《操作系统》大作业 1

年冬冬

18364073

2020 年 10 月 25 日

1. （30 分）生产者消费者问题

1. 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。
2. 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。
3. 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。
4. 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
5. 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。

Prod.c 代码：

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <math.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t *empty;
sem_t *full;

int shm_fd;
const int BUFFER_SIZE = 20;
const char *name = "OS";
int *ptr = NULL;
int in = 0;


void *producer(void *param){

      int item;
      int count = 0;
      int t;
      while(1){
           t = exp(count);
           sleep(t/100);
           count++;
           item = rand()%20 + 1;
```

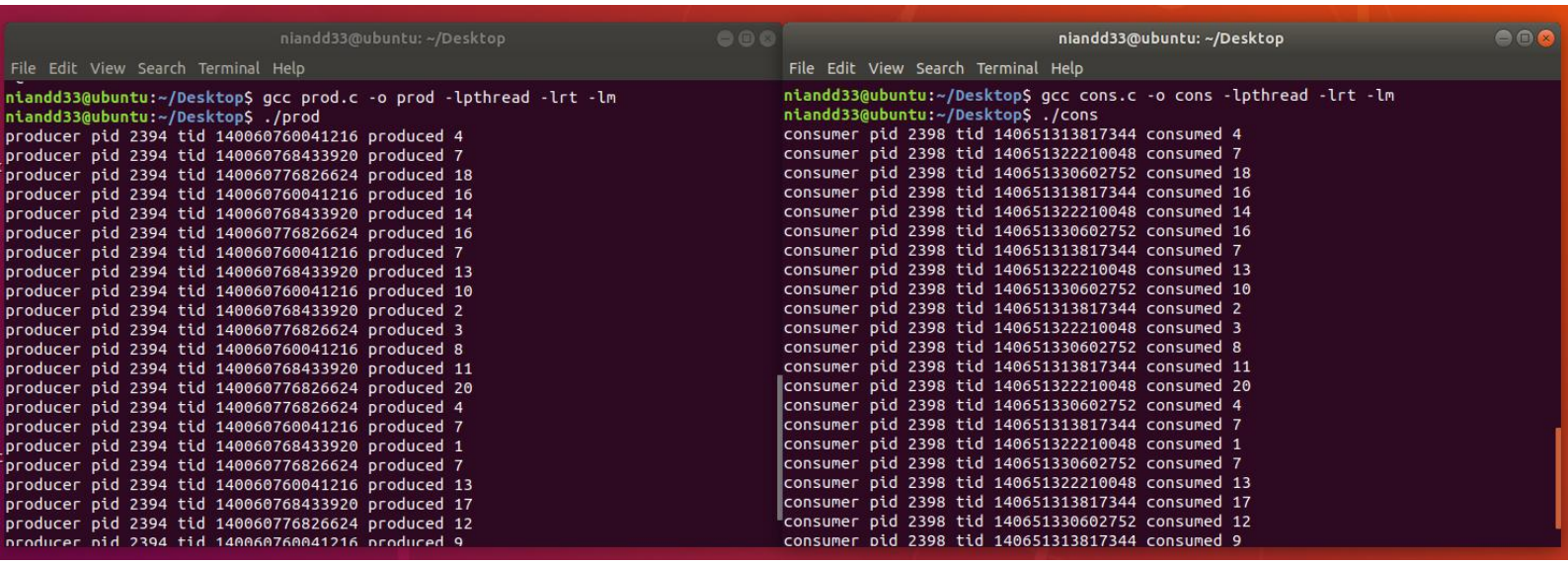
```
34.
35.             sem_wait(empty);
36.             pthread_mutex_lock(&mutex);
37.
38.             ptr[in] = item;
39.             in = (in + 1) % BUFFER_SIZE;
40.
41.              printf("producer pid %d tid %lu produced %d\n",
    getpid(), pthread_self(), item);
42.
43.             pthread_mutex_unlock(&mutex);
44.             sem_post(full);
45.
46.       }
47.
48. }
49.
50.
51. int main(int argc, char *argv[]){
52.
53.     shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
54.     ftruncate(shm_fd, BUFFER_SIZE);
55.      ptr = mmap(0, BUFFER_SIZE, PROT_WRITE, MAP_SHARED, sh
   m_fd, 0);
56.
57.     empty = sem_open("EMPTY", O_CREAT, 0666, 20);
58.     full = sem_open("FULL", O_CREAT, 0666, 20);
59.     pthread_mutex_init(&mutex, NULL);
60.
61.     pthread_t prod1,prod2,prod3;
62.     pthread_create(&prod1, NULL, producer, NULL);
63.     pthread_create(&prod2, NULL, producer, NULL);
64.     pthread_create(&prod3, NULL, producer, NULL);
65.
66.     pthread_join(prod1, NULL);
67.     pthread_join(prod2, NULL);
68.     pthread_join(prod3, NULL);
69.
70.     sem_destroy(empty);
71.     sem_destroy(full);
72.     pthread_mutex_destroy(&mutex);
73.     return 0;
74. }
```

Cons.c 代码：

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <math.h>

const int BUFFER_SIZE = 20;
const char *name = "OS";

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t *empty;
sem_t *full;

int shm_fd;
void *ptr = NULL;
int out = 0;

void *consumer(void *param){
     int count = 0;
     int t;
     while(1){
          t = exp(count);
          sleep(t/100);
          count++;

          sem_wait(full);
          pthread_mutex_lock(&mutex);

           printf("consumer pid %d tid %lu consumed %d\n", getpid(), pthread_self(), *((int*)ptr+out));
          out = (out + 1) % BUFFER_SIZE;

          pthread_mutex_unlock(&mutex);
          sem_post(empty);

     }

```

```
43. }
44.
45.
46. int main(int argc, char *argv[]){
47.
48.     empty = sem_open("EMPTY", O_CREAT, 0666, 20);
49.     full = sem_open("FULL", O_CREAT, 0666, 20);
50.     pthread_mutex_init(&mutex, NULL);
51.
52.     shm_fd = shm_open(name, O_RDONLY, 0666);
53.     ptr = mmap(0, BUFFER_SIZE, PROT_READ, MAP_SHARED, shm
    _fd, 0);
54.
55.     pthread_t cons1,cons2,cons3;
56.     pthread_create(&cons1, NULL, consumer, NULL);
57.     pthread_create(&cons2, NULL, consumer, NULL);
58.     pthread_create(&cons3, NULL, consumer, NULL);
59.
60.     pthread_join(cons1, NULL);
61.     pthread_join(cons2, NULL);
62.     pthread_join(cons3, NULL);
63.
64.     sem_destroy(empty);
65.     sem_destroy(full);
66.     pthread_mutex_destroy(&mutex);
67.
68.     return 0;
69. }
```

运行结果：



2. （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

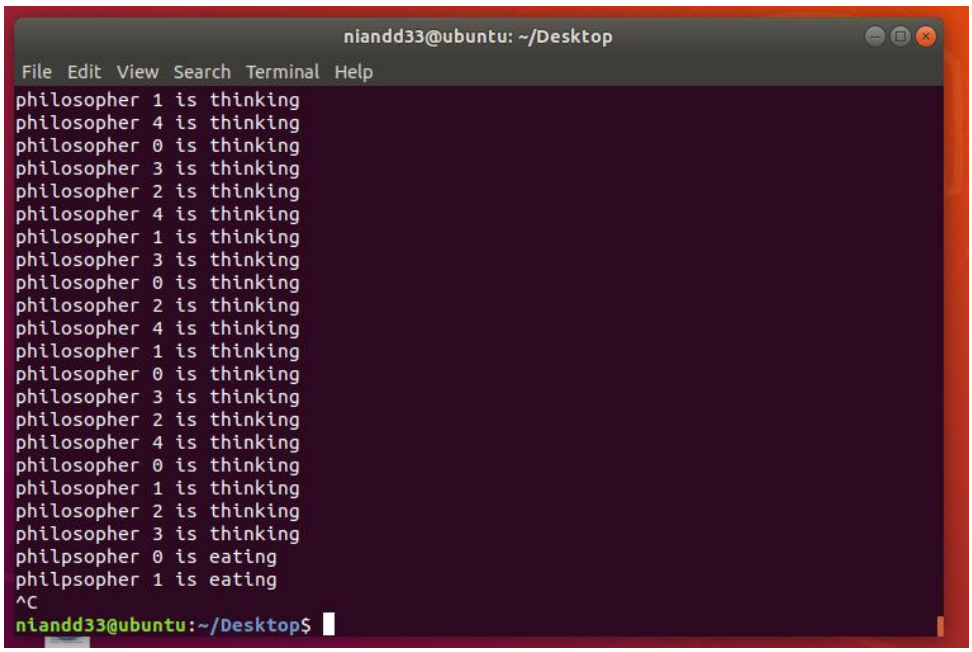
2. 要求通过 make，能输出 dph 文件，输出哲学家们的状态。打印结果，截屏放到作业报告中。

代码如下：

```
1.  #include<stdio.h>
2.  #include<pthread.h>
3.
4.  int chops = 5;
5.
6.  struct timespec timeout;
7.
8.  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
9.  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
10. pthread_mutex_t lockp = PTHREAD_MUTEX_INITIALIZER;
11.
12. void *pcon(void *num)
13. {
14.     int flag = 1;
15.     while(1)
16.     {
17.         flag =(flag ? 0:1);
18.         if(!flag)
19.         {
20.             pthread_mutex_lock(&lock);
21.             if(chops > 0)
22.             {
23.                chops--;
24.                printf("philosopher %d is thinking\n",(int)
   num);
25.             }
26.             pthread_mutex_unlock(&lock);
27.         }
28.         else
29.         {
30.             pthread_mutex_lock(&lock);
31.              if(pthread_cond_timedwait(&cond, &lock, &time
   out) == 0)
32.             {
33.                 if(chops > 0)
34.                 {
35.                     printf("philpsopher %d is eating\n",(
   int)num);
```

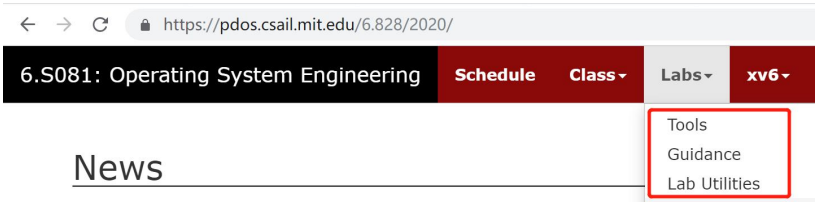
```
36.                         sleep(rand()%5);
37.                         chops++;
38.                     }
39.                 }
40.                 else
41.                 {
42.                     pthread_mutex_lock(&lockp);
43.                     chops++;
44.                     pthread_mutex_unlock(&lockp);
45.                 }
46.                 pthread_mutex_unlock(&lock);
47.                 pthread_cond_broadcast(&cond);
48.             }
49.             usleep(100);
50.         }
51. }
52.
53. int main(void)
54. {
55.     pthread_t tid[5];
56.     int i;
57.     timeout.tv_sec =rand()%3+1 ;
58.     timeout.tv_nsec = 0;
59.     for(i = 0; i < 5; i++)
60.         pthread_create(&tid[i], NULL, pcon,(void*)i);
61.     for(i = 0; i < 5; i++)
62.         pthread_join(tid[i], NULL);
63. }
```

运行结果：



3. （50 分）MIT 6.S081 课程实验

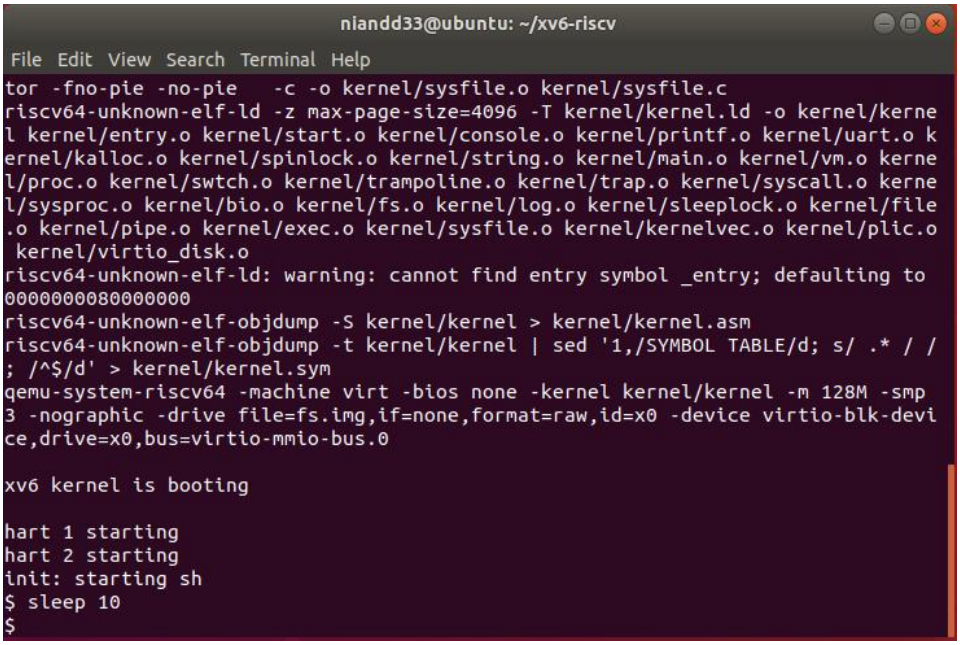
(1)（20 分）阅读 MIT 6.S081 项目介绍，如下图；完成 xv6 的安装和启动（Ctrl-a x 可退出）；完成 Lab: Xv6 and Unix utilities 中的 sleep（easy）任务，即在 user/下添加 sleep.c 文件。在报告中提供 sleep.c 的代码，并提供 sleep 运行的屏幕截图。提示：在 vmware 下安装 ubuntu20，可以较为顺利完成 xv6 安装和编译。



Sleep.c 代码：

```c
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if(argc != 2) {
    fprintf(2, "usage: sleep ticks\n");
    exit(1);
  }
  sleep(atoi(argv[1]));
  exit(0);
}
```

运行结果：



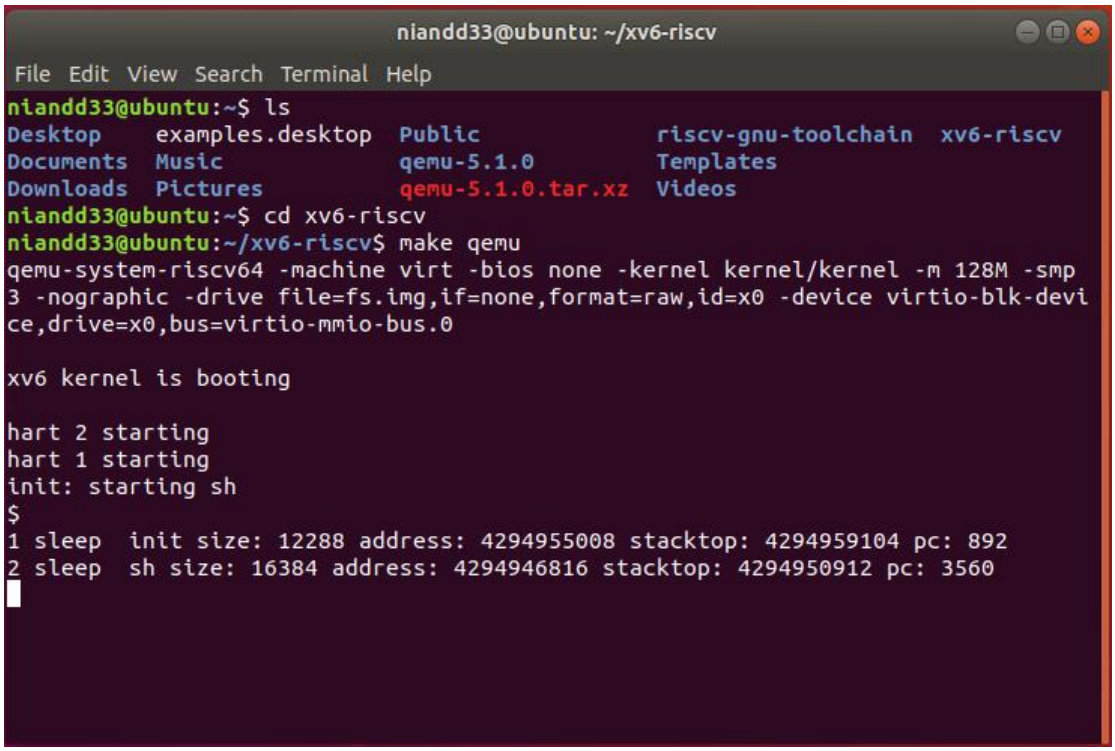
(2)(30 分)结合 xv6 book 第 1、2、7 章，阅读 xv6 内核代码(kernel/目录下)的进程和调度相关文件，围绕 swtch.S，proc.h/proc.c，理解进程的基本数据结构，组织方式，以及调度方法。提示：用 source insight 阅读代码较为方便。

(a)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。

procdump 函数：

```
1.  // Print a process listing to console.  For debugging.
2.  // Runs when user types ^P on console.
3.  // No lock to avoid wedging a stuck machine further.
4.  void
5.  procdump(void)
6.  {
7.    static char *states[] = {
8.    [UNUSED]    "unused",
9.    [SLEEPING]  "sleep ",
10.   [RUNNABLE]  "runble",
11.   [RUNNING]   "run   ",
12.   [ZOMBIE]    "zombie"
13.   };
14.   struct proc *p;
15.   char *state;
16.
17.   printf("\n");
18.   for(p = proc; p < &proc[NPROC]; p++){
19.     if(p->state == UNUSED)
20.       continue;
21.     if(p->state >= 0 && p->state < NELEM(states) && state
    s[p->state])
22.       state = states[p->state];
23.     else
24.       state = "???";
25.     printf("%d %s %s size: %u address: %u stacktop: %u pc:
     %u", p->pid, state, p->name, p->sz, p->kstack, p->trapfra
    me->kernel_sp, p->trapframe->epc);
26.     printf("\n");
27.   }
28. }
```

运行结果：



(b)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。

```
1.  //每个 CPU 都有 scheduler
2.  //每个 CPU 在初始化自身后都会调用进程调度函数
3.  //进程调度函数没有返回值，它不断循环，重复做着：
4.  //  -选择一个进程来运行
5.  //  -用 swtch 函数开始运行这个进程
6.  //  -最终该进程通过调用 swtch 函数将 cpu 控制权交还给 scheduler
7.  void
8.  scheduler(void)
9.  {
10.   struct proc *p;
11.   struct cpu *c = mycpu();
12.
13.   c->proc = 0;
14.   for(;;){
15.     //通过确保进程可以中断来避免死锁
16.     intr_on();
```

```
17.
18.       for(p = proc; p < &proc[NPROC]; p++) {
19.         acquire(&p->lock);//进程 p 获得锁
20.         if(p->state == RUNNABLE) {
21.           // 如果进程处于就绪状态，它会被选择。
22.           // 进程会释放锁，在返回之前重新获得锁
23.           p->state = RUNNING;//将进程 p 的状态设置为正在运行
24.           c->proc = p;//将本 cpu 当前运行的进程指针改为该被选中的进程指针
25.           swtch(&c->context, &p->context);//交换上下文
26.
27.           // 进程正在被执行，在返回前改变状态
28.           c->proc = 0;
29.         }
30.         release(&p->lock);//释放进程 p 的锁
31.       }
32.     }
33.   }
34.
35.   // 在进入 scheduler 之前必须持有 p->lock，
36.   // 而且 proc 的状态已经被改变，保存 intena
37.   // 因为 intena 的所有权属于内核线程而不是 CPU
38.   void
39.   sched(void)
40.   {
41.     int intena;//intena 用于判断在上锁之前系统中断是否已经被阻止
42.     struct proc *p = myproc();
43.
44.     if(!holding(&p->lock))//判断是否获得锁
45.       panic("sched p->lock");
46.     if(mycpu()->noff != 1)//判断是否被锁住
47.       panic("sched locks");
48.     if(p->state == RUNNING)//判断是否正在运行
49.       panic("sched running");
50.     if(intr_get())//判断是否可中断
51.       panic("sched interruptible");
52.
53.     intena = mycpu()->intena;//储存 intena
54.     swtch(&p->context, &mycpu()->context);//调用 swtch 函数将当前进程的上下文与 scheduler 的上下文进行切换
55.     mycpu()->intena = intena;//重新给 intena 赋值
56.   }
57.
```

```
58.  // 交出 CPU 使用权
59.  void
60.  yield(void)
61.  {
62.    struct proc *p = myproc();
63.    acquire(&p->lock);//获得锁
64.    p->state = RUNNABLE;// 将当前进程的状态从 RUNNING 改为其他
     状态
65.    sched();//进入 sched，交出 CPU 使用权
66.    release(&p->lock);//释放锁
67.  }
```

对进程调度框架的理解：

(1)xv6 进程调度框架，是基于时间片实现的，每个进程运行一个时间片就要交出对 cpu 的控制权。在调度过程中，每个进程的优先级是平等的，他的执行顺序仅与任务提交的先后顺序有关。

(2)一个进程让出 CPU 控制权有两种方式，一是调用 sleep 进入休眠状态，sleep 完成准备工作后调用 sched，sched 然后再调用 swtch。另一种方式是调用函数调用 yield，yield 再调用 sched，然后 sched 调用 swtch。

(3)用户进程之间互相切换并不是直接交换两个进程的上下文，而是通过一个中间的协调者。

(c)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。

不足：

（1）xv6 进程调度没有按照 CFS 的原则来排就绪队列。而是按照固定的每个任务被提交的顺序(即进程在进程表中的顺序)来排序。举例来说，当表中第 1 个进程正在 running 时，第 3 个进程处于就绪(RUNNABLE)状态，那么该进程就必须等待第 1 个进程交出 cpu 的控制权。但如果在等待的过程中第 2 个进程也完成了必要的工作从而处于就绪状态(RUNNABLE)，那么接下来调度器便不会把 cpu 的控制权优先交给先进入就绪队列的第 3 个进程，而是交给后来的第 2 个进程。因为第 2 个进程先被提交，在进程表中排在第 3 个进程前面。

（2）且 xv6 进程调度框架，是基于时间片实现的，每个进程运行一个时间片就要交出对 cpu 的控制权。所以当时间片非常小时，系统将把大量的时间都花在切换进程上，这会使系统的效率降低。

改进：

在进程结构体 struct proc 中新加一个变量:优先级 priority,先进入 RUNNABLE 状态的 process 优先级。scheduler 在进程表 proc[NPROC]中寻找下一个 RUNNABLE 的进程时，如果出现两个 RUNNABLE 的进程，会先比较它们的优先级，谁的优先级高，谁便先进入 RUNNING 状态；如果两个进程优先级相同，则比较它们在进程表 proc[NPROC]中的顺序，谁先被提交，谁便先 RUNNING。