**os-assignment1实验报告**

18364067 罗淦元

1. **实验要求**

在github上创建os-assignment1项目，项目包括三个题目的结果:

1. 生产者消费者(producer-consumer)问题：prod.c和cons.c
2. 哲学家就餐（dinning philosophers）问题：dph.c
3. MIT S.081实验：xv6阅读和实验过程

该项目目录下，手工编写Makefile，通过命令行下的make all命令，可以同时产生上述文件对应的可执行文件：prod, cons, dph；通过单独的make 目标，可以产生单独的bin文件。比如make dph可以单独生成dph。提交之前，保证这三个程序均可正确执行，评分的时候将考察运行过程。

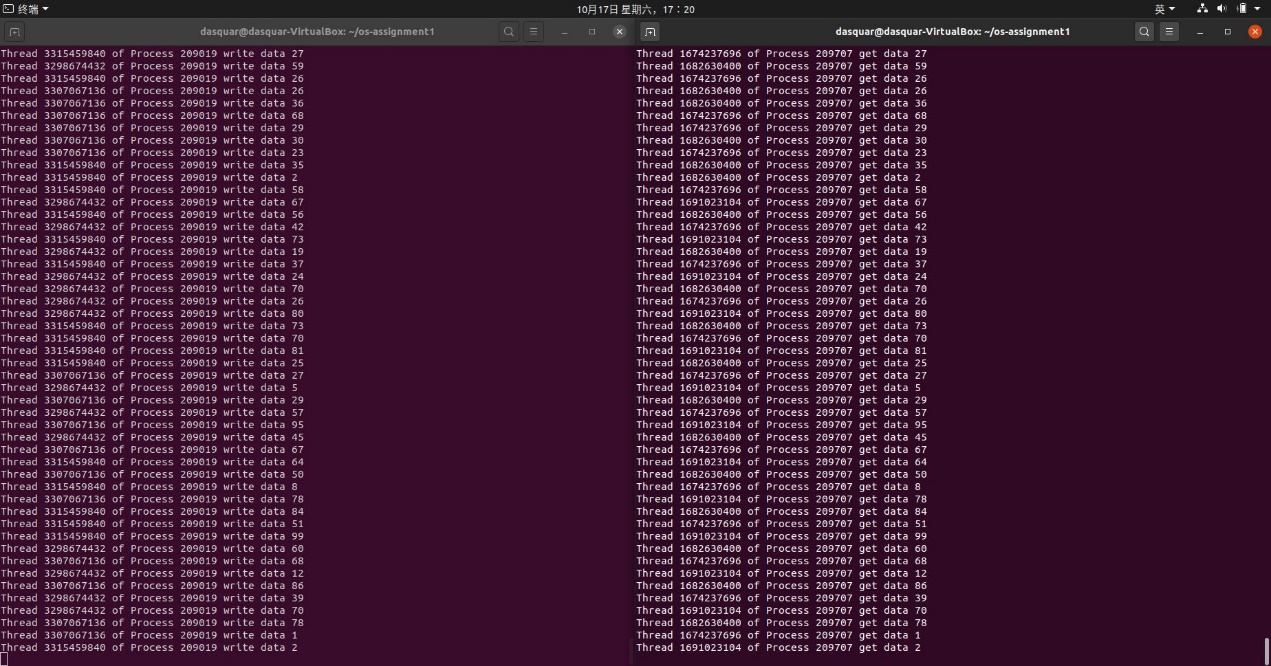
该目录下，同时存放1个word文件，按上述顺序包含实验内容报告，每个题目安排一大节。

1. **（30分）生产者消费者问题**
2. 需要创建生产者和消费者两个进程（注意：不是线程），一个prod，一个cons，每个进程有3个线程。两个进程之间的缓冲最多容纳20个数据。
3. 每个生产者线程随机产生一个数据，打印出来自己的id（进程、线程）以及该数据；每个消费者线程取出一个数据，然后打印自己的id和数据。
4. 生产者和消费者这两个进程之间通过共享内存来通信，通过信号量来同步。
5. 生产者生成数据的间隔和消费者消费数据的间隔，按照负指数分布来控制，各有一个控制参数,
6. 运行的时候，开两个窗口，一个./prod ，另一个./cons ，要求测试不同的参数组合，打印结果，截屏放到作业报告里。

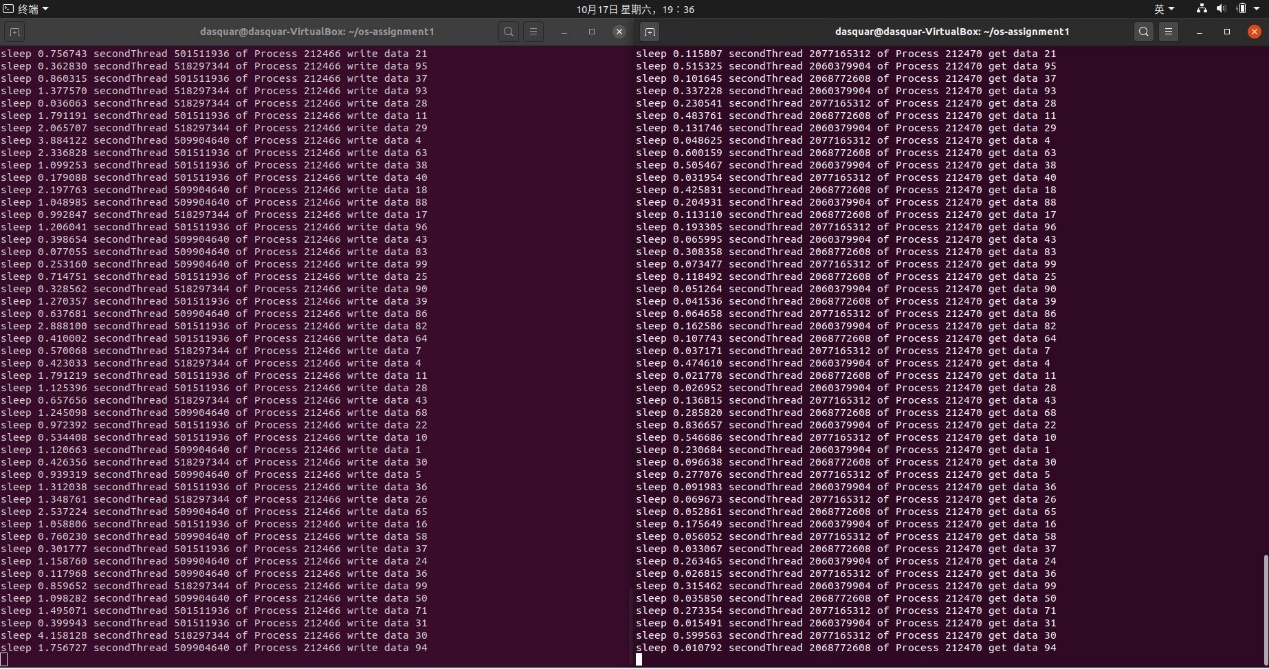
**1.1实验结果**

以下展示不同参数组合下，生产者进程（左）和消费者进程（右）的输出结果

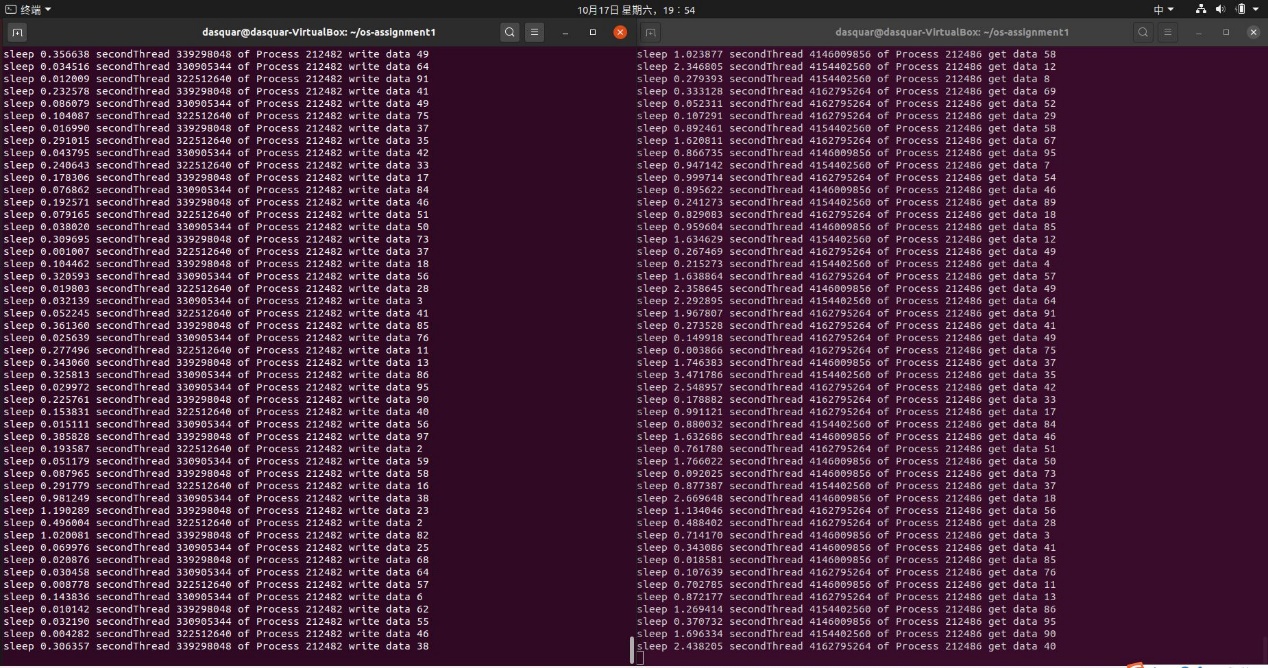
1.1.1 （）



* + 1. （）



* + 1. （）



**1.2实验结果分析**

当λp<=λc时，cons几乎同步输出prod产生的数据。说明两进程间通讯的进度主要受限于prod进程产生数据的速度，这和实际代码相符。

而当λp>λc时，cons进程消费的数据为prod进程在BUFFER\_SIZE个数据中产生最早的那个。说明两者通讯速度受限于cons消费速度，prod进程在等待cons消耗后空出缓存区。

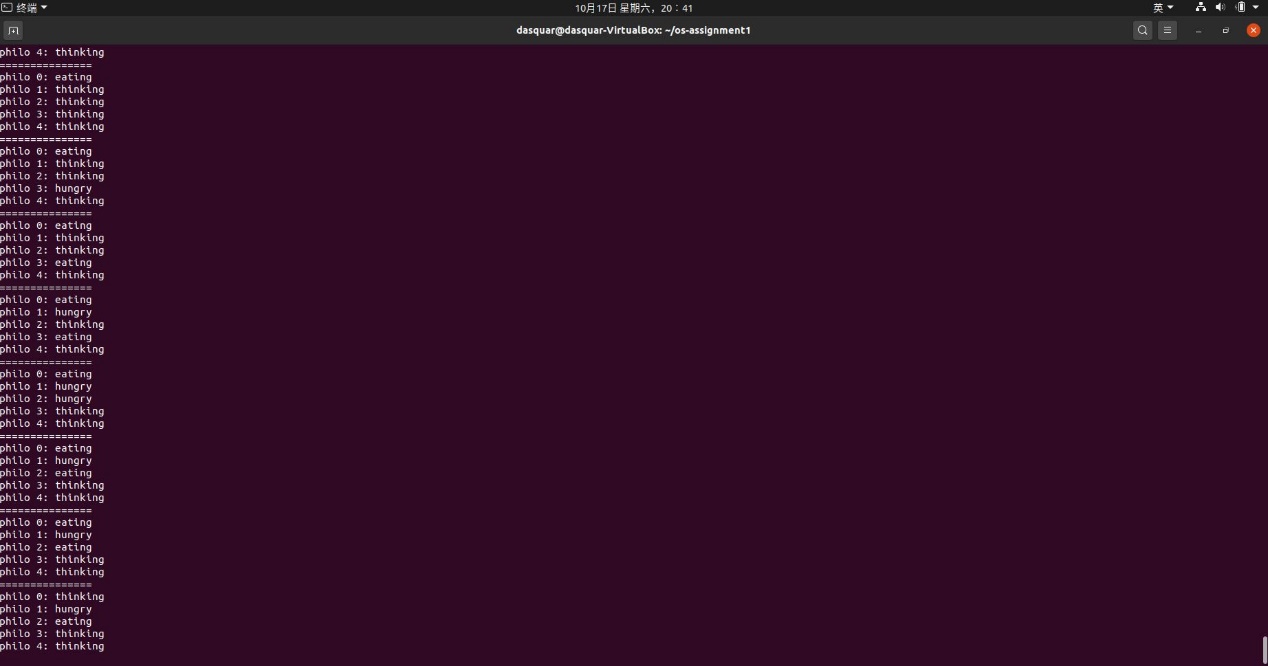
1. **（20分）哲学家就餐问题**

**参考课本（第十版）第7章project 3的要求和提示**

**1. 使用POSIX实现**

**5. 要求通过make，能输出dph文件，输出哲学家们的状态。打印结果，截屏放到作业报告中。**

2.1实验结果



2.2 实验分析

本实验中解决哲学家就餐问题的思路是，每当一名哲学家进入饥饿状态，就会“尝试就餐”。“尝试就餐”行为将检查其相邻两位哲学家是否在就餐。

如果两者都在非就餐状态，就马上将自身状态转为“就餐”。否则，就会将自身挂起等待信号self[i]并释放mutex[i]锁。

1. **void** pickup\_forks(**int** i){
2. state[i] = HUNGRY;
3. tryeat(i);
4. pthread\_mutex\_lock(&mutex[i]);
5. **while**(state[i] != EATING){
6. pthread\_cond\_wait(&self[i], &mutex[i]);
7. }
8. pthread\_mutex\_unlock(&mutex[i]);
9. }

而当一名哲学家完成“就餐”，他有义务通知相邻的两名哲学家他已完成就餐，并使得他们再次“尝试就餐”，同时自身回到思考状态。

1. **void** return\_forks(**int** i){
2. state[i] = THINKING;
3. printf("philosopher %d says he has finished dinner\n", i);
4. tryeat((i+1)%5);
5. tryeat((i+4)%5);
6. }

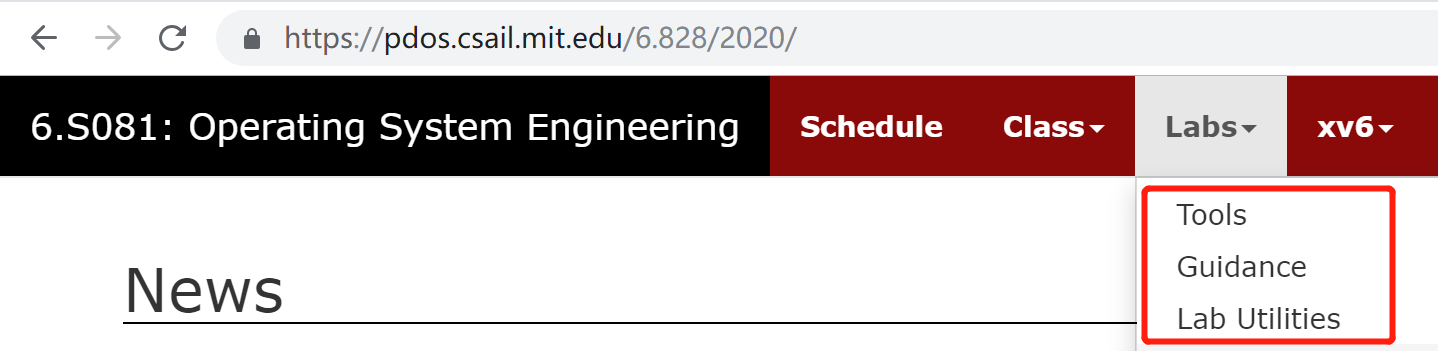
“尝试就餐”行为的源代码如下。

1. **void** tryeat(**int** i){
2. printf("philosopher %d try to eat\n", i);
3. **if**(state[i] == HUNGRY && (state[(i+4)%5] != EATING) && state[(i+1)%5] != EATING) {
4. pthread\_mutex\_lock(&mutex[i]);
5. state[i] = EATING;
6. pthread\_cond\_signal(&self[i]);
7. pthread\_mutex\_unlock(&mutex[i]);
8. }
9. **else** printf("philosopher %d fail to eat\n", i);
10. }

显然，当哲学家i尝试调用相邻哲学家的tryeat函数时，如果满足条件，将会夺取其下标对应的互斥锁mutex[i2](i2为相邻哲学家下标)避免在此过程中其他哲学家更改该哲学家状态state[i2]，同时使i2哲学家进入进餐状态并通知其解除挂起等待，释放互斥锁。这样就使得哲学家们能够有序就餐。

如实验结果所示，这种处理方式能够有效避免两名哲学家争夺筷子导致的“死锁”问题。同时具有一个缺陷，就是如果一名哲学家左右的两名哲学家恰好交替进行就餐行为，将会导致当前哲学家被“饿死”。这是本算法无法避免地问题。

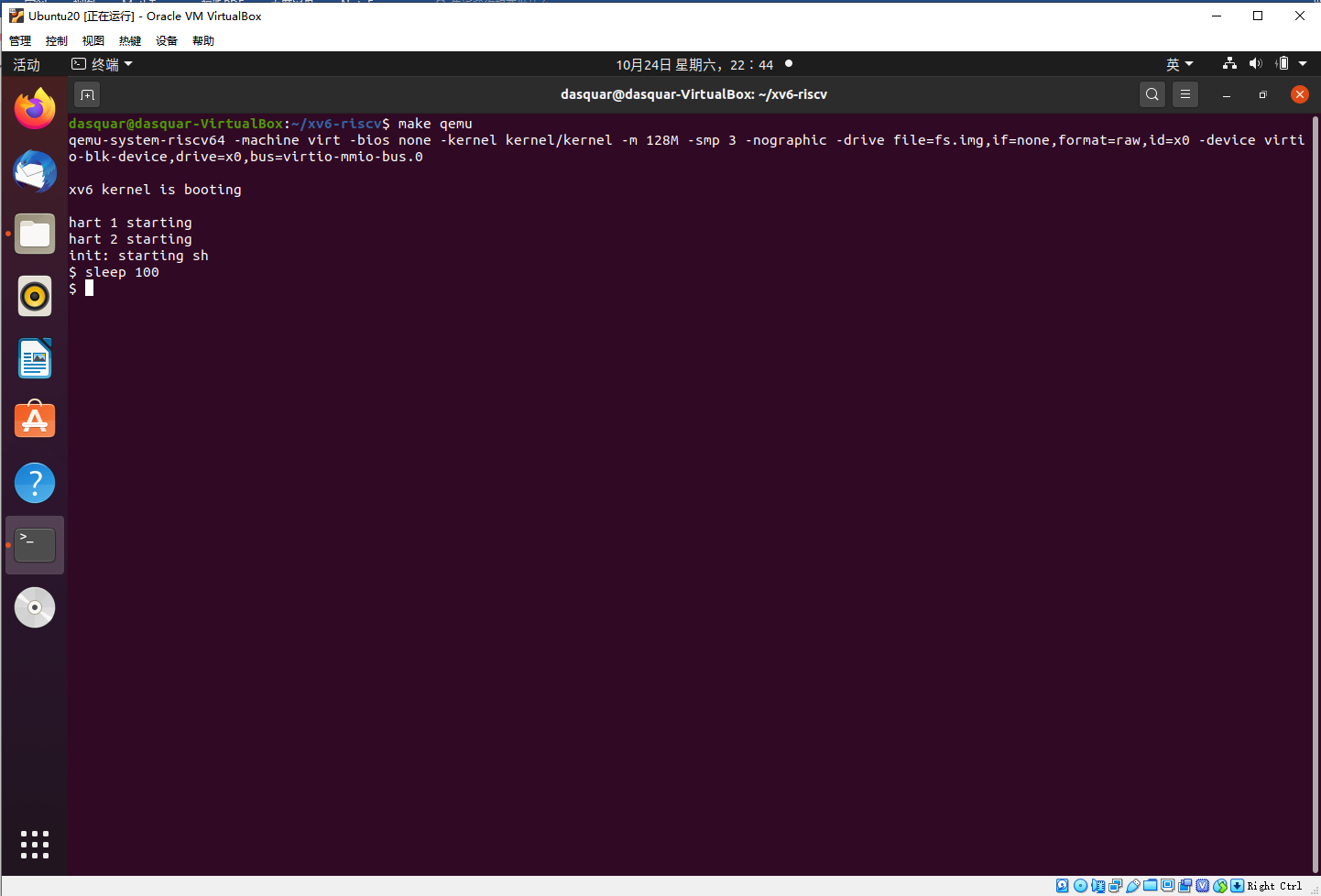
1. **（50分）MIT 6.S081课程实验**
2. （20分）阅读MIT 6.S081[项目介绍](https://pdos.csail.mit.edu/6.828/2020/),如下图；完成xv6的安装和启动（Ctrl-a x可退出）；完成Lab: Xv6 and Unix utilities中的sleep（easy）任务，即在user/下添加sleep.c文件。在报告中提供sleep.c的代码，并提供sleep运行的屏幕截图。提示：在vmware下安装ubuntu20，可以较为顺利完成xv6安装和编译。



* 1. 实验结果

源代码详见附件。

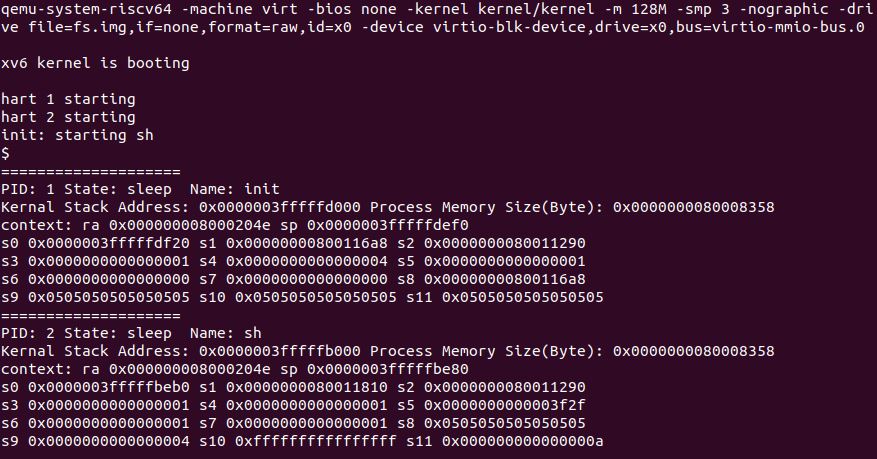
运行结果如图所示。



可以看到，当sleep指令没有参数时，将会向用户抛出异常。当收到参数时，首先将其转为数字，然后调用根据user/user.h内给出的用户程序可用的sleep函数，进一步调用system call并最后使用汇编完成sleep操作。需要注意在代码最后要使用exit()退出程序。通过编辑Makefile中的UPROGS，使得程序成为xv6 sleep指令指向的代码。

**3.2（30分）结合[xv6 book](https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf) 第1、2、7章，阅读xv6内核代码(kernel/目录下)的进程和调度相关文件，围绕swtch.S，proc.h/proc.c，理解进程的基本数据结构，组织方式，以及调度方法。提示：用source insight阅读代码较为方便。**

* 1. **修改proc.c中procdump函数，打印各进程的扩展信息，包括大小（多少字节）、内核栈地址、关键寄存器内容等，通过^p可以查看进程列表，提供运行屏幕截图。**



* 1. **在报告中，要求逐行对swtch.S，scheduler(void)，sched(void)，yield(void)等函数的核心部分进行解释，写出你对xv6中进程调度框架的理解。阐述越详细、硬件/软件接口部分理解越深，评分越高。**

3.2.1 proc.h

作为开始，首先要在proc.h中了解一些要用到的数据结构的定义。

1. context上下文结构

1. //整个context结构为当前进程的上下文，用于进程切换时保存状态和恢复状态
2. **struct** context {
3. uint64 ra;    //ra寄存器，存储了当前进程的pc值
4. uint64 sp;    //堆栈寄存器值
6. // callee-saved
7. uint64 s0;    //一些易失性寄存器的值
8. uint64 s1;
9. uint64 s2;
10. uint64 s3;
11. uint64 s4;
12. uint64 s5;
13. uint64 s6;
14. uint64 s7;
15. uint64 s8;
16. uint64 s9;
17. uint64 s10;
18. uint64 s11;
19. };

2.CPU结构

1. // 每个cpu的状态
2. **struct** cpu {
3. **struct** proc \*proc;          // 在当前cpu上运行的进程的指针
4. **struct** context context;     // schedular的context，swtch函数通过进入schedular的上下文来转换进程
5. **int** noff;                   // push\_off的深度
6. **int** intena;                 // push\_off前中断的开启状态
7. };

3. Proc结构

1. // 每个进程的状态
2. **struct** proc {
3. **struct** spinlock lock; //自旋锁
5. // p->lock must be held when using these:
6. **enum** procstate state;        // 进程状态，枚举为UNUSED、SLEEPING、RUNNING等
7. **struct** proc \*parent;         //父进程指针
8. **void** \*chan;                  //非空则在chan指针上睡眠，相同chan的进程将在wake时一同唤醒
9. **int** killed;                  // 如果非零，进程已被杀死
10. **int** xstate;                  // 退出时返回父进程的返回值
11. **int** pid;                     // 进程id
13. // these are private to the process, so p->lock need not be held.
14. uint64 kstack;               //内核栈虚拟地址
15. uint64 sz;                   // 进程内存大小（byte为单位）
16. pagetable\_t pagetable;       // 用户页表
17. **struct** trapframe \*trapframe; //trampoline.S的数据页
18. **struct** context context;      // 进程上下文
19. **struct** file \*ofile[NOFILE];  // 打开的文件
20. **struct** inode \*cwd;           //最近目录
21. **char** name[16];               // 进程名，用于debug
22. };
    * 1. swtch.S

swtch的主要作用用于交换old和new进程的上下文。它将进程的上下上下文压入到内核栈中，并加载new进程先前压入的上下文。这里可以看出，当进程在执行时，其内核栈上下文总是空的，进程在使用用户栈

1. # Context switch
2. #
3. #   void swtch(struct context \*old, struct context \*new);
4. #
5. # Save current registers in old. Load from new.

8. .globl swtch                    #.global关键字用来让一个符号对链接器可见，可以供其他链接对象模块使用。
9. swtch:                          #第一步，保存old进程的上下文。a0传递了pld指针，a1传递了new指针。
10. sd ra, 0(a0)            #保存ra寄存器的值，即pc值，到a0指向位置的第0位起的8位，便于重新调度时返回到swtch前的状态
11. sd sp, 8(a0)            #保存堆栈寄存器的值，即存放栈的偏移地址，到a0指向地址的第8位起8位。后面同理
12. sd s0, 16(a0)           #保存12个易失性寄存器的值，如果这些值需要在调用返回时恢复的话。
13. sd s1, 24(a0)
14. sd s2, 32(a0)
15. sd s3, 40(a0)
16. sd s4, 48(a0)
17. sd s5, 56(a0)
18. sd s6, 64(a0)
19. sd s7, 72(a0)
20. sd s8, 80(a0)
21. sd s9, 88(a0)
22. sd s10, 96(a0)
23. sd s11, 104(a0)
24. #从a1，即new指针指向的地址上获取保存的上下文
25. ld ra, 0(a1)          #从a1第0位起读取8位保存到值ra寄存器，恢复新进程的pc值
26. ld sp, 8(a1)            #同理，恢复堆栈寄存器值
27. ld s0, 16(a1)           #同理，恢复12个易失性寄存器值
28. ld s1, 24(a1)
29. ld s2, 32(a1)
30. ld s3, 40(a1)
31. ld s4, 48(a1)
32. ld s5, 56(a1)
33. ld s6, 64(a1)
34. ld s7, 72(a1)
35. ld s8, 80(a1)
36. ld s9, 88(a1)
37. ld s10, 96(a1)
38. ld s11, 104(a1)
40. ret
    * 1. yield()函数（由于定时器中断进入进程调度）
41. **void**
42. yield(**void**)
43. {
45. **struct** proc \*p = myproc();    //myproc函数获取到当前cpu上的进程
46. acquire(&p->lock);     //请求获得自旋锁
47. p->state = RUNNABLE;   //设置进程为可执行状态
48. sched();          //调用sched
49. release(&p->lock);     //释放自旋锁
50. }
    * 1. sched()函数
51. **void**
52. sched(**void**)
53. {
54. **int** intena;
55. **struct** proc \*p = myproc();    //获取当前cpu上的进程
57. /\*检查当前进程是否持有自旋锁、push\_off深度是否不为1、进程是否回到RUNNABLE状态以及是否关闭中断\*/
58. **if**(!holding(&p->lock))
59. panic("sched p->lock");
60. **if**(mycpu()->noff != 1)
61. panic("sched locks");
62. **if**(p->state == RUNNING)
63. panic("sched running");
64. **if**(intr\_get())
65. panic("sched interruptible");
67. intena = mycpu()->intena;  //由于中断是内核线程的权利而不是cpu，因此保存其intena
68. swtch(&p->context, &mycpu()->context);  //从当前进程的上下文转换到schedular的上下文
69. mycpu()->intena = intena;  //恢复intena
70. }
    * 1. schedular函数
71. **void**
72. scheduler(**void**)
73. {
74. **struct** proc \*p;
75. **struct** cpu \*c = mycpu();  //获取当前cpu结构，在start时
76. //已经通过对每个cpu地tp寄存器赋值
77. //来作为每个cpu的id，并映射到cpu结构数组
79. c->proc = 0;       //将当前cpu上的进程设为空
80. **for**(;;){          //显然是个死循环，将会不断遍历进程表来查找是否有RUNNABLE的进程，并切换到该进程
81. //确保设备中断打开避免死锁.
82. intr\_on();  //打开中断
84. **for**(p = proc; p < &proc[NPROC]; p++) {
85. acquire(&p->lock)              //进程p请求获得自旋锁
86. **if**(p->state == RUNNABLE) {         //如果p是可运行状态
87. //进入进程p。进程p需要释放自旋锁和重新请求自选锁来返回schedular
88. p->state = RUNNING;              //将p进程状态设为运行
89. c->proc = p;                 //并将cpu上进程设为p
90. swtch(&c->context, &p->context);          //交换p和schedular上下文。开始执行进程p
92. // Process is done running for now.
93. // It should have changed its p->state before coming back.
94. c->proc = 0;                 //清空当前cpu上的进程
95. }
96. release(&p->lock);             //进程p释放自旋锁
97. }
98. }
99. }

3.2.6 sleep和wakeup

1. // 在chan上sleep和wake的原子操作
2. // 被唤醒时从新索取锁
3. **void**
4. sleep(**void** \*chan, **struct** spinlock \*lk)
5. {
6. **struct** proc \*p = myproc();
8. // 必须获取p上的自旋锁
9. // 以更改p的state并调用sched
10. // 只要持有当前进程锁，就能保证不会错过wakeup
11. // 因为wakeup将索取进程锁，所以可以释放lk
12. **if**(lk != &p->lock){
13. acquire(&p->lock);
14. release(lk);
15. }
17. // 开始在chan上睡眠
18. p->chan = chan;
19. p->state = SLEEPING;
21. sched();  //调用sched
23. //清空chan
24. p->chan = 0;
26. // 重新获得lk锁
27. **if**(lk != &p->lock){
28. release(&p->lock);
29. acquire(lk);
30. }
31. }
33. // 唤醒在chan上睡眠的所有进程，也就是多个进程等待一个事件的发生
34. // 必须在不持有任何进程中自旋锁的前提下调用
35. **void**
36. wakeup(**void** \*chan)
37. {
38. **struct** proc \*p;
40. //查找睡眠中的进程，并将它设为RUNNABLE
41. //可以看到，进程在结束睡眠后并不是离开开始执行的
42. //而要进入到进程表中重新等待调度
43. **for**(p = proc; p < &proc[NPROC]; p++) {
44. acquire(&p->lock);
45. **if**(p->state == SLEEPING && p->chan == chan) {
46. p->state = RUNNABLE;
47. }
48. release(&p->lock);
49. }
50. }

3.2.7

基于上文对于swtch.S，scheduler(void)，sched(void)，yield(void)等函数的注释，我们开始分析整个xv6系统的进程调度过程。

1. 自旋锁

对于多处理器的系统来说，不能采用在标志寄存器中打开、关闭中断标志位的方式来防止中断处理中的并发，因此就需要因此spinlock自旋锁这个结构来解决。从保证临界区访问原子性目的来考量，自旋锁应该阻止包括中断、内核抢占和其他处理器对同一临界区的访问。我们查看spinlock结构可以看到，里面主要包括了locked属性，用于表示是否上锁；cpu属性，用于获得持有该锁的cpu。

1. **struct** spinlock {
2. uint locked;       // Is the lock held?
4. // For debugging:
5. **char** \*name;        // Name of lock.
6. **struct** cpu \*cpu;   // The cpu holding the lock.
7. };

2.请求和释放

相关的两个重要函数就是acquire()和release()。在此之前要先分析push\_off()和pop\_off()函数。

1. **void**
2. push\_off(**void**)
3. {
4. **int** old = intr\_get();     //获取当前中断使能状态
6. intr\_off();           //关闭中断
7. **if**(mycpu()->noff == 0) //如果这是第一层push\_off
8. mycpu()->intena = old;   //在当前cpu的intena参数中存储push\_off前的中断使能状态
9. mycpu()->noff += 1;
10. }

其中，intr\_get()通过内嵌汇编获取SIE寄存器的值，判断中断使能状态。而intr\_off()同样通过内嵌汇编向sstatus寄存器写入来关闭中断使能。

至于pop\_off()函数则于push\_off()函数相反。

1. **void**
2. pop\_off(**void**)
3. {
4. **struct** cpu \*c = mycpu();      //获取当前cpu
5. **if**(intr\_get())            //如果当前中断使能打开，说明出现异常
6. panic("pop\_off - interruptible");
7. **if**(c->noff < 1)         //如果当前push\_off()深度小于1，说明异常
8. panic("pop\_off");
9. c->noff -= 1;          //cpu的push\_off()深度-1
10. **if**(c->noff == 0 && c->intena)   //如果push\_off()深度为0且push\_off()前中断打开，则打开中断
11. intr\_on();
12. }

每执行一层push\_off，就对应于执行一层pop\_off。只有当所有pop\_off()执行完成，即noff == 0，cpu才能够恢复到push\_off前的中断使能状态。

我们回到acquire函数。其中holding函数返回当前cpu是否已经持有当前锁。

1. // 请求锁并在获得前自旋
2. **void**
3. acquire(**struct** spinlock \*lk)
4. {
5. push\_off();   // push\_off一次，关闭中断
6. **if**(holding(lk))   //如果已经持有该锁，再次请求将导致死锁，抛出异常
7. panic("acquire");
9. // 在 RISC-V, sync\_lock\_test\_and\_set用于进行原子交换操作
10. **while**(\_\_sync\_lock\_test\_and\_set(&lk->locked, 1) != 0)   //成功获取到锁前保持自旋
11. ;
12. //告诉C编译器和进程不要把加载和存储操作移到本点后
13. //保证临界区内存的操作严格在锁被请求后执行
14. \_\_sync\_synchronize();
16. // 记录持有锁的cpu方便debug
17. lk->cpu = mycpu();
18. }

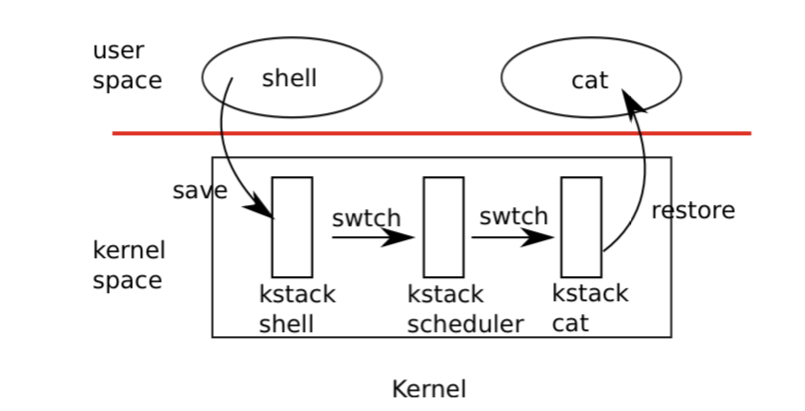
Realse函数也很简单。

1. // 释放锁
2. **void**
3. release(**struct** spinlock \*lk)
4. {
5. **if**(!holding(lk))  //没有获取就释放，抛出异常
6. panic("release");
8. lk->cpu = 0;   //清空锁上的cpu
10. \_\_sync\_synchronize();
12. //释放锁，等效于lk->locked=0。
13. //不使用C因为C标准可能用多个存储指令实现。
14. \_\_sync\_lock\_release(&lk->locked);
16. pop\_off();    //进行pop\_off
17. }

进行了一些基本操作的解释后，我们可以开始分析xv6的进程调度过程了。对于单个的CPU，schedular是主要的函数。CPU初始化后将会不断循环从进程队列中选择进程并执行。这一点在前文的注释中已经详细分析了schedular的运行方式。

进程回到sched()的方式有多种。进程结束时，将执行exit()函数，exit()函数的结尾调用了sched()函数，并由sched()函数将控制权移交给schedular。而sleep函数同样具有主动切换到调度器的功能。除此之外，定时中断导致的yield函数，也同样会调用sched函数来返回到schedular。

总的来说，一个进程在执行过程中，通过wait、sleep、exit和定时中断等系统调用和中断，主动或被动地进入到sched函数中，sched函数负责调用swtch函数将old进程地上下文保存到内核堆栈中，并将控制权交换给schedular调度器。



Schedular调度器对于每个CPU都是专有的，有着独立的寄存器和堆栈，以免被其他CPU访问到。Schedular调度器通过遍历进程表并获取第一个准备好运行的进程new，然后索取自旋锁避免多个cpu同时尝试切换同一进程new导致的竞争，接着调用swtch从内核栈中恢复new进程先保存的上下文，并从该上下文中ra指针指向的位置开始继续运行new进程（因为new进程在此之前可能执行过一段时间后被中断或处于等待而切换到其他进程）。进程如果要主动放弃CPU，则必须请求其进程的自旋锁，释放持有的其他锁，更新自己的状态并主动调用sched()（例如在程序结束时，调用exit()中的sched()）。而当一个程序长时间地执行时，将会由计时器中断调用yield()执行类似的过程。Sched中会检查每次程序进入是是否满足进程切换的条件，并且调用schedular继续上述循环。

* 1. **对照Linux的CFS进程调度算法，指出xv6的进程调度有何不足；设计一个更好的进程调度框架，可以用自然语言（可结合伪代码）描述，但不需要编码实现。**

我们可以看到，xv6进程调度主要是使用轮询的方法实现。调度器每次都从进程表中选取第一个状态为Runnable的进程执行。而Linux的CFS进程调度算法，其关键在于给每个进程的权重分配运行时间。

简化来说，CFS中进程在调度周期中占用的时间与进程权重与所有进程权重和有关。同时每个进程还有一个虚拟运行时间vruntime的量，其满足

Vruntime = 实际运行时间\*NI值为0的进程权重/进程权重 = （调度周期\*进程权重/总权重）\*NI值0的进程的权重/进程权重 = 调度周期\*NI值为0的进程权重/总权重。即是说，对于实际运行时间相同的进程，权重小的Vruntime增长更快。但总体上来说，每个进程的Vruntime又是同时推进的。根据这一点，选取vruntime小的进程来执行，一方面可以保证程序运行时间在虚拟时钟上是公平的，同时宏观上来说优先级高的程序总是有着更长的实际运行时间。

和CFS比起来，xv6的进程调度缺少对于每个进程的优先级别的判断，一方面导致了部分紧急的任务无法被优先执行。

CFS算法本身就是一个更好调度框架的例子。同时也是一种公平调度算法。整个调度器需要维护一个以上述vruntime组成的红黑树，其最左节点即vruntime最小的节点就是优先级最高的节点。每次进入调度器时，就可以加锁并将红黑树中最左节点设为运行状态再释放锁，同时向进程提供一个与A调度周期\*进程权重/总权重的时间片，并在进程执行完该时间片后中断并上下文切换回到调度器。而当进程调用sleep、exit函数时，则从红黑树中删除该节点，直到其重新进入就绪状态。通过这种方式，一方面能够保证每个进程在一个CPU的调度周期中都会被执行一段时间，使得短时间优先级低的进程也能被很快地完成。同时通过对nice值等优先级的设定，可以确保紧急进程更快地完成任务。