOslab/mylab4/lab4$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

```
oslab@oslab-VirtualBox: ~/MyOslab/mylab4/lab4
gccProducer 3 t: 3
lskProducer 5: produce
gccProducer 4 t: 3
gccConsumer 6: consume
lmeProducer 5 t: 3
gccProducer 2: produce
vmConsumer 6 t: 3
gccProducer 3: produce
gaProducer 2 t: 2
gccConsumer 6: consume
eybProducer 3 t: 2
gccProducer 4: produce
rqfConsumer 6: consume
kerProducer 4 t: 2
kerConsumer 6 t: 1
gccProducer 5: produce
dtConsumer 6: consume
gccProducer 5 t: 2
825Producer 2: produce
gccConsumer 6: consume
ldProducer 2 t: 1
erProducer 3: produce
OK:
make[1]: 离开目录"/home/oslab/MyOslab/mylab4/lab4/kernel"
make[1]: 进入目录"/home/oslab/MyOslab/mylab4/lab4/app"
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-pointer -Wall -Werror -O2 -I../lib -c -o main.o main
.c
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-pointer -Wall -Werror -O2 -I../lib -c -o ../lib/sysc
all.o ../lib/syscall.c
ld -m elf_i386 -e uEntry -Ttext 0x00000000 -o uMain.elf ./main.o ../lib/syscall.o
make[1]: 离开目录"/home/oslab/MyOslab/mylab4/lab4/app"
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
oslab@oslab-VirtualBox:~/MyOslab/mylab4/lab4$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

注意这里同样也要改一下 NR_SEGMENTS 到 16，否则无法容纳那么多（共 7 个）进程

(3) 读者写者问题（经过多次尝试还是没法解决，算了那就做一个选做好了。。）

我这里出现了 bug，为什么我在一个 reader 里面加了 readcount，但是在另一个 reader 进程中显示并没有呢？

问题在于这里 readcount 虽然我传的是指针，地址是一样的，但由于不同进程的段寄存器的值可能都不一样，所以相当于这些进程不共享 readcount 了，也就无法正确实现读写问题了

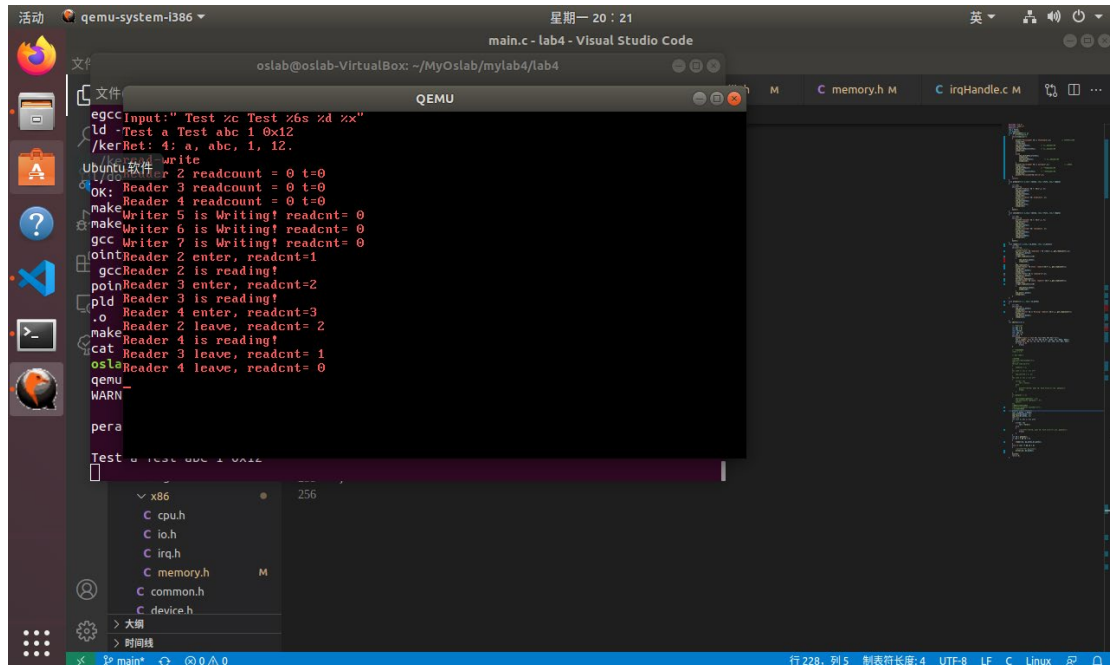
那么问题来了怎么来实现进程间的数据共享呢？我们可以结合下面“我提出的问题”板块的第二个问题看到，信号量是每个进程共享的，为什么呢？因为 sem 数组是开在内核空间的，每个进程共享。那么自然而然的我就想到是不是可以把 read count 也开在内核空间里面呢？进行实验之后，我发现确实是可以的！

我把 readcount 开在了 kvm.c 中也就是内核空间里面，初始值设定为 0，然后自己定义

了三个系统调用，分别是获取 readcount 的值，让 readcount 的值加一或者减一。
(get_readcount, add_readcount 和 decrease_readcount 函数)

经过以上的操作之后，就可以成功解决读写者问题了！

运行结果如下（我把这里的 t 置为 1 了，助教 gg 也可以去把读写者函数中的 t 改为比如 5）



```
egccInput: "Test %c Test %6s %d %x"
ld -Test a Test abc 1 0x12
/kerRet: 4: a, abc, 1, 12.
write
Reader 2 readcount = 0 t=0
OK: Reader 3 readcount = 0 t=0
make Writer 5 is Writing! readcnt= 0
make Writer 6 is Writing! readcnt= 0
gcc Writer 7 is Writing! readcnt= 0
ointReader 2 enter, readcnt=1
gccReader 2 is reading!
pointReader 3 enter, readcnt=2
pid Reader 3 is reading!
.o Reader 4 enter, readcnt=3
make Reader 2 leave, readcnt= 2
make Reader 4 is reading!
cat Reader 3 leave, readcnt= 1
oslaReader 4 leave, readcnt= 0
qemu
WARN
pera
Test a Test abc 1 0x12
```

一些我自己提出的问题：

1. 最后一句为啥可以 exit(0)，他的作用是什么？？直接 return 不能终止主进程吗？？？

答：我估计这里 exit 接收一个参数的时候就用的不是我们自己写的 exit 函数了，是 C 库里的函数了，自动将当前进程 destroy，我试了一下用 exit()也是可以的。

exit 用于结束正在运行的整个程序，它将参数返回给 OS，把控制权交给操作系统；而 return 是退出当前函数，返回函数值，把控制权交给调用函数。如果 return 的是 main 函数，会隐式调用 exit 函数。

在这里我尝试了一下，把 uEntry 返回值改成 void 类型，然后把 return 0 去掉，只留下 exit 函数，也可以正常运行。原因是 exit 函数只会终止当前进程，而不会终止自己的父子进程，所以我们 fork 出来的子进程还是可以运行的。

但如果把 exit 去掉只留下 return，就无法正常运行了。事实上我尝试下来有一个非常神奇的现象！（见下图）（我在哲学家函数开始加了一句显示当前 pid 的 print 语句）

