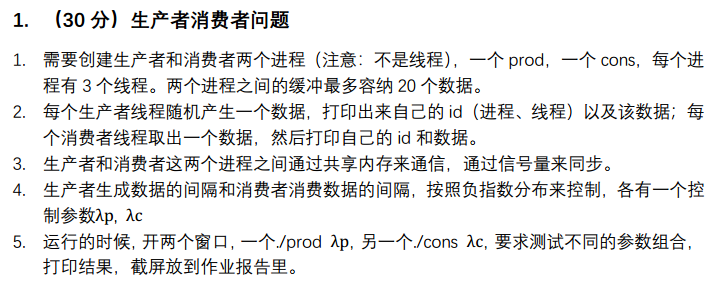
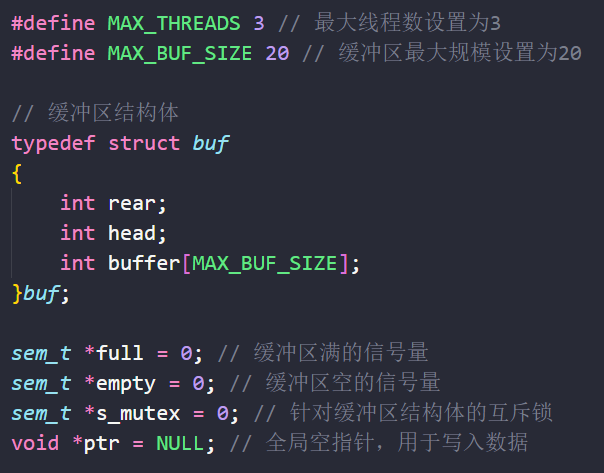
操作系统第一次大作业 实验报告

张灿晖 18364114 智科2班

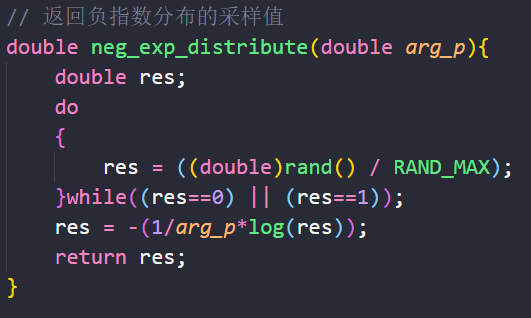
# 生产者消费者问题

**prod.c代码：**

在代码开头，预定义了一些常量宏、结构体、信号量和指针：



函数neg\_exp\_distribute为负指数分布采样函数，返回值为生产者生产数据之间的间隔时间：



函数prod\_func为生产者线程核心函数。该函数先将传入的参数arg\_p\_ptr空指针转为双精度浮点数，调用函数pthread\_self获取当前线程id，存储于变量temp\_thread中。

接下来，进入条件为1的while循环，先调用函数neg\_exp\_distribute获取本次写入数据的时间间隔，再调用函数usleep让线程休眠这段时间间隔。接下来，以函数rand为基础生成需要写入共享内存的随机数据，并存储于整形变量item中。

随后，将全局作用域中声明的指向void的指针ptr强制转型为指向共享内存区域的指针shared\_mem\_ptr，准备进行数据写入。在写入之前，要等待缓冲区空的信号量empty，并获取互斥锁s\_mutex。当准备工作就绪后，写入数据的过程也就是对循环队列数组buffer的尾端rear赋值的过程。写入完成后，打印写入的缓存区数组下标信息、线程标识符和写入的整形数据值，随后释放互斥锁并发出缓冲区满的信号量，等待消费者进程读取数据。

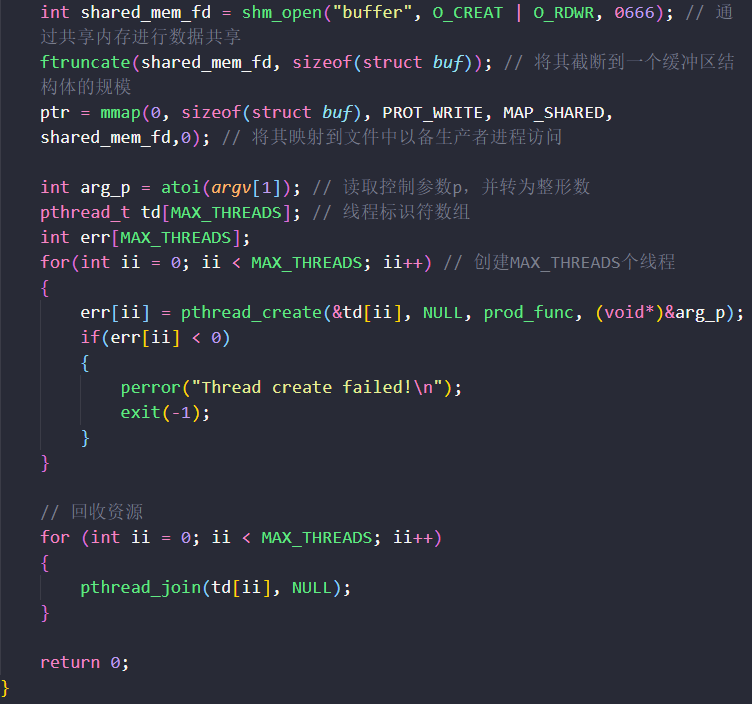


主函数main完成了变量的初始化和线程的创建工作。函数main的函数体之中，先创建了一个缓冲区的结构体并调用函数memset对其进行初始化。随后连续三次调用函数sem\_open打开信号量full、empty、s\_mutex，并紧接着调用函数sem\_init对它们进行初始化。

随后，调用函数shm\_open打开一个共享内存区域，并调用函数ftruncate对其进行截断，使其容量刚好为一个缓冲区结构体buf的规模，再调用函数mmap将其映射到文件中，以备后续进行写入和读取：



接下来，main函数要进行线程的创建，其核心就是通过for循环调用函数pthread\_create并传入控制参数p：



**cons.c代码：**

由于cons.c中预定义的数据和main函数的主要流程都与prod.c类似，这里就不再赘述，仅对消费者程序中的核心线程函数cons\_func作出说明。

函数cons\_func为生产者线程核心函数。该函数先将传入的参数arg\_c\_ptr空指针转为双精度浮点数，调用函数pthread\_self获取当前线程id，存储于变量temp\_thread中。

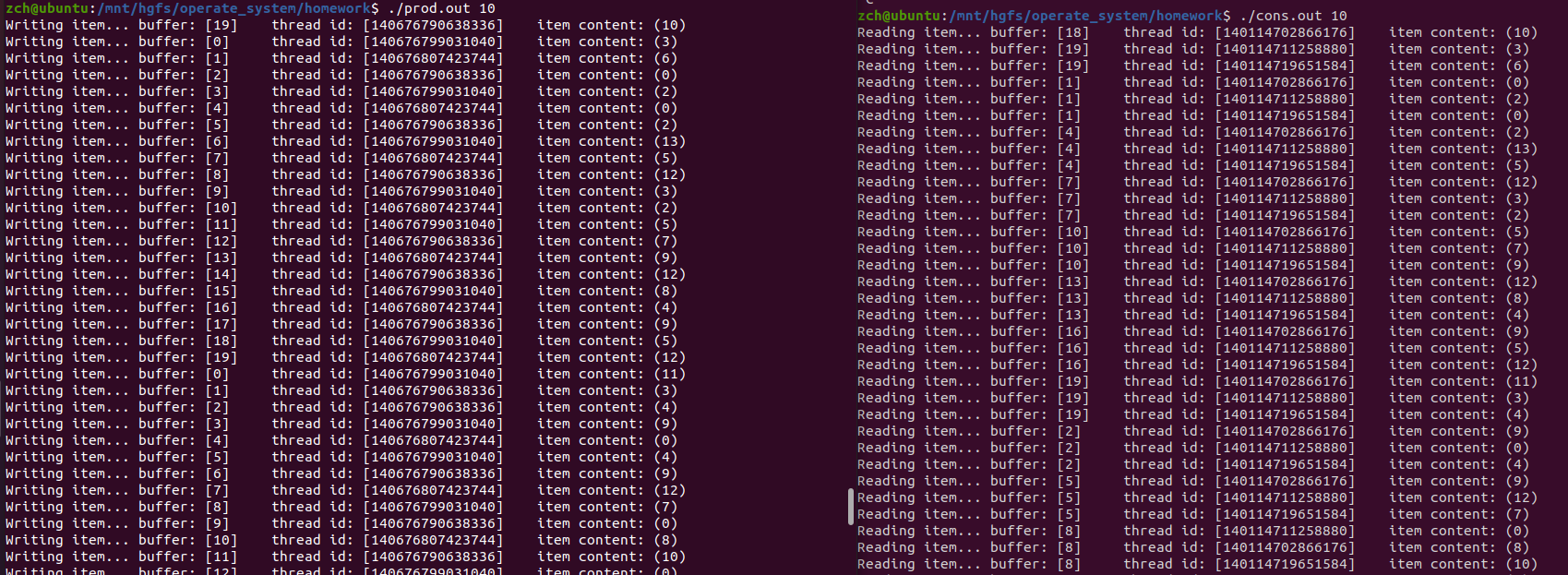
接下来，进入条件为1的while循环，先调用函数neg\_exp\_distribute获取本次写入数据的时间间隔，再调用函数sleep让线程休眠这段时间间隔。随后，将全局作用域中声明的指向void的指针ptr强制转型为指向共享内存区域的指针shared\_mem\_ptr，准备进行数据读取。在读取之前，要等待缓冲区满的信号量full，这说明生产者进程已经生产完毕，并获取互斥锁s\_mutex。当准备工作就绪后，读取数据的过程也就是提取循环队列数组buffer的头端head值的过程。

读取完成后，打印读取的缓存区数组下标信息、线程标识符和读取的整形数据值，随后释放互斥锁并发出缓冲区空的信号量，等待生产者进程继续生产数据。

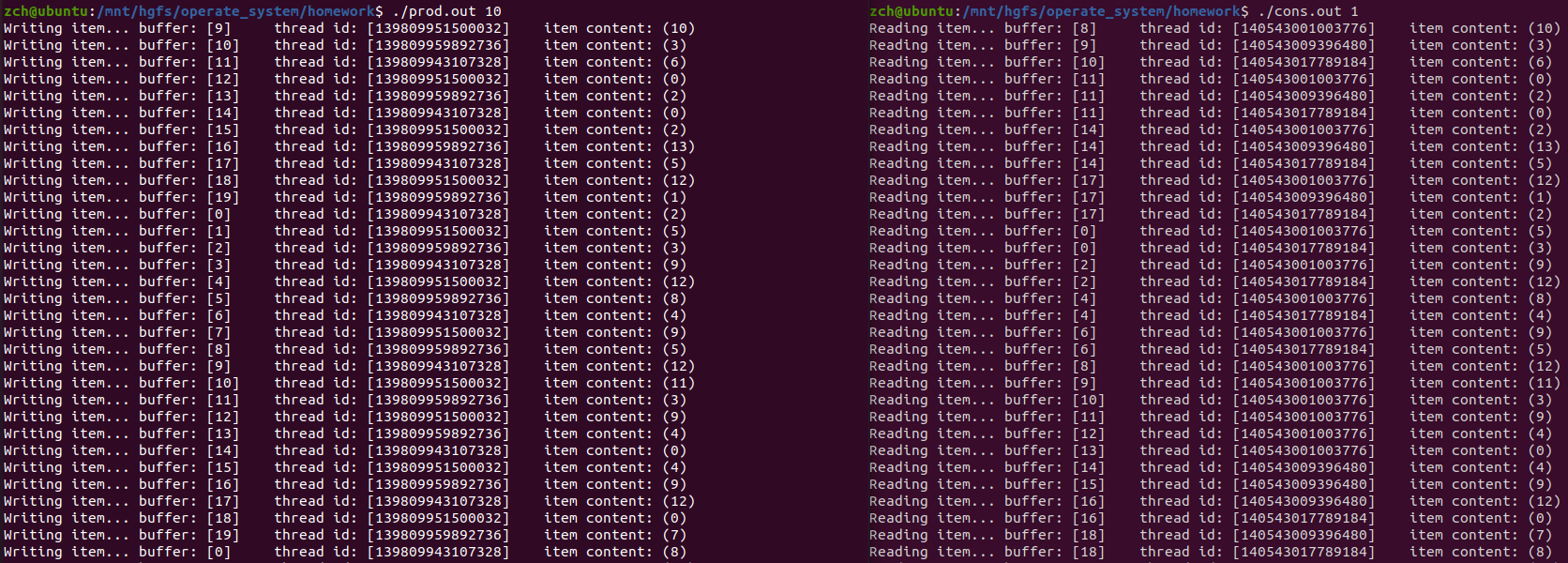


**运行截图：**

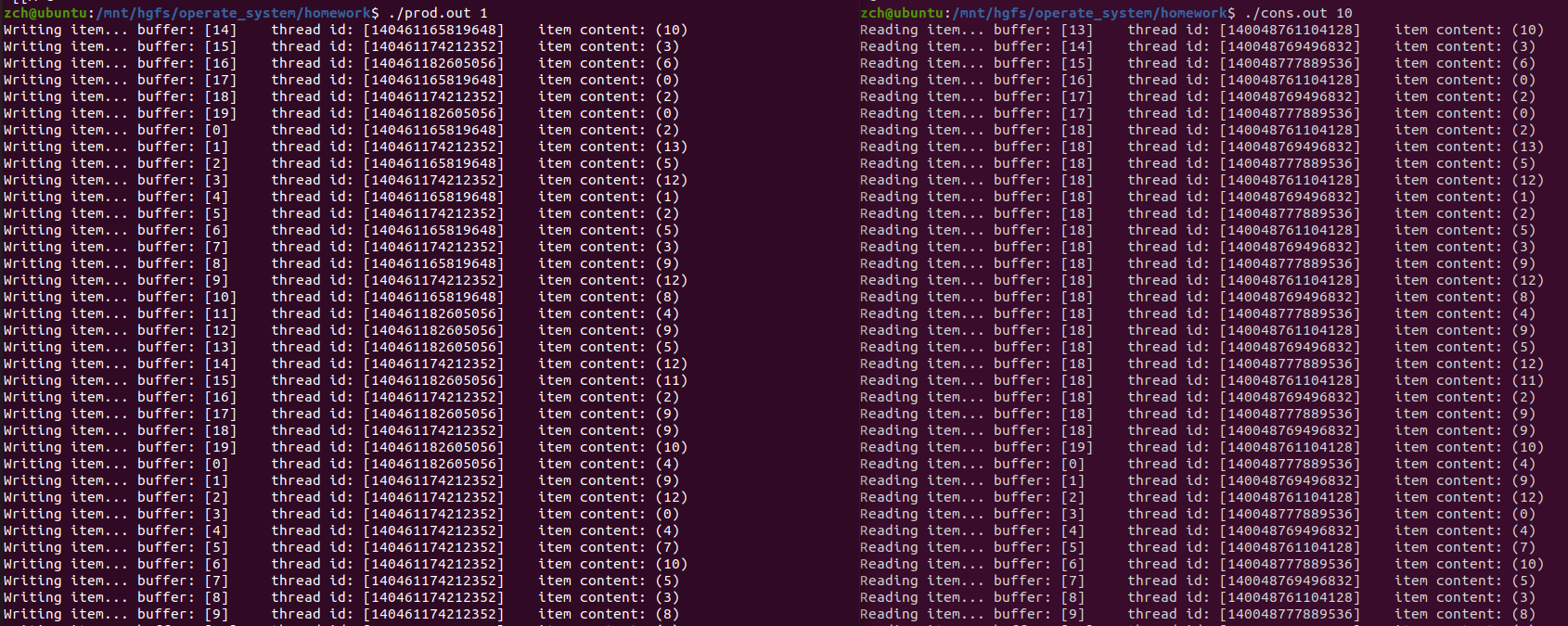
1.当控制参数p等于控制参数c时，即生产速率等于消费速率时，生产者进程生产数据与消费者消费数据同步进行，总的效率最高：



2.当控制参数p大于控制参数c时，即生产速率小于消费速率时，消费者进程的消费要等待生产者进程生产数据才可进行，总的效率中等：

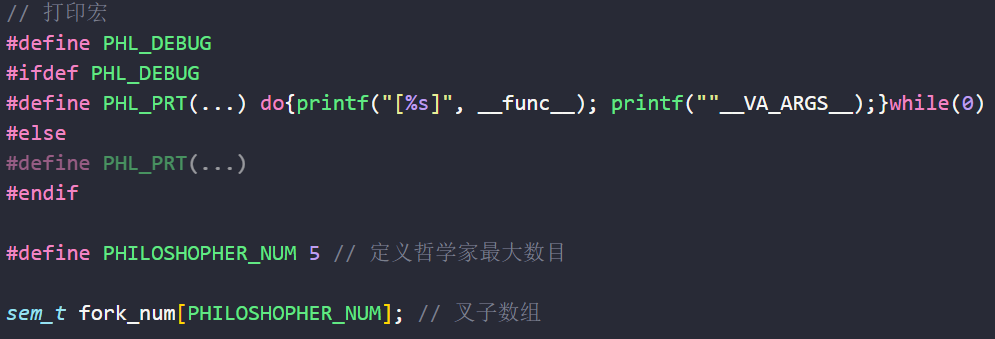


3.当控制参数p小于控制参数c时，即生产速率大于消费速率时，生产者进程的生产要等待消费者进程消费数据才可进行，总的效率中等：



# 哲学家就餐问题

先定义打印宏、哲学家最大数目常量和叉子数组（由POSIX信号量sem\_t组成）：



接下来定义的函数time\_of\_think\_or\_eat，用于生成哲学家思考或吃饭的时间，通过随机数函数rand为基础产生一个5s内的随机时间，然后sleep这个随机时间，以模拟思考或是吃饭。

接下来，定义了核心函数pickup\_forks，该函数先接受一个表征哲学家标识符的空指针，再在函数体中将其转化为整形变量，并取引用赋值给变量p\_id，由p\_id我们可以定义出表征左右手叉子的变量left\_fork和right\_fork，再定义变量left\_get和right\_get，以表征获取左右手的叉子是否成功。

接下来，函数进入while条件为1的无限循环，先让哲学家思考一段时间，即调用time\_of\_think\_of\_eat。在哲学家思考完毕后，进入吃饭环节，先分别尝试获取左右手的叉子，即调用POSIX函数sem\_trywait，尝试能否获取成功：

在上一步调用以后，我们就可以通过变量left\_fork和right\_fork的值判定左右手的叉子是否获取成功，倘若均为0，则表示都获取到了，可以开始吃饭，因此打印一条哲学家准备吃饭的信息，再调用函数time\_of\_think\_or\_eat模拟哲学家吃饭，吃完饭后哲学家要释放资源，即把叉子放回，以免造成死锁，因此调用POSIX函数sem\_post释放左右手叉子：

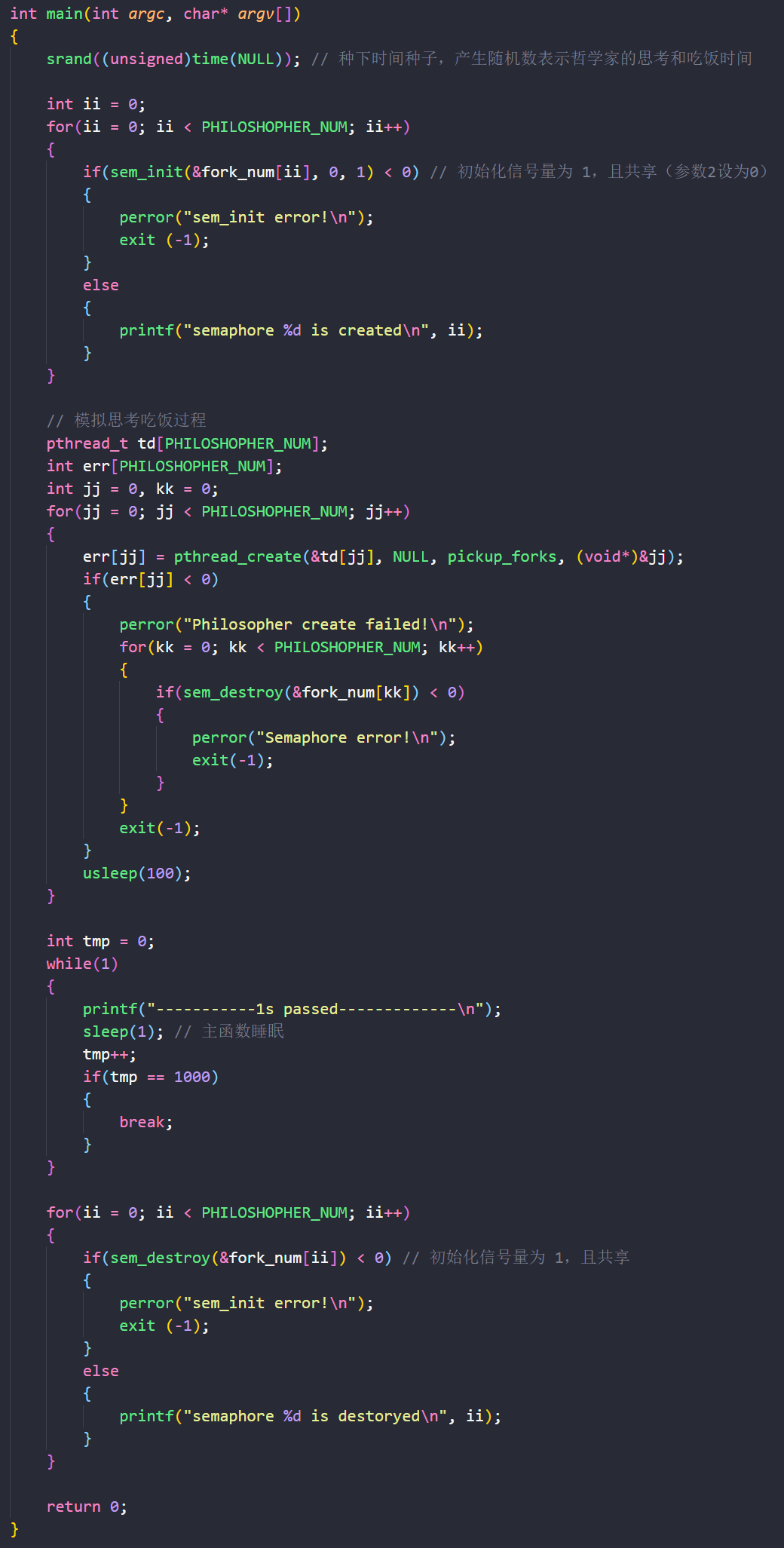
倘若哲学家没有获取到左右手的两只叉子，而是只获得了其中之一，或者干脆两只叉子都没获取到，则情况又有所不同，下面对着三种情况统一说明。两只叉子若都没有获取到，直接运行continue语句，让哲学家回到循环头部，继续思考；而如果左手或者右手的叉子获取成功，那么先调用函数sleep等候1ms再调用POSIX函数sem\_trywait尝试获取另一侧的叉子并将结果复制给对应get变量。倘若等待1ms后获取成功，则进行吃饭的步骤，并在吃完饭后释放两根叉子，返回循环头部继续思考。而倘若获取失败，则哲学家会释放已经获取到的左/右手叉子，让其他哲学家有吃饭的机会，代码均已贴出如下：



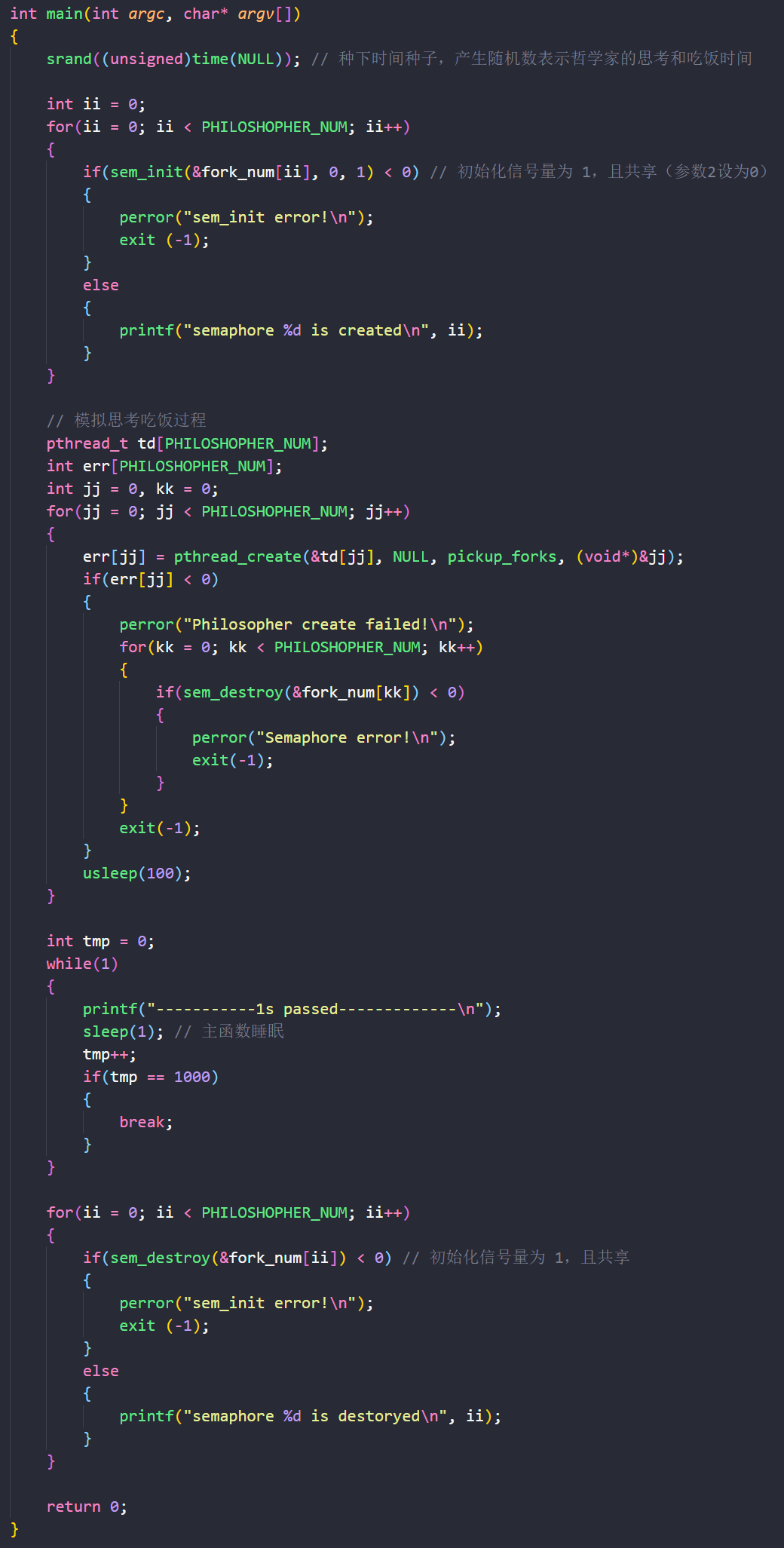


接下来，我们对main函数进行分析，main函数先调用函数srand，以当前时间为参数设置随机数种子，以确保函数time\_of\_think\_or\_eat中的函数rand可以正确地产生随机数。随后，main函数的第一个for循环完成了对信号量数组fork\_num的初始化。

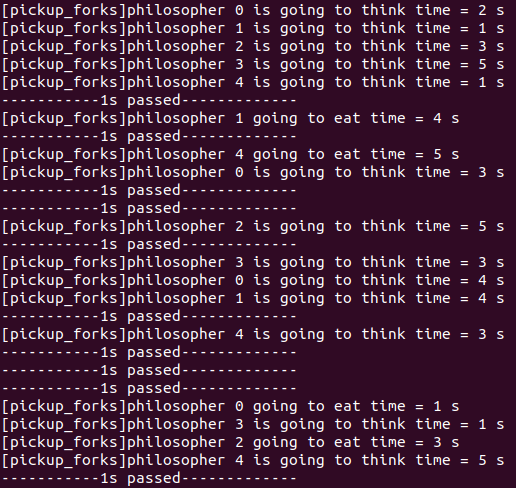
随后，进入第二个for循环，这一循环的主要工作即为调用POSIX函数pthread\_create创建多个执行函数pickup\_fork的进程，并且创建一个进程后就检查是否创建成功，如果创建失败，就要打印错误信息，并进入内层for循环调用POSIX函数sem\_destory释放所有已创建的进程，而在释放已创建进程的过程中，如果函数sem\_destory出现异常，则打印错误信息并直接调用函数exit(-1)终止程序。要注意的是，在内层循环结束后要调用函数usleep(100)，这主要是为了防止传入的jj值都一样。



main函数最后的部分很简单，即让主函数在条件为1的while循环中不断调用函数sleep休眠，让5个哲学家交替思考与吃饭，当计数变量tmp达到预设值后跳出while循环。最后的for循环调用POSIX函数sem\_destory释放已创建的进程，防止资源泄漏。



dph.c编译运行的结果截图如下：



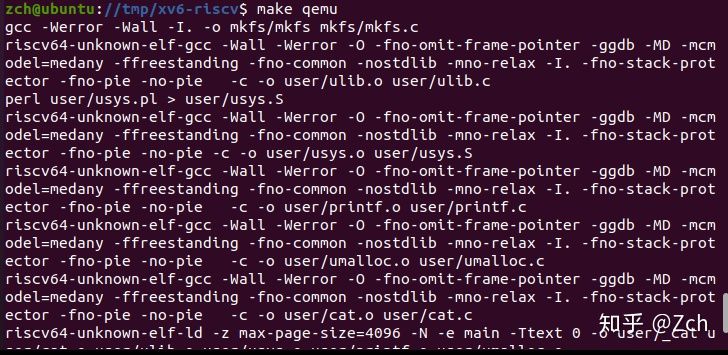
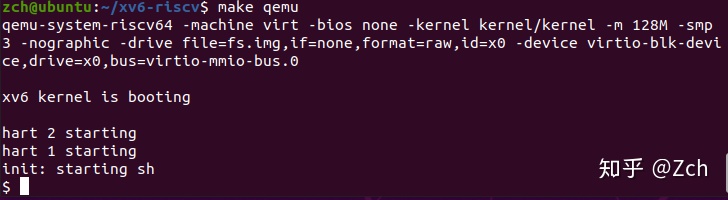
可以发现，哲学家都能够轮替吃饭与思考，说明我们的程序编写成功了，使哲学家用餐问题得到了解决。

# MIT 6.S081课程实验

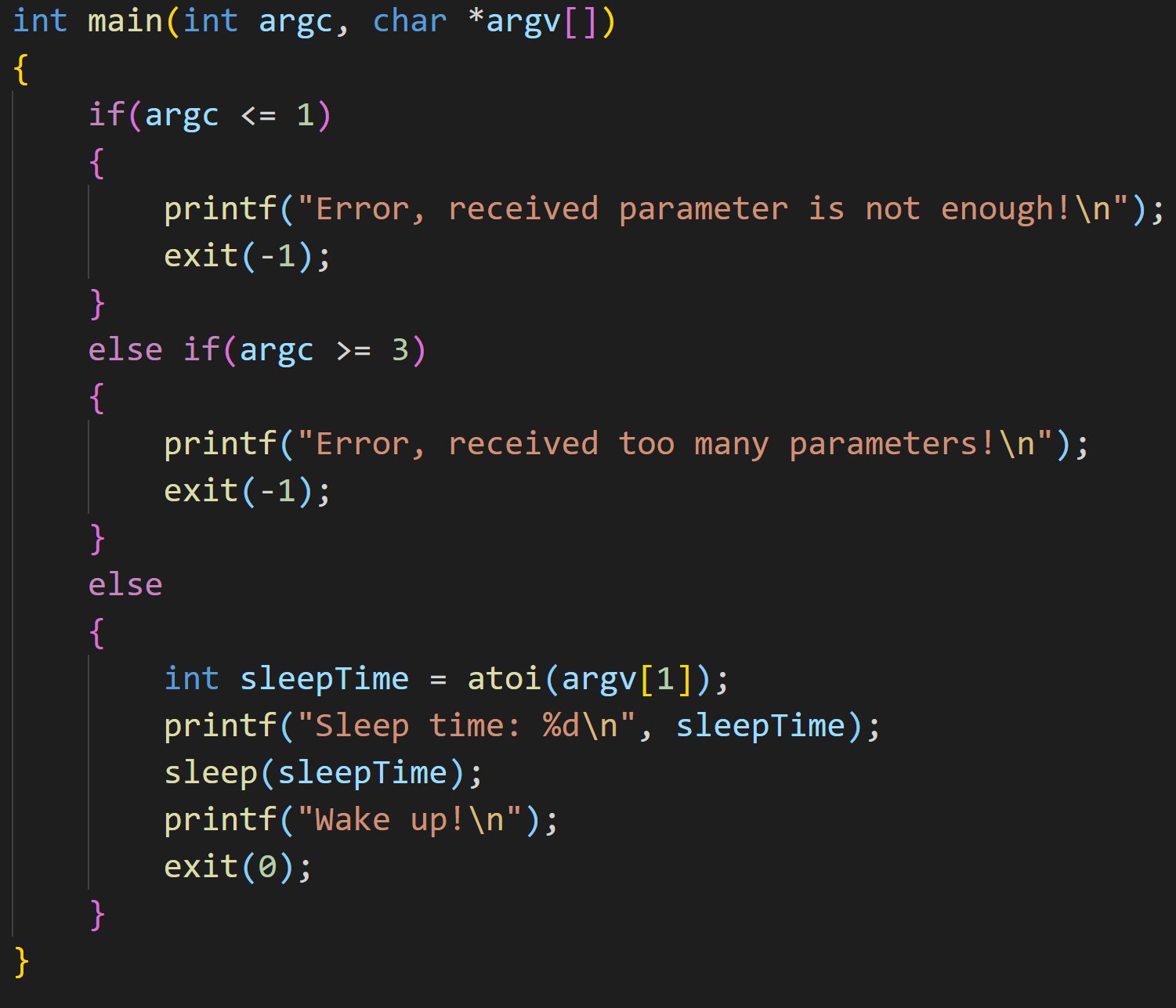
**1.xv6安装**

xv6操作系统详细的安装流程可以点击链接：

<https://zhuanlan.zhihu.com/p/267159664>

在我创作的知乎文章中查看细节，其中包含了从虚拟机安装ubuntu到从ubuntu安装xv6和qemu的全过程，这里仅展示安装完成并运行的截图如下：

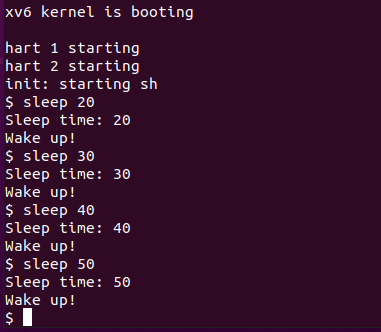
**2.sleep.c文件**

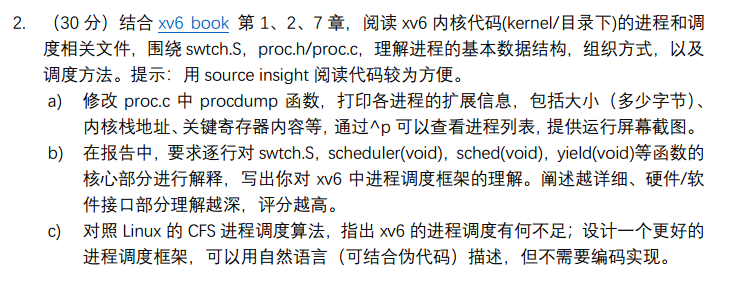
如下图，代码的核心部分如下：

其中，main函数接收命令行参数以实现用户输入控制效果。main函数接收的第一个整形参数argc为用户输入的命令字符串的数量，第二个字符串指针数组参数argv储存了用户依次输入的各个命令字符串。在这里，用户输入的正常格式应该为“sleep 数字”，如sleep 10，即为让系统休眠10毫秒。

接下来，main函数对接收的参数进行处理，接受到的参数数量应该正好为2，超过规定数量和少于规定数量都是不符合规定的，因此前两个判定条件对参数的数量进行判断，确保参数数量刚好为2，否则打印错误信息并调用函数exit(-1)退出。

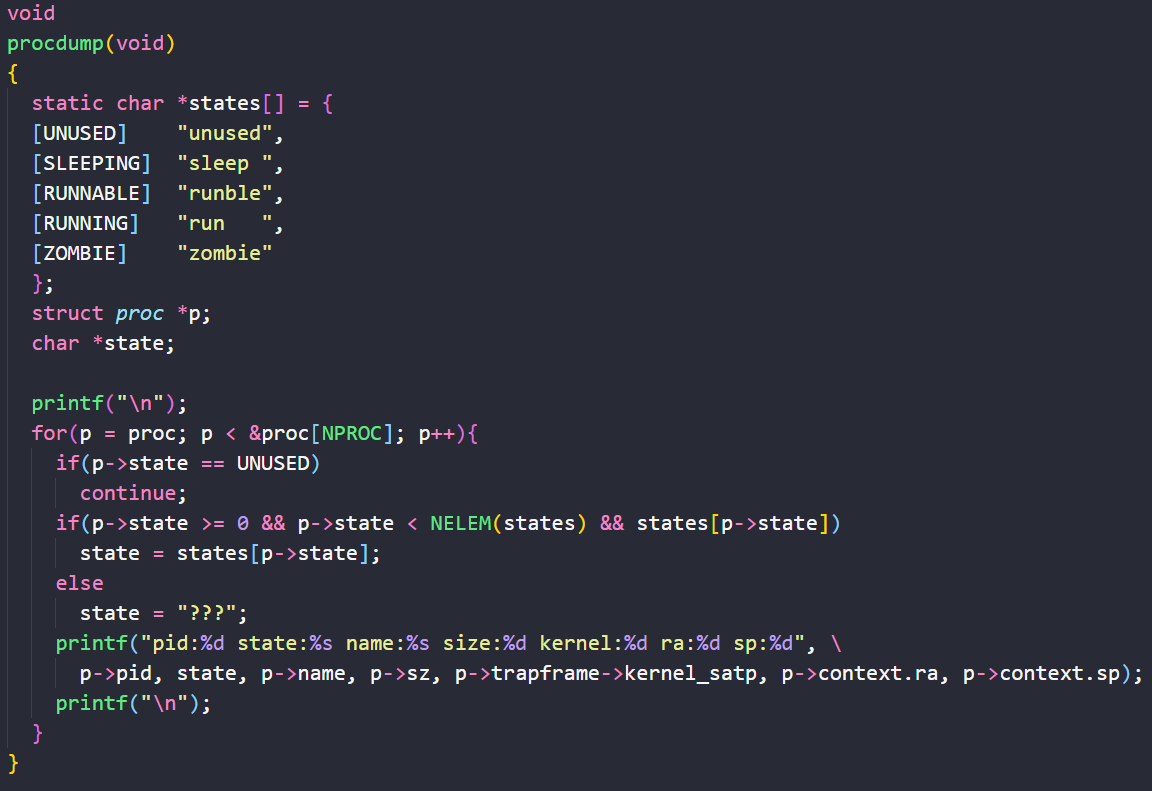
最后，若传入的参数数量无误，则将接收的第二个字符串argv[1]，用函数atoi转换为整形数，并存储在变量sleepTime中，并打印休眠时间信息。接下来，调用标准函数sleep并传入休眠时间这一整形数作为参数，让系统“真正”休眠，在休眠结束后返回并打印“Wake up!\n”，最后调用函数exit(0)退出，运行结果截图如下：



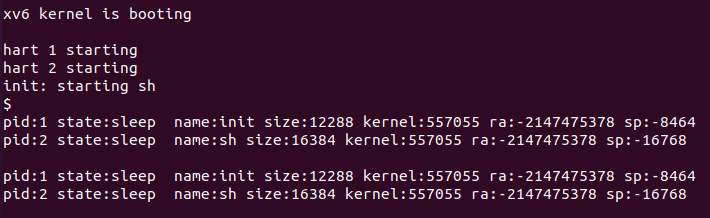


**a）：**

修改后的函数procdump代码如下：



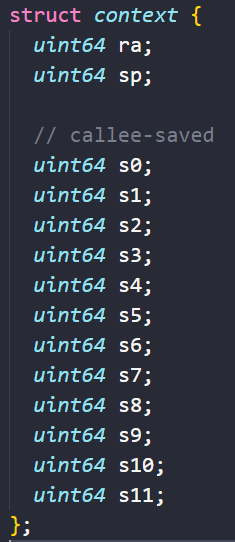
本题目中要求对函数procdump执行的修改主要集中在其printf语句中，只要将进程结构体的各个成员弄懂，打印出来这些成员的值其实很简单，直接将输出附加到printf语句即可。在这里我们附加打印出进程的大小p->sz、内核堆栈地址p->trapframe->kernel\_satp以及返回地址寄存器的地址p->context.ra和栈顶指针寄存器的地址p->context.sp，运行结果如下：



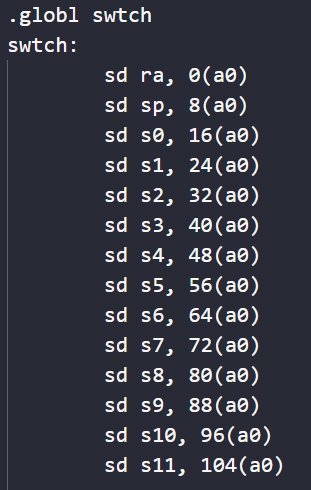
**b）：**

**1.swtch.S**

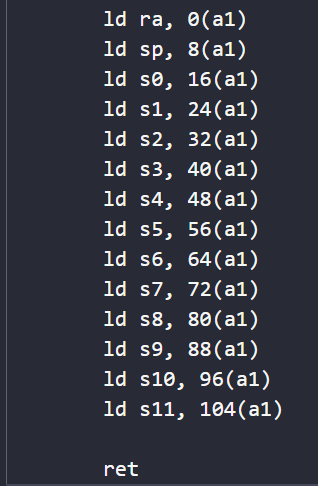
该函数的C接口为void swtch(struct context\* old, struct context\* new)。其中，根据结构体context的定义，我们知道context保存着被调用函数必须手动保存的寄存器的值，即这一结构体的实质即为返回地址寄存器ra、栈顶指针寄存器sp和s0-s11这12个寄存器的状态。知道结构体context的细节对我们理解函数swtch是很重要的，因为xv6使用context切换来完成进程切换操作。



接下来，我们回到文件swtch.S，可以看到，语句.golbl swtch定义了函数swtch为外部函数，使得外部文件可以找到该函数并通过接口对它进行调用。在下面的第一张图，即swtch函数主体的上半部分中，包含了14条sd语句，由于sd语句的含义是将数据从寄存器存储到存储器中，例如语句sd ra, 0(a0)，其意义是将参数a0的寄存器ra的地址加上偏移量0，再将此地址的8位数据存入存储器，对照结构体context的成员来看，不难发现，这14条sd语句将传入的第一个参数old的所有成员存入了存储器。

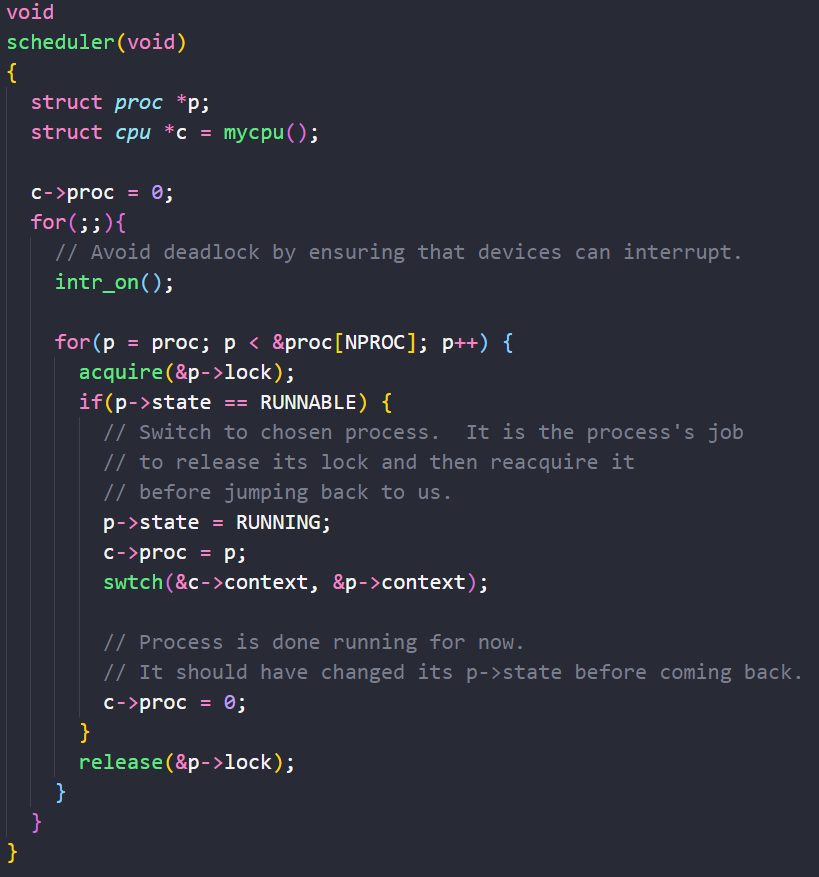


再来看swtch函数主体的下半部分，包含了14条ld语句，与上半部分的结构是对称的，再结合指令ld的意义，我们可以知道这14条ld语句的作用为将参数a1的所有成员的数据载入14个寄存器ra、sp、s1-s11中。



最后，调用指令ret，返回调用子程序swtch的位置继续执行。

**2.scheduler(void)**

该函数是单CPU处理器调度函数，每个CPU都在设置好自身以后调用该函数，函数scheduler无返回值，进入函数后循环进行下列操作：首先选取一个进程，随后将CPU切换到该进程运行，最后通过切回scheduler来让该进程传导控制。本质上，函数scheduler运行了一个普通的循环：找到一个进程来运行，运行到它停止为止，然后继续循环。

进入函数后，先建立一个进程结构体p，再通过函数mycpu建立一个cpu结构体c，将c的proc属性置空。接下来，进入for循环，先调用函数intr\_on，这是由于可能进程都在等待 I/O，从而找不到一个RUNNABLE的进程，如果调度器一直不允许中断，I/O 就永远无法到达了，这样就会产生死锁的问题。

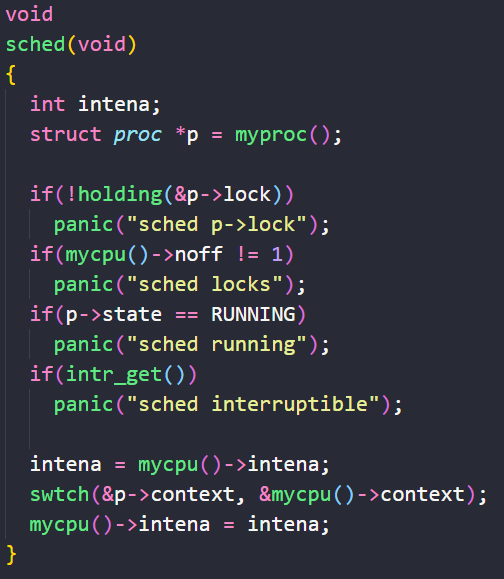
接下来，进入内层循环for(p = proc; p < &proc[NPROC]; p++){…}，令先前建立的进程结构体p为进程数组proc的首元素地址，令其与进程数组proc的最末尾元素的后一个单元的地址&proc[NPROC]比较，倘若相等则说明数组proc已遍历完成。

在该内层循环for(p = proc; p < &proc[NPROC]; p++){…}中，先调用函数acquire获取当前进程p的锁，确保该进程执行的安全性。接下来，if语句检查当前进程p的状态是否为可运行RUNNABLE，如果该条件为真，则将当前进程p的状态改为运行中RUNNING，将CPU的proc域赋值为当前进程p，即让CPU执行当前进程p，再调用函数swtch切换到上下文执行当前进程p的任务，而当函数swtch切回时，说明任务已执行完毕，重新令cpu结构体c的proc域置空。需要注意的是，在函数swtch切回时，当前进程p的状态p->state应该已在执行结束前更新。

最后，离开if语句，调用函数release释放当前进程p的锁p->lock，该语句的作用是要释放进程表锁，这是为了防止CPU闲置时，由于当前CPU一直占有锁，其他CPU无法调度运行而导致的死锁，如果一个闲置的调度器一直持有锁，那么其他 CPU 就不可能执行上下文切换或任何和进程相关的系统调用了，也就更不可能将某个进程标记为RUNNABLE，然后让闲置的调度器能够跳出循环了。进入下一循环。

**3.sched(void)**

该函数的作用是切换到调度器，其执行的必要条件是获取执行进程p的锁p->lock，同时进程的状态proc->state已被改变。此外，该函数还具备保存中断的作用，中断是内核线程的属性，而不是CPU的属性，中断的检查应查看两个变量proc->intena和proc->noff，变量proc->intena表示在中断发生前中断是否被允许，变量proc->noff表示中断的深度，即中断每多一层，变量proc->noff就会加一。需要指出的是，中断也可能发生于一些有锁但无进程之处。

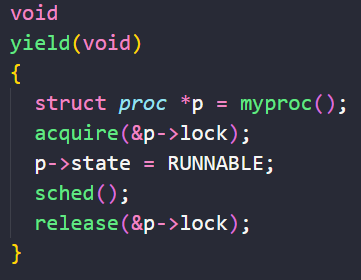


进入函数体，函数先声明了整形变量intena和进程结构体指针p，p用函数myproc的返回值初始化，即表示当前正在执行的进程。接下来，函数进入一系列if判定语句，第一句为if(!holding(&p->lock))，即判定CPU是否正持有进程p的锁p->lock，如果CPU已持有锁，则if语句通过，如果CPU尚未持有锁，则打印关于p->lock的信息；第二个if语句为if(mycpu()->noff != 1)，即判定当前CPU的中断的层级是否为多层嵌套中断，如果是则打印信息，否则if语句通过；第三个if语句为if(p->state == RUNNING)，即判定当前进程的状态是否为正在运行，如果是则打印信息，否则if语句通过；第四个if语句为if(intr\_get())，即调用函数intr\_get判定设备中断是否已经可以进行，如果返回结果为否则打印信息，否则if语句通过。

最后，在函数体开头声明的intena变量先临时保存由函数mycpu返回的当前CPU结构体的intena域，即mycpu()->intena的值，再调用函数swtch切换上下文执行进程，执行完毕后再将原先存储的intena值赋回给mycpu()->intena。

**4.yield(void)**

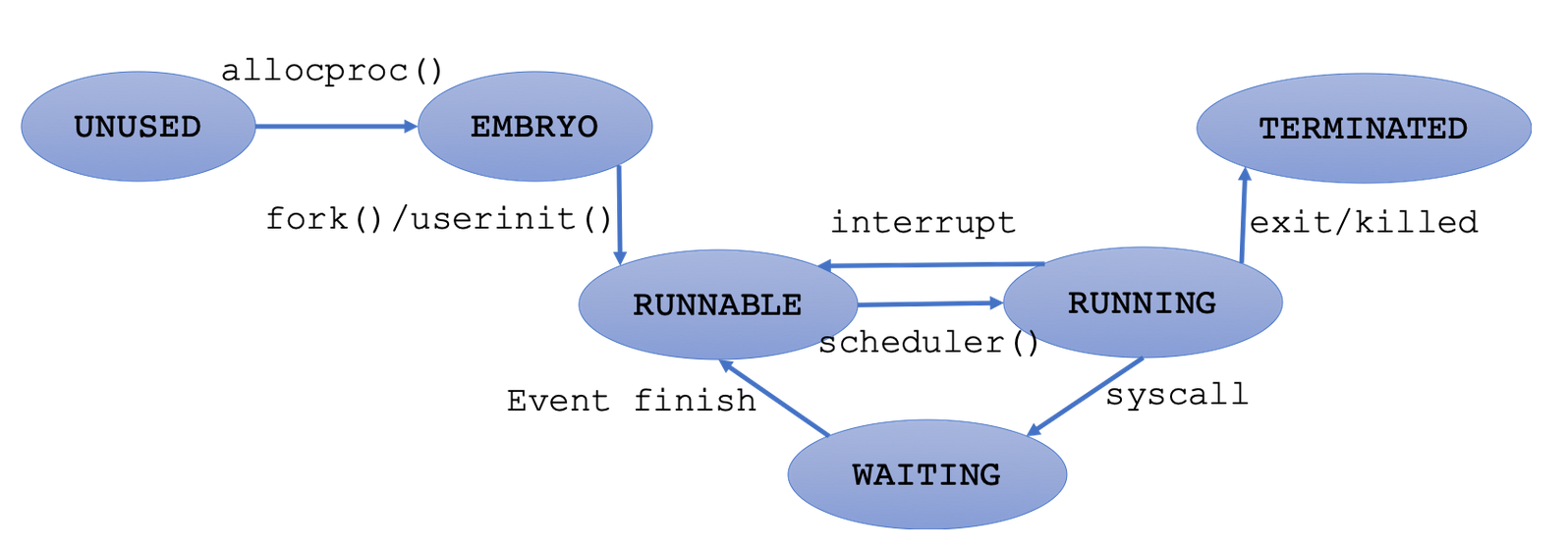
该函数使得某一个调度周期内，进程持有的CPU被放弃，当前进程会让出CPU资源来给其他线程执行。



其中，先定义进程结构体指针p，调用函数myproc为它初始化，获取当前进程。接下来调用函数acquire获取进程p的锁p->lock，再将进程p的状态p->state改为可执行，即RUNNABLE，接下来调用上文提到的函数sched使进程p让出CPU，切换到CPU原有的上下文mycpu()->context执行，最后释放进程p的锁p->lock。

**5.总结**

总体来说，xv6是支持多处理器多进程的操作系统，每个CPU都能够并行地运行不同的进程，同一个CPU也能通过不断地切换进程达到并发的效果，xv6在时钟中断机制下，在很短的时间内完成进程切换操作使得宏观上感觉在同一个CPU下能出现“并行”的效果。

xv6进程包括UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE这6种状态，即未使用态、初始态、等待态、就绪态、运行态、僵尸态。状态转换图如下：

xv6永远不会直接从一个进程的上下文切换到另一个进程的上下文，这些都是通过一个中间的内核线程实现的：内核调度器线程。

当进程用完它的CPU时间片时，时钟中断会调用yield函数来让出CPU给新的进程，yield调用sched函数，sched调用swtch来切换到调度器线程。调度器线程从进程表中找到一个就绪进程，并初始化进程运行环境，然后调用swtch切换到新的进程。而调度器线程仅仅是简单地进行轮转调度，一旦找到就绪线程便切换到新的线程。

**c):**

**1.Linux调度算法：**