SYSU 18364012 陈鑫锐

OS ASSIGNMENT1

一、生产者消费者问题

实验步骤：

1. 创建生产者 **prod** 和消费者 **cons** 两个进程

2. 每个进程创建 **3** 个线程

```
#define NUM_THREADS 3
pthread_t tid[NUM_THREADS];
pthread_attr_t attr;

pthread_attr_init(&attr);

if(pthread_attr_getscope(&attr,&scope)!=0)
    fprintf(stderr,"Unable to get scheduling scope\n");
else{
    if(scope==PTHREAD_SCOPE_PROCESS)
        printf("PTHREAD_SCOPE_PROCESS\n");
    else if(scope==PTHREAD_SCOPE_SYSTEM)
        printf("PTHREAD_SCOPE_SYSTEM\n");
    else
        fprintf(stderr,"Illegal scope value.\n");
}
pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);
for(i=0;i<NUM_THREADS;i++)
    pthread_create(&tid[i],&attr,producer,import);
for(i=0;i<NUM_THREADS;i++)
    pthread_join(tid[i],NULL);
```

3. 两个进程之间的缓冲最多容纳 **20** 个数据

```
#define BUFFER_SIZE 20
typedef int buffer_item;
struct buf{
        int rear;
        int front;
        buffer_item buffer[BUFFER_SIZE];
};
```

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

```
full = sem_open("full",O_CREAT,0666,0);//sem_open returns a sem_t pointer
empty = sem_open("empty",O_CREAT,0666,BUFFER_SIZE);
s_mutex = sem_open("mutex",O_CREAT,0666,1);

int shm_fd = shm_open("buffer",O_CREAT | O_RDWR,0666); //Create the shared memory
ftruncate(shm_fd,sizeof(struct buf));
ptr = mmap(0,sizeof(struct buf),PROT_WRITE,MAP_SHARED,shm_fd,0);//Mapped it into file, which can shared by different process
```

5. 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

```
void *producer(void *param){
    int lambda_p = *(int *)param;
    do{
        double interval_time =(double)lambda_p;
        usleep((unsigned int)(produce_time(interval_time)*1e6)); // sleep the time by the distribution of negative exponetial.
        buffer_item item = rand() % 255;
        struct buf *shm_ptr = ((struct buf *)ptr);// read the round queue's information from shared memory.
        sem_wait(empty); // If there is no empty buffer, block.
        sem_wait(s_mutex);//lock the binary-mutex and execute the code in critical area.
        printf("Producing the data %d to buffer[%d] by id %ld,%ld\n",item,shm_ptr->rear,(long)getpid(),(long)pthread_self());
        shm_ptr->buffer[shm_ptr->rear] = item; // Put the data to round queue.
        shm_ptr->rear = (shm_ptr->rear+1) % BUFFER_SIZE;
        sem_post(s_mutex);//Unlock the binary-mutex
        sem_post(full);// Add a full buffer
    }while(1);
    printf("end\n");
    pthread_exit(0);
}
```

6. 每个消费者线程取出一个数据，然后打印自己的 **id** 和数据

```
void *consumer(void *param){
	int lambda_c = *(int *)param;
	do{
		double interval_time =(double)lambda_c;
		sleep((unsigned int)produce_time(interval_time));
		struct buf *shm_ptr = ((struct buf*)ptr);
		sem_wait(full);//Wait for a full buffer
		sem_wait(s_mutex);
		buffer_item item = shm_ptr->buffer[shm_ptr->front];
		shm_ptr->front = (shm_ptr->front+1) % BUFFER_SIZE;
		sem_post(s_mutex);
		sem_post(empty);//Add a empty buffer
		printf("Consuming the data %d by pid %ld,%ld\n",item,(long)getpid(),(long)pthread_self());
	}while (1);
	pthread_exit(0);
}
```

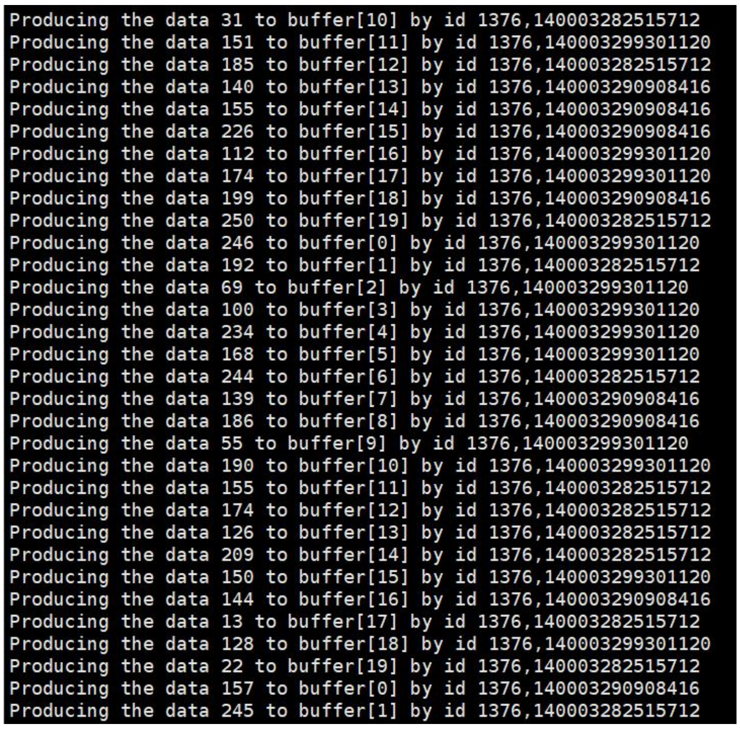
7. 生产者生成数据的间隔和消费者消费数据的间隔，按照负指数分布来控制，各有一个控制参数 λ **p**， λ **c**

```
double produce_time(double lambda_p){
	double z;
	do
	{
		z = ((double)rand() / RAND_MAX);
	}
	while((z==0) || (z == 1));
	return (-1/lambda_p * log(z));
}
```

8. 测试不同的参数组合λp，λc

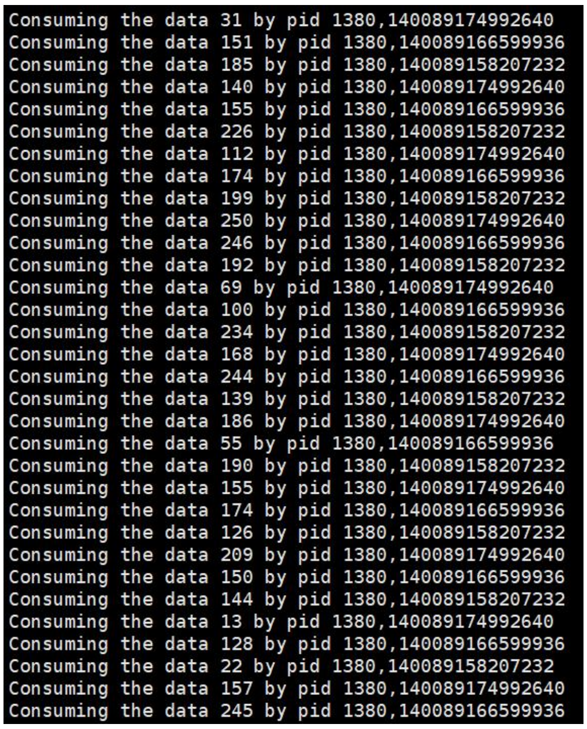
（1）λp=1，λc=5

生产者进程运行结果：



```
Producing the data 31 to buffer[10] by id 1376,140003282515712
Producing the data 151 to buffer[11] by id 1376,140003299301120
Producing the data 185 to buffer[12] by id 1376,140003282515712
Producing the data 140 to buffer[13] by id 1376,140003290908416
Producing the data 155 to buffer[14] by id 1376,140003290908416
Producing the data 226 to buffer[15] by id 1376,140003290908416
Producing the data 112 to buffer[16] by id 1376,140003299301120
Producing the data 174 to buffer[17] by id 1376,140003299301120
Producing the data 199 to buffer[18] by id 1376,140003290908416
Producing the data 250 to buffer[19] by id 1376,140003282515712
Producing the data 246 to buffer[0] by id 1376,140003299301120
Producing the data 192 to buffer[1] by id 1376,140003282515712
Producing the data 69 to buffer[2] by id 1376,140003299301120
Producing the data 100 to buffer[3] by id 1376,140003299301120
Producing the data 234 to buffer[4] by id 1376,140003299301120
Producing the data 168 to buffer[5] by id 1376,140003299301120
Producing the data 244 to buffer[6] by id 1376,140003282515712
Producing the data 139 to buffer[7] by id 1376,140003290908416
Producing the data 186 to buffer[8] by id 1376,140003290908416
Producing the data 55 to buffer[9] by id 1376,140003299301120
Producing the data 190 to buffer[10] by id 1376,140003299301120
Producing the data 155 to buffer[11] by id 1376,140003282515712
Producing the data 174 to buffer[12] by id 1376,140003282515712
Producing the data 126 to buffer[13] by id 1376,140003282515712
Producing the data 209 to buffer[14] by id 1376,140003282515712
Producing the data 150 to buffer[15] by id 1376,140003299301120
Producing the data 144 to buffer[16] by id 1376,140003290908416
Producing the data 13 to buffer[17] by id 1376,140003282515712
Producing the data 128 to buffer[18] by id 1376,140003299301120
Producing the data 22 to buffer[19] by id 1376,140003282515712
Producing the data 157 to buffer[0] by id 1376,140003290908416
Producing the data 245 to buffer[1] by id 1376,140003282515712
```

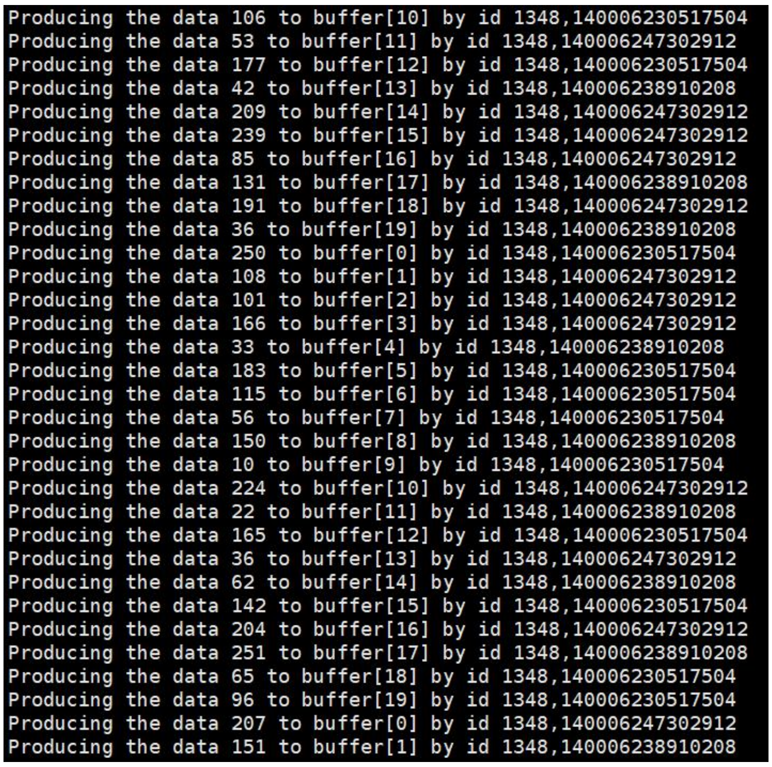
消费者进程运行结果：



```
Consuming the data 31 by pid 1380,140089174992640
Consuming the data 151 by pid 1380,140089166599936
Consuming the data 185 by pid 1380,140089158207232
Consuming the data 140 by pid 1380,140089174992640
Consuming the data 155 by pid 1380,140089166599936
Consuming the data 226 by pid 1380,140089158207232
Consuming the data 112 by pid 1380,140089174992640
Consuming the data 174 by pid 1380,140089166599936
Consuming the data 199 by pid 1380,140089158207232
Consuming the data 250 by pid 1380,140089174992640
Consuming the data 246 by pid 1380,140089166599936
Consuming the data 192 by pid 1380,140089158207232
Consuming the data 69 by pid 1380,140089174992640
Consuming the data 100 by pid 1380,140089166599936
Consuming the data 234 by pid 1380,140089158207232
Consuming the data 168 by pid 1380,140089174992640
Consuming the data 244 by pid 1380,140089166599936
Consuming the data 139 by pid 1380,140089158207232
Consuming the data 186 by pid 1380,140089174992640
Consuming the data 55 by pid 1380,140089166599936
Consuming the data 190 by pid 1380,140089158207232
Consuming the data 155 by pid 1380,140089174992640
Consuming the data 174 by pid 1380,140089166599936
Consuming the data 126 by pid 1380,140089158207232
Consuming the data 209 by pid 1380,140089174992640
Consuming the data 150 by pid 1380,140089166599936
Consuming the data 144 by pid 1380,140089158207232
Consuming the data 13 by pid 1380,140089174992640
Consuming the data 128 by pid 1380,140089166599936
Consuming the data 22 by pid 1380,140089158207232
Consuming the data 157 by pid 1380,140089174992640
Consuming the data 245 by pid 1380,140089166599936
```

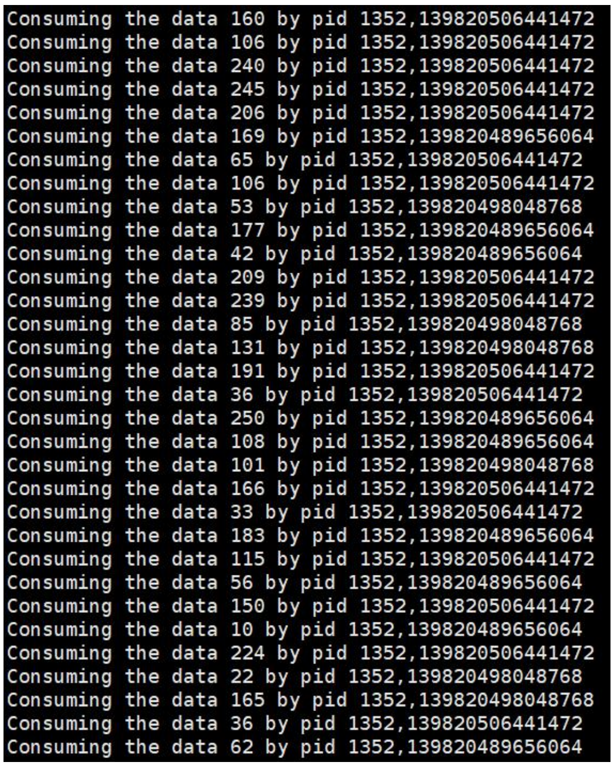
（2）λp=8，λc=1

生产者进程运行结果：



```
Producing the data 106 to buffer[10] by id 1348,140006230517504
Producing the data 53 to buffer[11] by id 1348,140006247302912
Producing the data 177 to buffer[12] by id 1348,140006230517504
Producing the data 42 to buffer[13] by id 1348,140006238910208
Producing the data 209 to buffer[14] by id 1348,140006247302912
Producing the data 239 to buffer[15] by id 1348,140006247302912
Producing the data 85 to buffer[16] by id 1348,140006247302912
Producing the data 131 to buffer[17] by id 1348,140006238910208
Producing the data 191 to buffer[18] by id 1348,140006247302912
Producing the data 36 to buffer[19] by id 1348,140006238910208
Producing the data 250 to buffer[0] by id 1348,140006230517504
Producing the data 108 to buffer[1] by id 1348,140006247302912
Producing the data 101 to buffer[2] by id 1348,140006247302912
Producing the data 166 to buffer[3] by id 1348,140006247302912
Producing the data 33 to buffer[4] by id 1348,140006238910208
Producing the data 183 to buffer[5] by id 1348,140006230517504
Producing the data 115 to buffer[6] by id 1348,140006230517504
Producing the data 56 to buffer[7] by id 1348,140006230517504
Producing the data 150 to buffer[8] by id 1348,140006238910208
Producing the data 10 to buffer[9] by id 1348,140006230517504
Producing the data 224 to buffer[10] by id 1348,140006247302912
Producing the data 22 to buffer[11] by id 1348,140006238910208
Producing the data 165 to buffer[12] by id 1348,140006230517504
Producing the data 36 to buffer[13] by id 1348,140006247302912
Producing the data 62 to buffer[14] by id 1348,140006238910208
Producing the data 142 to buffer[15] by id 1348,140006230517504
Producing the data 204 to buffer[16] by id 1348,140006247302912
Producing the data 251 to buffer[17] by id 1348,140006238910208
Producing the data 65 to buffer[18] by id 1348,140006230517504
Producing the data 96 to buffer[19] by id 1348,140006230517504
Producing the data 207 to buffer[0] by id 1348,140006247302912
Producing the data 151 to buffer[1] by id 1348,140006238910208
```

消费者进程运行结果：



```
Consuming the data 160 by pid 1352,139820506441472
Consuming the data 106 by pid 1352,139820506441472
Consuming the data 240 by pid 1352,139820506441472
Consuming the data 245 by pid 1352,139820506441472
Consuming the data 206 by pid 1352,139820506441472
Consuming the data 169 by pid 1352,139820489656064
Consuming the data 65 by pid 1352,139820506441472
Consuming the data 106 by pid 1352,139820506441472
Consuming the data 53 by pid 1352,139820498048768
Consuming the data 177 by pid 1352,139820489656064
Consuming the data 42 by pid 1352,139820489656064
Consuming the data 209 by pid 1352,139820506441472
Consuming the data 239 by pid 1352,139820506441472
Consuming the data 85 by pid 1352,139820498048768
Consuming the data 131 by pid 1352,139820498048768
Consuming the data 191 by pid 1352,139820506441472
Consuming the data 36 by pid 1352,139820506441472
Consuming the data 250 by pid 1352,139820489656064
Consuming the data 108 by pid 1352,139820489656064
Consuming the data 101 by pid 1352,139820498048768
Consuming the data 166 by pid 1352,139820506441472
Consuming the data 33 by pid 1352,139820506441472
Consuming the data 183 by pid 1352,139820489656064
Consuming the data 115 by pid 1352,139820506441472
Consuming the data 56 by pid 1352,139820489656064
Consuming the data 150 by pid 1352,139820506441472
Consuming the data 10 by pid 1352,139820489656064
Consuming the data 224 by pid 1352,139820506441472
Consuming the data 22 by pid 1352,139820498048768
Consuming the data 165 by pid 1352,139820498048768
Consuming the data 36 by pid 1352,139820506441472
Consuming the data 62 by pid 1352,139820489656064
```

9. 关闭并销毁命名信号量以及共享内存

```
void func(int signum){
	printf("\nKilling the producers...\n");
	for (int i=0;i<NUM_THREADS;i++){
		pthread_cancel(tid[i]);
	}
	sem_close(full);
	sem_close(empty);
	sem_close(s_mutex);
	sem_unlink("full");
	sem_unlink("empty");
	sem_unlink("mutex");
	shm_unlink("buffer");
	printf("Killed producers.\n");
}
```

注意：如若命名信号量以及共享内存未能及时销毁，再次运行生产者和消费者两个进程时，会出现内存访问错误 **segmentation fault(core dumped)**；而由于生产者和消费者的线程函数 **producer** 和 **consumer** 都使用了 **while(1)**的循环结构，两个进程之间的通信是无休止的，当我们人为地终止两个进程之间的通信时，程序无法执行到关闭并销毁命名信号量以及共享内存的程序段；本实验的解决方案是，在 **main** 函数中写一个信号处理重写函数 **signal(SIGINT,func)**，当我们按下 **Ctrl+C** 时，**Linux** 内核捕获该特殊信息，并向进程发送一个 **SIGINT** 信号。如果有线程处于被挂起的状态，**SIGINT** 信号将会被 **fun** 函数处理，从而实现线程的取消，解决了命名信号量以及共享内存的关闭和销毁问题。

二、哲学家就餐问题

解决方案：只有一个哲学家的两根筷子都可用时，他才能拿起他们（他必须在临界区内拿起两根筷子）

实验步骤：

1. 定义哲学家的三种基本状态

```
enum {THINKING,HUNGRY,EATING}state[5];
```

2. 使用条件使得哲学家们在不满足吃饭条件时进行等待，每一个条件变量配一把互斥锁

```
#define NUM_THREADS 5
pthread_cond_t self[NUM_THREADS];// Behalf on each philosophers
pthread_mutex_t mutex[NUM_THREADS]; // For conditional variables
```

3. 定义五个线程以及给线程函数传递的线程索引函数

```
pthread_t tid[NUM_THREADS]; /*the thread identifier*/
int id[NUM_THREADS]={0,1,2,3,4};// For philo function

pthread_attr_t attr; /*set of thread attributes*/
   /*get the default attributes*/
   pthread_attr_init(&attr);

   /*create the thread*/
   for(i=0;i<NUM_THREADS;i++)
      pthread_create(&tid[i],&attr,philo,ptr+i);

  /*wait for the thread to exit*/
   for(i=0;i<NUM_THREADS;i++)
      pthread_join(tid[i],NULL);
```

4. 定义哲学家函数

```
void *philo(void *param){
	do{
	int id = *( (int *)param);
	/* Try to pickup a chopstick*/
	pickup_forks(id);
	printf("The philosopher %d is eating...\n",id);
	/* Eat a while*/
	srand((unsigned)time(NULL));
	int sec = (rand()%((3-1)+1)) +1;// make sec in [1,3]
	sleep(sec);
	/* Return a chopstick */
	return_forks(id);
	printf("The philosopher %d is thinking...\n",id);
	/* Think a while */
	srand((unsigned)time(NULL));
	sec = (rand()%((3-1)+1)) +1;// make sec in [1,3]
	sleep(sec);
	}while(TRUE);
	pthread_exit(NULL);
}
```

5. 定义拿起筷子的函数

```
void pickup_forks(int i){
	state[i] = HUNGRY;// Wants to eat
	test(i);// Check can eat or not
	pthread_mutex_lock(&mutex[i]);
	while (state[i] != EATING){
		pthread_cond_(&self[i],&mutex[i]);//Wait his neighbors ate
	}
	pthread_mutex_unlock(&mutex[i]);
}
```

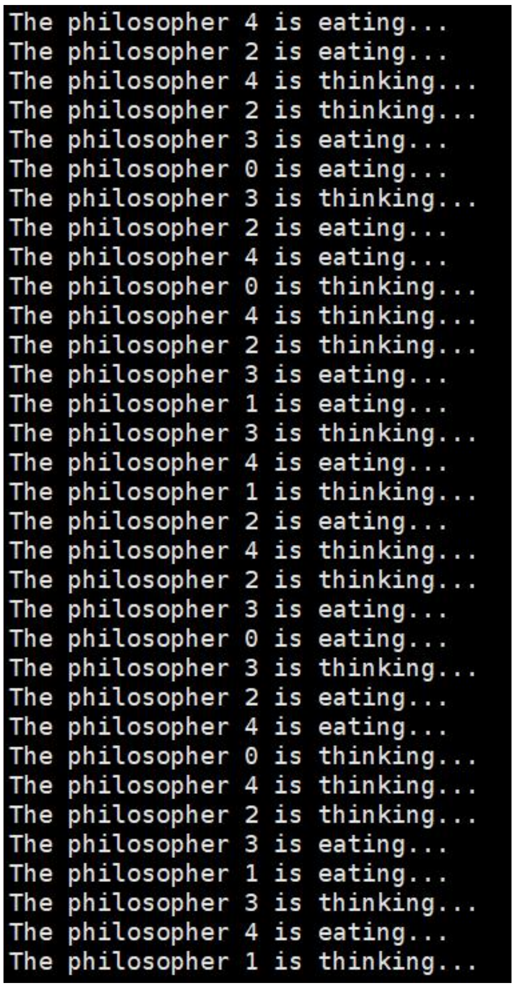
6. 定义放下筷子的函数

```
void return_forks(int i){
	state[i] = THINKING;
	//Notify his neighbor that I was ate.
	test((i+4)%5);
	test((i+1)%5);
}
```

7. 定义判断哲学家状态的函数

```
void test(int i){
	// A philosopher can eat when he wants to eat and his neighbors aren't eating.
	if ( (state[(i+4)%5] != EATING)&&
	(state[i] == HUNGRY) &&
	(state[(i+1)%5] != EATING)){
		pthread_mutex_lock(&mutex[i]);
		state[i] = EATING;
		pthread_cond_signal(&self[i]);
		pthread_mutex_unlock(&mutex[i]);
	}
}
```

8. 输出哲学家的状态



```
The philosopher 4 is eating...
The philosopher 2 is eating...
The philosopher 4 is thinking...
The philosopher 2 is thinking...
The philosopher 3 is eating...
The philosopher 0 is eating...
The philosopher 3 is thinking...
The philosopher 2 is eating...
The philosopher 4 is eating...
The philosopher 0 is thinking...
The philosopher 4 is thinking...
The philosopher 2 is thinking...
The philosopher 3 is eating...
The philosopher 1 is eating...
The philosopher 3 is thinking...
The philosopher 4 is eating...
The philosopher 1 is thinking...
The philosopher 2 is eating...
The philosopher 4 is thinking...
The philosopher 2 is thinking...
The philosopher 3 is eating...
The philosopher 0 is eating...
The philosopher 3 is thinking...
The philosopher 2 is eating...
The philosopher 4 is eating...
The philosopher 0 is thinking...
The philosopher 4 is thinking...
The philosopher 2 is thinking...
The philosopher 3 is eating...
The philosopher 1 is eating...
The philosopher 3 is thinking...
The philosopher 4 is eating...
The philosopher 1 is thinking...
```

9. 编写 Makefile 文件

说明：编写 **Makefile** 旨在通过命令行下的 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。

```
all : prod cons dph
.PHONY : clean

prod : prod.c
	gcc prod.c -lrt -lpthread -lm -o prod

cons : cons.c
	gcc cons.c -lrt -lpthread -lm -o cons

dph : dph.c
	gcc dph.c -lrt -lpthread -lm -o dph

clean :
	rm	prod
	rm	cons
	rm	dph
	rm	*.o
```

三、MIT 6.S081 课程实验

1. Sleep(easy)任务

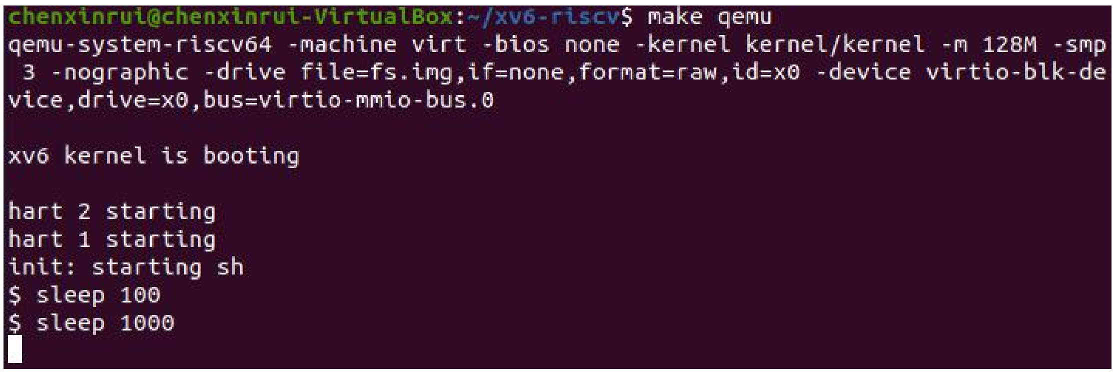
Codes:

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

int main(int argc,char *argv[]){
    int bedtime;
    if(argc<2){
        printf("Please input the sleeping time!\n");
        exit(0);
    }

    bedtime=atoi(argv[1]);
    if(bedtime>0){
        sleep(bedtime);
    }
    else{
        printf("Invalid sleeping time %s\n",argv[1]);
    }
    exit(0);
}
```

运行结果：



```
chenxinrui@chenxinrui-VirtualBox:~/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sleep 100
$ sleep 1000
```

2.

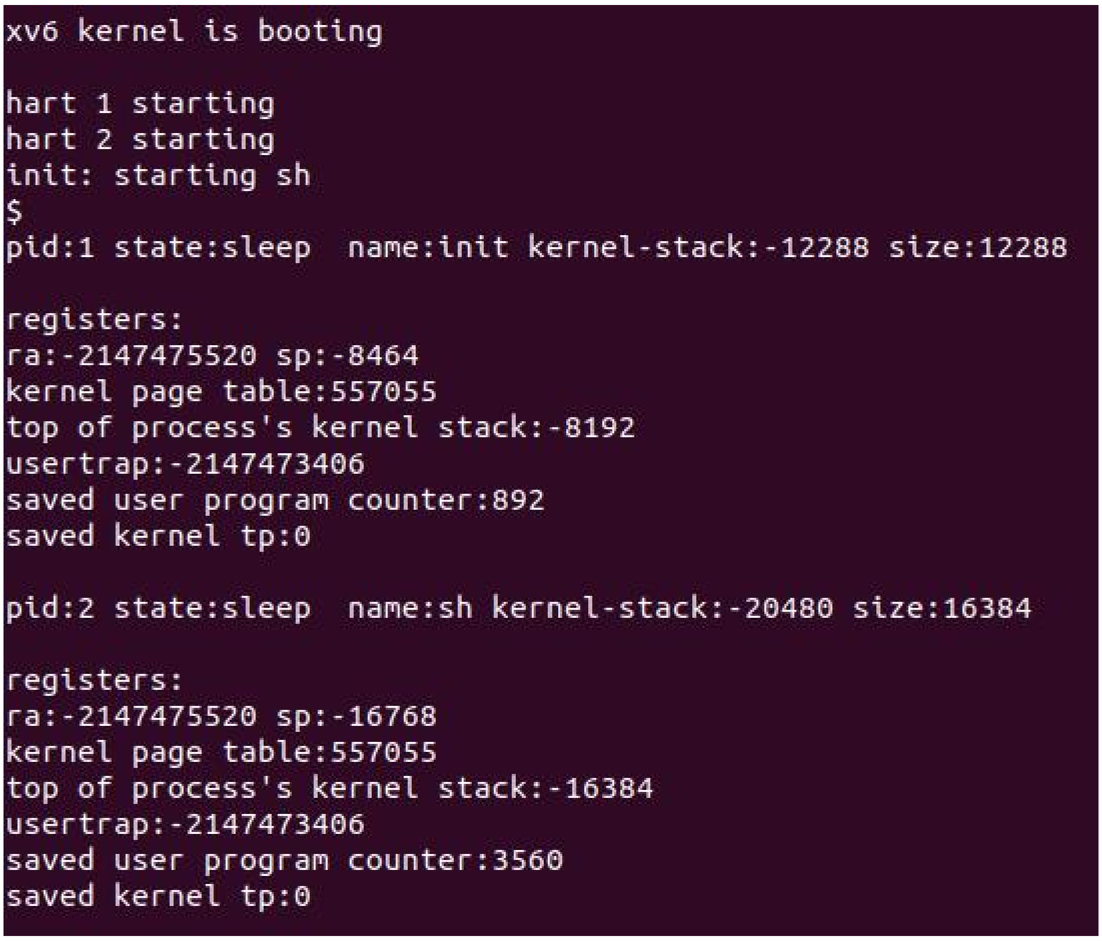
(a) 修改 proc.c 中 procdump 函数，打印各进程的扩展信息，包括大小（多少字节）、内核栈地址、关键寄存器内容等

Codes:

```
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  struct proc *p;
  char *state;

  printf("\n");
  for(p = proc; p < &proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
                        printf("pid:%d      state:%s      name:%s      kernel-stack:%d
size:%d\n",p->pid,state,p->name,p->kstack,p->sz);
    printf("\nregisters:\n");
    printf("ra:%d sp:%d\n",p->context.ra,p->context.sp);
    printf("kernel page table:%d\n",p->trapframe->kernel_satp);
    printf("top of process's kernel stack:%d\n",p->trapframe->kernel_sp);
    printf("usertrap:%d\n",p->trapframe->kernel_trap);
    printf("saved user program counter:%d\n",p->trapframe->epc);
    printf("saved kernel tp:%d\n",p->trapframe->kernel_hartid);
    printf("\n");
  }
}
```

运行结果：



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
pid:1 state:sleep  name:init kernel-stack:-12288 size:12288

registers:
ra:-2147475520 sp:-8464
kernel page table:557055
top of process's kernel stack:-8192
usertrap:-2147473406
saved user program counter:892
saved kernel tp:0

pid:2 state:sleep  name:sh kernel-stack:-20480 size:16384

registers:
ra:-2147475520 sp:-16768
kernel page table:557055
top of process's kernel stack:-16384
usertrap:-2147473406
saved user program counter:3560
saved kernel tp:0
```

(b) 结合 swtch.S，scheduler(void)，sched(void)，yield(void)等函数的核心部分谈谈对 xv6 中进程调度框架的理解

xv6 进程调度框架为了实现 CPU 的多线程化需要解决一系列问题。第一个问题是 CPU 如何在进程间切换？xv6 的解决方案是调用 scheduler(void) 和 sched(void)两个函数：CPU 启动之后，无限循环执行 scheduler 函数；当前进程通过调用 yield 函数进行进程切换；yield 函数调用 sched，sched 函数启用 swtch 函数完成进程切换。

sched 是一个死循环函数，该循环不断在进程表中扫描，从进程表中找到一个 RUNNABLE 的进程，通过 scheduler 切换器切换为进程的上下文，开始执行该进程；当进程运行结束时，调用 return 函数，切换回 CPU 的上下文，进入下一个循环。scheduler()、sched()以及 yield()函数的逐行代码解释如下；

scheduler(void):

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();//在每次执行一个进程之前，需要开启 CPU 的中断

//遍历进程表找到一个进程执行
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);//获取进程表的锁，避免其他 CPU 更改进程表
      if(p->state == RUNNABLE) {//如果进程表的状态为可以运行，则执行
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
//切换到选择的进程，释放进程表锁，当进程结束时，再重新获取
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);//释放进程表锁
    }
  }
}
```
//在每次循环之后，都要及时释放进程表锁，这样可以避免当进程表中暂时没有可以运行的程序时，进程表会一直被该 CPU 锁死，其他 CPU 便不能访问

sched(void):

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))//是否获取到了进程表锁
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)//判断程序是否仍处于运行状态
    panic("sched running");
  if(intr_get())//判断中断是否可以关闭
    panic("sched interruptible");

  intena = mycpu()->intena;
//上下文切换至 scheduler
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

yield(void):

```
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);//获取进程表锁
  p->state = RUNNABLE;//将进程状态设为可运行，以便下次遍历时可以被唤醒
  sched();//执行 sched 函数，准备将 CPU 切换到 scheduler context
  release(&p->lock);//释放进程表锁
}
```
//这里可能出现的一种情况是，调用 yield 之后，进程状态被改为了 RUNNABLE，但是它的上下文没有保存在 context 中，cpu->proc 还指向它，cpu 仍然在该栈上运行，scheduler 此时不能安全的调度它；因此，yield 在修改状态为 RUNNABLE 之前，必须先获取该进程的锁，避免了调度器线程切换到该进程的情况出现。

xv6 调度框架需要解决的第二个问题是如何让进程间的切换变得透明，要进行对用户进程透明的方式进行进程切换，就需要进行上下文切换，调用 swtch.S 函数实现上下文切换，代码如下：

swtch.S:

```
.globl swtch
swtch://两个参数分别指向旧进程 context 和新进程 context，保存在 a0、a1
//保存 callee-saved 寄存器，以及返回地址 ra，栈顶指针 sp
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)//都是 uint64，所以都是相隔 8 个字节
//加载恢复寄存器
        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret//将 ra 的内容设置为 pc，注意这里的 ra 是要切换到的进程之前执行 switch 的时候保存的 ra，也就是此次执行完 switch 之后，不是接着当前位置的下一行运行，而是另一处 switch 的下一行运行
```

xv6 进程调度框架需要解决的第三个问题是需要通过建立锁机制来避免竞争。对于多处理器架构而言，需要用到进程表的时候都需要事先获得表的锁，当结束之后再释放，这样保证了对进程表操作的原子化，可以避免多处理器的竞争问题。

xv6 进程调度框架需要面临的第四个问题是如何实现进程之间的自主切换。xv6 的解决方案是建立睡眠与唤醒机制：如果一个程序需要等待 IO，则 CPU 会将其设置为睡眠状态，此时不能被执行。当 IO 信号到达时，执行的进程会将 IO 信号对应的进程设置为 RUNNABLE，即唤醒。下一个 scheduler 周期的时候，该进程就可能会被执行，处理 IO 信号，sleep 函数和 wakeup 函数如下；

sleep();

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  
  // Must acquire p->lock in order to
  // change p->state and then call sched.
  // Once we hold p->lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup locks p->lock),
  // so it's okay to release lk.
//释放锁 lk
  if(lk != &p->lock){   //DOC: sleeplock0
    acquire(&p->lock);   //DOC: sleeplock1
    release(lk);
  }
//更改状态为 sleeping
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
//切换至 CPU context
  sched();

  // Tidy up.
  p->chan = 0;
//重新获得刚刚释放的 lk 锁
  // Reacquire original lock.
  if(lk != &p->lock){
    release(&p->lock);
```

```
        acquire(lk);
    }
}
void
wakeup(void *chan)
{
    struct proc *p;
//遍历进程表
    for(p = proc; p < &proc[NPROC]; p++) {
//先获取锁，确保 sleep 不会执行，避免出现 missed wakeup
        acquire(&p->lock);
        if(p->state == SLEEPING && p->chan == chan) {
            p->state = RUNNABLE;
//当发现有符合运行条件的程序时，将其标记为 RUNNABLE
        }
//唤醒结束，释放锁
        release(&p->lock);
    }
}
```

(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

1. CFS 进程调度算法基本思想

CFS 算法允许每个进程运行一段时间、循环轮转、选择运行最少 (并非物理运行时间，而是一个基于物理运行时间和 nice 值的虚拟运行时间 vruntime) 的进程作为下一个运行进程。CFS 不再采用“依靠 nice 值来计算时间片，然后分配给每个进程时间片”的做法，而是在所有可运行进程总数基础上计算出一个进程应该运行多久。 在这个过程中，nice 值被用于计算进程获得的处理器运行比的权重，越高的 nice 值 (越低的优先级) 进程获得更低的处理器使用权重。而 nice 值和权重并不是线性对应的，目的是使得降低一个 nice 值时，使用的 CPU 时间增加 10% (而不是时间片增加一段固定的时间)。

2. xv6 进程调度存在的不足

对照 Linux 的 CFS 进程调度算法和 xv6 的进程调度，我发现二者最大的差异在于选择下一个运行进程的方式：CFS 算法选择下一个进程是依赖一个基于物理运行时间和 nice 值的虚拟运行时间 vruntime。如果一个进程得以执行，随着时间的增长，其 vruntime 将不断增大。没有得到执行的进程 vruntime 不变。而调度器总是选择 vruntime 跑得最慢的那个进程来执行。为了区别不同优先级的进程，优先级高的进程 vruntime 增长得慢，以至于它可能得到更多的运行机会。因此，CFS 进程调度算法对于每一个进程来说是完全公平的；

相反，根据前面部分对于 xv6 进程调度框架的描述，可以看出 xv6 选择下一个运行进程的方式比较随机，仅仅是遍历所有的进程并寻找处于 RUNNABLE 状态的进程作为下一个运行进程。xv6 进程调度使用的是时间片轮转调度算法：时间片轮转法（Round-Robin，RR）主要用于分时系统中的进程调度。为了实现轮转调度，系统把所有就绪进程按先入先出的原则排成一个队列。新来的进程加到就绪队列末尾。每当执行进程调度时，进程调度程序总是选出就绪队列的队首进程，让它在 CPU 上运行一个时间片的时间。时间片是一个小的时间单位，通常为 10~100ms 数量级。当进程用完分给它的时间片后，系统的计时器发出时钟中断，调度程序便停止该进程的运行，把它放入就绪队列的末尾；然后，把

CPU 分给就绪队列的队首进程，同样也让它运行一个时间片，如此往复。由此可见 xv6 进程调度总是调度实时优先级最高的进程运行，运行时间和运行机会上都很不公平。这正是相比起 CFS 进程调度算法，xv6 进程调度存在的不足。

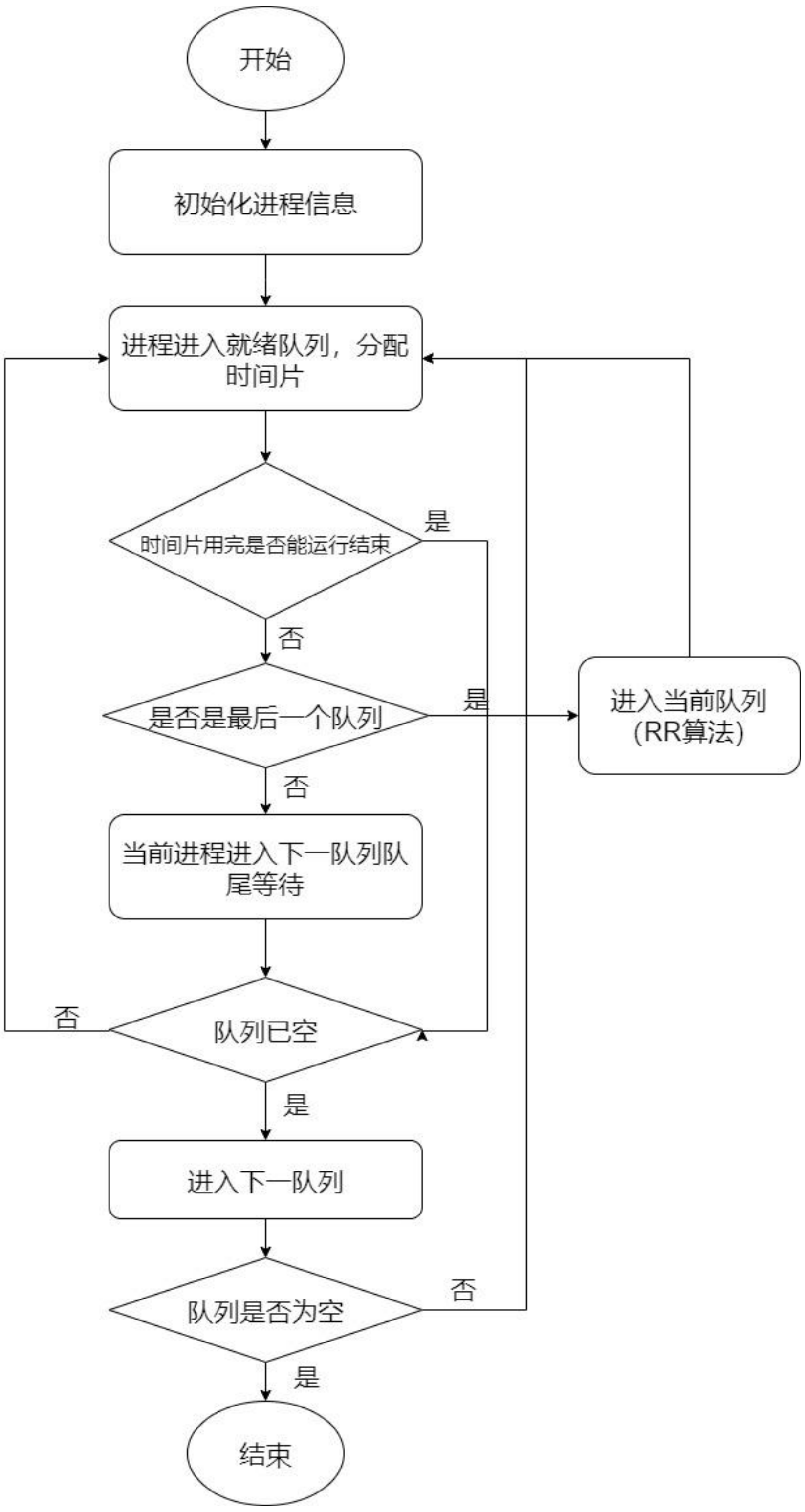
3. 进程调度框架设计

进程调度的核心问题是，当有若干个进程同时处于就绪状态，我们应当依照哪一种策略来决定哪些进程优先占用 CPU？相比起 xv6 使用的时间片轮转调度算法，公认的比较好的一种进程调度算法是多级反馈队列调度算法。其优势在于不必事先知道各种进程所需的执行时间，而且还可以满足各种进程的需要，本实验拟设计多级反馈队列进程调度框架，实施过程如下：

(1) 设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 i+1 个队列的时间片要比第 i 个队列的时间片长一倍；

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列便采取按时间片轮转的方式运行；

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 1～(i-1)队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。


开始
初始化进程信息
进程进入就绪队列，分配时间片
时间片用完是否能运行结束
是
否
是否是最后一个队列
是
进入当前队列(RR算法)
否
当前进程进入下一队列队尾等待
队列已空
否
是
进入下一队列
队列是否为空
否
是
结束