Experiments of Operating System 1

操作系统实验报告 1

2020.10.26

姓名：庞雨贤

学号： 18364075

邮箱：1909642325@qq.com

Problem 1: Producer-consumer problem

1.实验内容及要求

a. 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。

b. 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。

c. 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。

d. 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λp, λc

e. 运行的时候，开两个窗口，一个./prod λp，另一个./cons λc，要求测试不同的参数组合，打印结果，截屏放到作业报告里。

2.生产者-消费者问题问题分析

在同一个进程地址空间内执行两个线程。生产线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻挡，直到新的物品被生产出来。

该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。要解决该问题，就必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。该问题也能被推广到多个生产者和消费者的情形。

该问题需要注意的几点：在缓冲区为空时，消费者不能再进行消费；在缓冲区为满时，生产者不能再进行生产；在一个线程进行生产或消费时，其余线程不能再进行生产或消费等操作，即保持线程间的同步；注意条件变量与互斥锁的顺序。

由于前两点原因，因此需要保持线程间的同步，即一个线程消费（或生产）完，其他线

程才能进行竞争 CPU，获得消费（或生产）的机会。对于这一点，可以使用条件变量进行线程间的同步：生产者线程在 product 之前，需要 wait 直至获取自己所需的信号量之后，才会进行 product 的操作；同样，对于消费者线程，在 consume 之前需要 wait 直到没有线程在访问共享区（缓冲区），再进行 consume 的操作，之后再解锁并唤醒其他可用阻塞线程。在访问共享区资源时，为避免多个线程同时访问资源造成混乱，需要对共享资源加锁，从而保证某一时刻只有一个线程在访问共享资源。

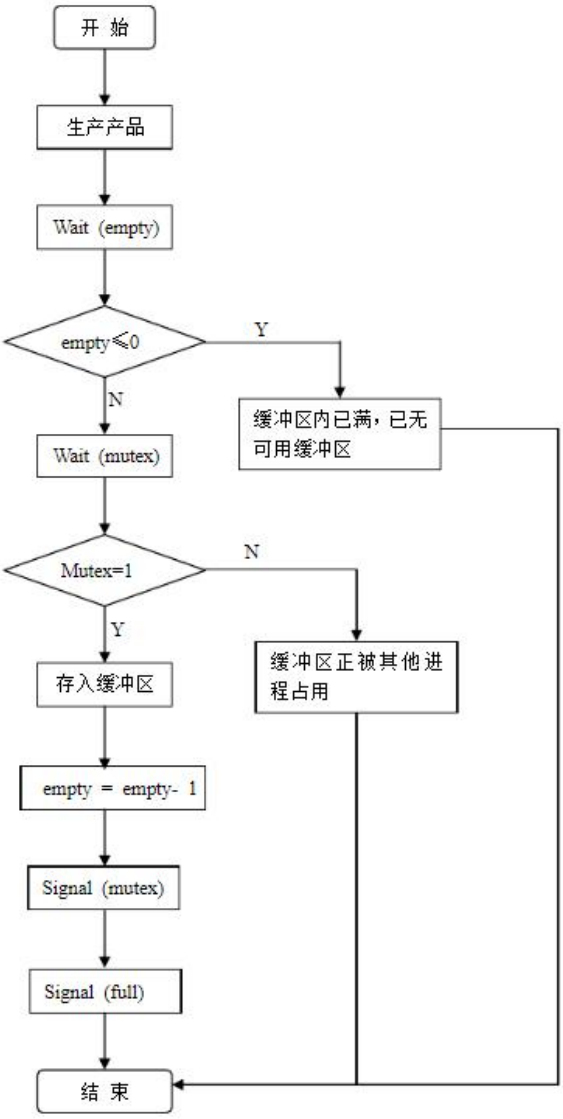
3.实验环境

操作系统环境：WSL+Ubuntu20.04

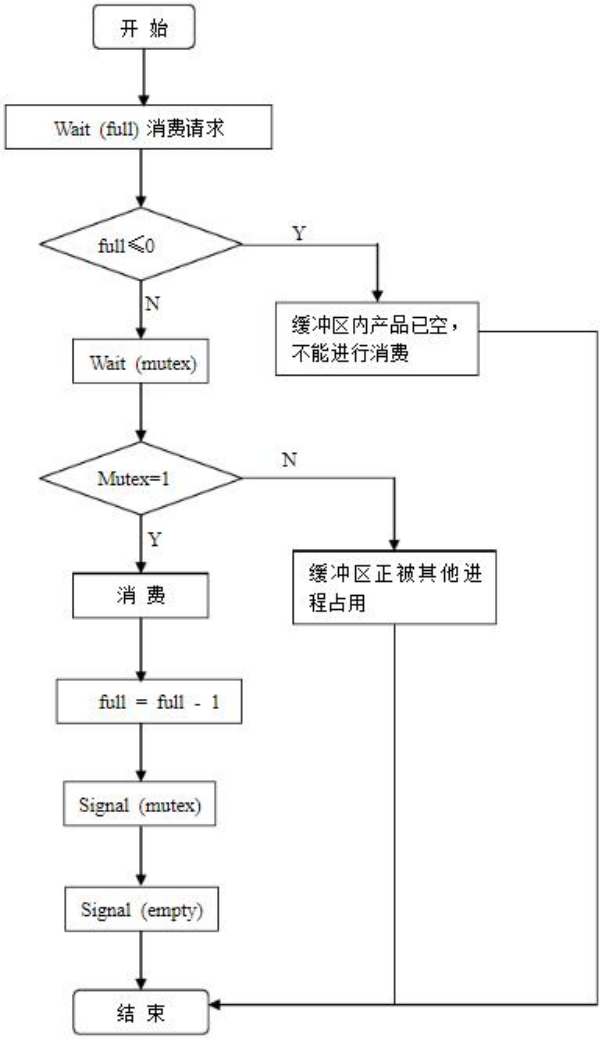
编程语言：C#

4.思路与设计

a.生产者



b.消费者



5.代码实现

定义缓冲区数据结构，使用循环队列来完成缓冲区定义：

```
typedef int buffer_item;
typedef struct buffer{
    int rear;
    int front;
    buffer_item buffer[BUFFER_SIZE];
}buffer;
```

生产者可以为消费者生产满的缓冲区，消费者可以为生产者生产空的缓冲区。定义两个信号量，以表示有多少个满的缓冲区，有多少个空的缓冲区。使用 sem_t（人为规定的二值信号量）对 buffer 上锁：

```
sem_t *full;
sem_t *empty;
sem_t *mutex;
```

我们让生产者按照负指数分布睡眠一段时间，返回一个符合负指数分布的随机变量的函数如下：

```
double sleep_time(double lambda_p){
    double r;
    r = ((double)rand() / RAND_MAX);

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

生产者函数：

```
void *producer(void *param){
    double lambda_p = *(double *)param;
    do{
        double interval_time = lambda_p;
        unsigned int sleepTime = (unsigned int)sleep_time(interval_time
);
        sleep(sleepTime);
        buffer_item item = rand() % 200;
        buffer *shm_ptr = ((buffer *)ptr);
        sem_wait(empty);
        sem_wait(mutex);
        printf("Sleep Time: %d s | Producing the data %d to buffer[%d]
by thread %ld in process %d.\n", sleepTime, item,shm_ptr->rear,gettid()
, getpid());
        shm_ptr->buffer[shm_ptr->rear] = item;
        shm_ptr->rear = (shm_ptr->rear+1) % BUFFER_SIZE;
        sem_post(mutex);
        sem_post(full);
    }while(1);
    pthread_exit(0);
}
```

消费者函数：

```
void *consumer(void *param){
    double lambda_c = *(double *)param;
    do{
        double interval_time = lambda_c;
        unsigned int sleepTime;
        sleepTime = (unsigned int)sleep_time(interval_time);
        sleep(sleepTime);
        buffer *shm_ptr = ((buffer *)ptr);
        sem_wait(full);
        sem_wait(mutex);
        buffer_item item  = shm_ptr->buffer[shm_ptr->front];
        printf("Sleep Time: %d s | Consuming the data %d from the buffe
r[%d] by the thread %ld in  the process %d.\n", sleepTime, item, shm_pt
r->front, gettid(), getpid());
        shm_ptr->front = (shm_ptr->front+1) % BUFFER_SIZE;
        sem_post(mutex);
        sem_post(empty);
    }while(1);
    pthread_exit(0);
}
```

生产者 main 函数实现：

```
int main(int argc, char *argv[])
{
    buffer bf;
    memset(&bf, 0, sizeof(buffer));

    double lambda_p = atof(argv[1]);//读取 lambda p 转化为数字

    //打开具名信号量并初始化
    full = sem_open("full", O_CREAT, 0666, 0);
    empty = sem_open("empty", O_CREAT, 0666, 0);
    mutex = sem_open("mutex", O_CREAT, 0666, 0);
    sem_init(full, 1, 0);
    sem_init(empty, 1, BUFFER_SIZE);
    sem_init(mutex, 1, 1);

    if (argc != 2){
        printf("The number of supplied arguments are false.\n");
        return -1;
    }
```

```
    if (atof(argv[1]) < 0){
        printf("The lambda entered should be greater than 0.\n");
        return -1;
    }

    //创建共享内存
    int shm_fd = shm_open("Buffer", O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(buffer));
    ptr = mmap(0, sizeof(buffer), PROT_WRITE, MAP_SHARED, shm_fd, 0);

    //创建 3 个线程
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++){
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i], &attr[i], producer, &lambda_p);
    }

    for (int j = 0; j < NUM_THREADS; j++){
        pthread_join(tid[j], NULL);
    }

    return 0;
}
```

消费者 main 函数实现：

```
int main(int argc, char *argv[])
{
    double lambda_c = atof(argv[1]);//读取 lambda c 转化为数字

    //打开具名信号量
    full = sem_open("full", O_CREAT, 0666, 0);
    empty = sem_open("empty", O_CREAT, 0666, 0);
    mutex = sem_open("mutex", O_CREAT, 0666, 0);

    if (argc != 2){
        printf("The number of supplied arguments are false.\n");
        return -1;
    }

    if (atof(argv[1]) < 0){
        printf("The lambda entered should be greater than 0.\n");
```

```
        return -1;
    }

    //创建共享内存
    int shm_fd = shm_open("Buffer", O_RDWR, 0666);
    ptr = mmap(0, sizeof(buffer), PROT_READ | PROT_WRITE, MAP_SHARED, s
hm_fd, 0);

    //创建 3 个线程
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++){
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i], &attr[i], consumer, &lambda_c);
    }

    for (int j = 0; j < NUM_THREADS; j++){
        pthread_join(tid[j], NULL);
    }

    return 0;
}
```

6.实验过程及结果

根据实验要求 e：运行的时候，开两个窗口，一个./prod λp，另一个./cons λc，要求测试不同的参数组合，打印结果。

a. 当λp = λc = 0.1 时，写入读取速度相似

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./prod 0.1
Sleep Time: 5 s | Producing the data 115 to buffer[0] by thread 221 in process
 219.
Sleep Time: 15 s | Producing the data 135 to buffer[1] by thread 222 in proces
s 219.
Sleep Time: 18 s | Producing the data 92 to buffer[2] by thread 220 in process
 219.
Sleep Time: 4 s | Producing the data 21 to buffer[3] by thread 222 in process
219.
Sleep Time: 3 s | Producing the data 27 to buffer[4] by thread 220 in process
219.
Sleep Time: 6 s | Producing the data 59 to buffer[5] by thread 222 in process
219.
Sleep Time: 4 s | Producing the data 126 to buffer[6] by thread 220 in process
 219.
Sleep Time: 24 s | Producing the data 26 to buffer[7] by thread 221 in process
 219.
Sleep Time: 1 s | Producing the data 136 to buffer[8] by thread 221 in process
 219.
Sleep Time: 0 s | Producing the data 168 to buffer[9] by thread 221 in process
 219.
Sleep Time: 1 s | Producing the data 29 to buffer[10] by thread 221 in process
 219.
Sleep Time: 1 s | Producing the data 130 to buffer[11] by thread 221 in proces
s 219.
Sleep Time: 1 s | Producing the data 123 to buffer[12] by thread 221 in proces
s 219.
Sleep Time: 10 s | Producing the data 135 to buffer[13] by thread 220 in proce
ss 219.
Sleep Time: 7 s | Producing the data 2 to buffer[14] by thread 220 in process
219.
Sleep Time: 9 s | Producing the data 58 to buffer[15] by thread 220 in process
```

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./cons 0.1
Sleep Time: 5 s | Consuming the data 115 from the buffer[0] by the thread 225 in
 the process 223.
Sleep Time: 15 s | Consuming the data 135 from the buffer[1] by the thread 226 in
  the process 223.
Sleep Time: 18 s | Consuming the data 92 from the buffer[2] by the thread 224 in
 the process 223.
Sleep Time: 2 s | Consuming the data 21 from the buffer[3] by the thread 224 in
the process 223.
Sleep Time: 16 s | Consuming the data 27 from the buffer[4] by the thread 225 in
 the process 223.
Sleep Time: 4 s | Consuming the data 59 from the buffer[5] by the thread 224 in
the process 223.
Sleep Time: 3 s | Consuming the data 126 from the buffer[6] by the thread 224 in
 the process 223.
Sleep Time: 8 s | Consuming the data 26 from the buffer[7] by the thread 224 in
the process 223.
Sleep Time: 14 s | Consuming the data 136 from the buffer[8] by the thread 225 in
  the process 223.
Sleep Time: 24 s | Consuming the data 168 from the buffer[9] by the thread 226 in
  the process 223.
Sleep Time: 6 s | Consuming the data 29 from the buffer[10] by the thread 224 in
 the process 223.
Sleep Time: 4 s | Consuming the data 130 from the buffer[11] by the thread 226 in
  the process 223.
Sleep Time: 9 s | Consuming the data 123 from the buffer[12] by the thread 225 in
  the process 223.
Sleep Time: 7 s | Consuming the data 135 from the buffer[13] by the thread 224 in
  the process 223.
Sleep Time: 10 s | Consuming the data 2 from the buffer[14] by the thread 224 in
 the process 223.
Sleep Time: 24 s | Consuming the data 58 from the buffer[15] by the thread 225 in
```

b. 当λp＞λc 时，写入速度比读取快（λp =0.5， λc = 0.1）

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./prod 0.5
Sleep Time: 1 s | Producing the data 115 to buffer[0] by thread 275
in process 273.
Sleep Time: 3 s | Producing the data 135 to buffer[1] by thread 274
in process 273.
Sleep Time: 3 s | Producing the data 186 to buffer[2] by thread 276
in process 273.
Sleep Time: 0 s | Producing the data 21 to buffer[3] by thread 276 i
n process 273.
Sleep Time: 1 s | Producing the data 27 to buffer[4] by thread 276 i
n process 273.
Sleep Time: 0 s | Producing the data 59 to buffer[5] by thread 276 i
n process 273.
Sleep Time: 2 s | Producing the data 126 to buffer[6] by thread 274
in process 273.
Sleep Time: 4 s | Producing the data 140 to buffer[7] by thread 275
in process 273.
Sleep Time: 0 s | Producing the data 136 to buffer[8] by thread 275
in process 273.
Sleep Time: 0 s | Producing the data 168 to buffer[9] by thread 275
in process 273.
Sleep Time: 0 s | Producing the data 29 to buffer[10] by thread 275
in process 273.
Sleep Time: 0 s | Producing the data 130 to buffer[11] by thread 275
 in process 273.
Sleep Time: 0 s | Producing the data 123 to buffer[12] by thread 275
 in process 273.
Sleep Time: 2 s | Producing the data 135 to buffer[13] by thread 274
 in process 273.
Sleep Time: 1 s | Producing the data 2 to buffer[14] by thread 274 i
n process 273.
```

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./cons 0.1
Sleep Time: 5 s | Consuming the data 115 from the buffer[0] by the t
hread 279 in  the process 277.
Sleep Time: 15 s | Consuming the data 135 from the buffer[1] by the
thread 280 in  the process 277.
Sleep Time: 18 s | Consuming the data 186 from the buffer[2] by the
thread 278 in  the process 277.
Sleep Time: 2 s | Consuming the data 21 from the buffer[3] by the th
read 278 in  the process 277.
Sleep Time: 16 s | Consuming the data 27 from the buffer[4] by the t
hread 279 in  the process 277.
Sleep Time: 4 s | Consuming the data 59 from the buffer[5] by the th
read 278 in  the process 277.
Sleep Time: 3 s | Consuming the data 126 from the buffer[6] by the t
hread 278 in  the process 277.
Sleep Time: 14 s | Consuming the data 140 from the buffer[7] by the
thread 279 in  the process 277.
Sleep Time: 8 s | Consuming the data 136 from the buffer[8] by the t
hread 278 in  the process 277.
Sleep Time: 24 s | Consuming the data 168 from the buffer[9] by the
thread 280 in  the process 277.
Sleep Time: 6 s | Consuming the data 29 from the buffer[10] by the t
hread 279 in  the process 277.
Sleep Time: 4 s | Consuming the data 130 from the buffer[11] by the
thread 280 in  the process 277.
Sleep Time: 9 s | Consuming the data 123 from the buffer[12] by the
thread 278 in  the process 277.
Sleep Time: 7 s | Consuming the data 135 from the buffer[13] by the
thread 279 in  the process 277.
Sleep Time: 10 s | Consuming the data 2 from the buffer[14] by the t
hread 279 in  the process 277.
```

c. 当λp＜λc 时，读取速度比写入快（λp =0.1， λc = 0.5）

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./prod 0.1
Sleep Time: 5 s | Producing the data 115 to buffer[11] by thread 282
 in process 281.
Sleep Time: 15 s | Producing the data 135 to buffer[12] by thread 28
4 in process 281.
Sleep Time: 18 s | Producing the data 92 to buffer[13] by thread 283
 in process 281.
Sleep Time: 4 s | Producing the data 21 to buffer[14] by thread 284
in process 281.
Sleep Time: 3 s | Producing the data 27 to buffer[15] by thread 283
in process 281.
Sleep Time: 4 s | Producing the data 59 to buffer[16] by thread 283
in process 281.
Sleep Time: 6 s | Producing the data 163 to buffer[17] by thread 284
 in process 281.
Sleep Time: 24 s | Producing the data 26 to buffer[18] by thread 282
 in process 281.
Sleep Time: 1 s | Producing the data 136 to buffer[19] by thread 282
 in process 281.
Sleep Time: 0 s | Producing the data 168 to buffer[0] by thread 282
in process 281.
Sleep Time: 1 s | Producing the data 29 to buffer[1] by thread 282 i
n process 281.
Sleep Time: 1 s | Producing the data 130 to buffer[2] by thread 282
in process 281.
Sleep Time: 1 s | Producing the data 123 to buffer[3] by thread 282
in process 281.
Sleep Time: 10 s | Producing the data 135 to buffer[4] by thread 284
 in process 281.
Sleep Time: 7 s | Producing the data 2 to buffer[5] by thread 284 in
 process 281.
Sleep Time: 24 s | Producing the data 58 to buffer[6] by thread 283
```

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./cons 0.5
Sleep Time: 1 s | Consuming the data 46 from the buffer[12] by the t
hread 287 in  the process 285.
Sleep Time: 3 s | Consuming the data 129 from the buffer[13] by the
thread 286 in  the process 285.
Sleep Time: 3 s | Consuming the data 57 from the buffer[14] by the t
hread 288 in  the process 285.
Sleep Time: 3 s | Consuming the data 95 from the buffer[15] by the t
hread 287 in  the process 285.
Sleep Time: 0 s | Consuming the data 145 from the buffer[16] by the
thread 288 in  the process 285.
Sleep Time: 4 s | Consuming the data 167 from the buffer[17] by the
thread 286 in  the process 285.
Sleep Time: 0 s | Consuming the data 164 from the buffer[18] by the
thread 287 in  the process 285.
Sleep Time: 2 s | Consuming the data 87 from the buffer[19] by the t
hread 288 in  the process 285.
Sleep Time: 0 s | Consuming the data 76 from the buffer[0] by the th
read 286 in  the process 285.
Sleep Time: 1 s | Consuming the data 178 from the buffer[1] by the t
hread 287 in  the process 285.
Sleep Time: 1 s | Consuming the data 3 from the buffer[2] by the thr
ead 288 in  the process 285.
Sleep Time: 0 s | Consuming the data 199 from the buffer[3] by the t
hread 287 in  the process 285.
Sleep Time: 1 s | Consuming the data 132 from the buffer[4] by the t
hread 286 in  the process 285.
Sleep Time: 1 s | Consuming the data 60 from the buffer[5] by the th
read 288 in  the process 285.
Sleep Time: 2 s | Consuming the data 12 from the buffer[6] by the th
read 288 in  the process 285.
Sleep Time: 4 s | Consuming the data 186 from the buffer[7] by the t
```

Problem 2：Dinning Philosophers problem

1.实验内容及要求

参考课本（第十版）第 7 章 project 3 的要求和提示

a.使用 POSIX 实现

b.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.哲学家就餐问题分析

有五个哲学家共用一张放有五把椅子的餐桌，每人坐在一把椅子上，桌子上有五个碗和五只筷子，每人两边各放一只筷子。哲学家们是交替思考和进餐的，饥饿时便试图取其左右最靠近他的筷子。条件：（1）只有拿到两只筷子时，哲学家才能吃饭；（2）如果筷子已被别人拿走，则必须等别人吃完后才能拿到筷子；（3）任意一个哲学家在自己未拿到两只筷子吃饭前，不会放下手中拿到的筷子。

假如所有的哲学家都同时拿起左侧筷子，看到右侧筷子不可用，又都放下左侧筷子，等一会儿，又同时拿起左侧筷子，如此这般，永远重复。对于这种情况，即所有的程序都在无限期地运行，但是都无法取得任何进展，即出现饥饿，所有哲学家都吃不上饭。假设规定哲学家在拿到左侧的筷子后，先检查右面的筷子是否可用。如果不可用，则先放下左侧筷子，等一段时间再重复整个过程。

然而当出现以下情形，在某一个瞬间，所有的哲学家都同时启动这个算法，拿起左侧的筷子，而看到右侧筷子不可用，又都放下左侧筷子，等一会儿，又同时拿起左侧筷子....如此这样永远重复下去。对于这种情况，所有的程序都在运行，但却无法取得进展，即出现饥饿，所有的哲学家都吃不上饭。

为了避免死锁，有三种解法为：

a.最多允许四个哲学家坐在桌子周围。

b.单数的哲学家先拿起左边的筷子，再拿起右边的筷子。吃完饭时先放下右边的筷子，再放下左边的筷子；双数的哲学家先拿起右边的筷子，再拿起左边的筷子。吃完饭时先放下左边的筷子，再放下右边的筷子。这样可以防止一种死锁情况的发生：所有哲学家都先拿起左边的筷子。

c.把哲学家分为三种状态，思考、饥饿和进食，并且一次拿到两只筷子，否则不拿。仅当一个哲学家左右两边筷子都可用时，才允许他拿筷子。这样要么一次占有两只筷子在吃面，然后释放所有资源；要么不占用资源。这样就不会导致死锁了。

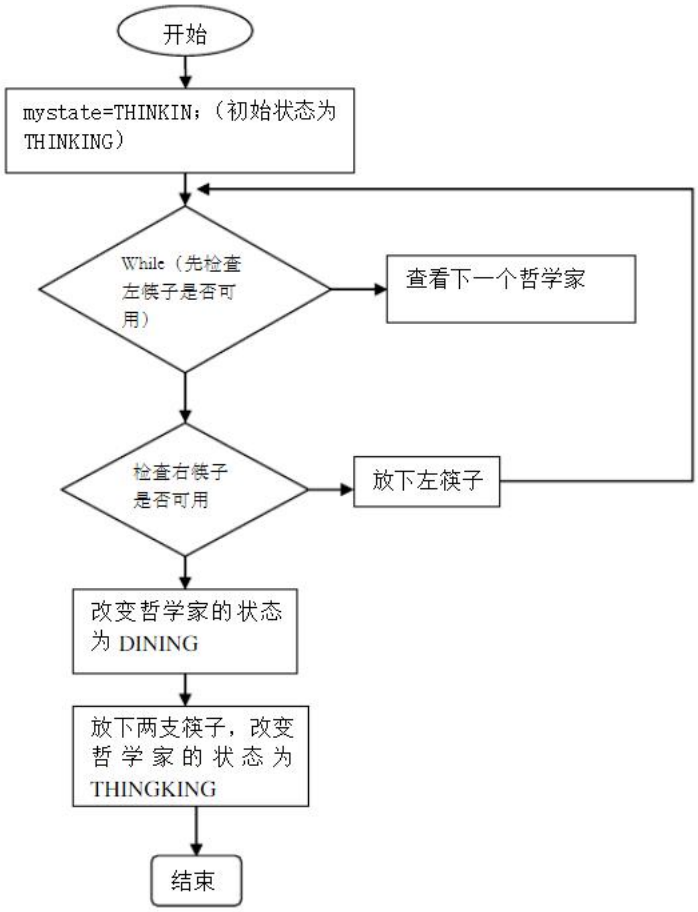
在这里我采用了第三种方法。

3.实验环境

操作系统环境：WSL+Ubuntu20.04

编程语言：C#

4.思路与设计



5.代码实现

首先我们定义哲学家的三种基本状态 THINKING,HUNGRY 和 EATING:

```
#define NUM_PH 5

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

使用条件变量来使得哲学家们在还不满足吃饭条件的时候进行等待，并给每一个哲学家

的条件变量配上一把互斥锁:

```
pthread_cond_t self[NUM_PH];
pthread_mutex_t mutex[NUM_PH];
```

哲学家函数，我们传递一个索引给线程函数，使其知道自己是第几号哲学家。然后哲学家试图拿起两只筷子，如果成功拿起两只筷子，就随机睡眠一段时间（模拟吃饭）。然后放下筷子，并随机睡眠一段时间（模拟思考），重复上述过程：

```
void *philo(void *param){
    do{
    int id = *( (int *)param);
    /* 尝试拿起筷子*/
    pickup_forks(id);
    printf("The philosopher %d is eating...\n",id);
    /* 吃一会*/
    srand((unsigned)time(NULL));
    int sec = (rand()%((3-1)+1)) +1;// 生成[1,3]随机数
    sleep(sec);
    /* 放下筷子 */
    return_forks(id);
    printf("The philosopher %d is thinking...\n",id);
    /* 思考一会 */
    srand((unsigned)time(NULL));
    sec = (rand()%((3-1)+1)) +1;// 生成[1,3]随机数
    sleep(sec);
    }while(1);
    pthread_exit(NULL);
}
```

拿起筷子，pickup_forks()函数接受一个参数 i，代表哲学家的索引。拿起筷子前首先要把哲学家的状态设为饥饿，这样他才有机会拿起筷子，然后调用 test 函数判断是否满足吃饭条件，即判断他的邻居们是否都不在吃饭。如果不满足上述条件，该哲学家（基于条件变量）将会被挂起，等待别人通知他可以吃饭（基于条件变量的唤醒）：

```
void pickup_forks(int i){
    state[i] = HUNGRY;// 等待吃饭
    test(i);// 检查是否可以吃饭
    pthread_mutex_lock(&mutex[i]);
    while (state[i] != EATING){
        pthread_cond_wait(&self[i],&mutex[i]);//等待邻居们吃完
    }
    pthread_mutex_unlock(&mutex[i]);
}
```

放下筷子，哲学家吃完之后会陷入思考阶段。并通知他的邻居们“我吃完饭了”：

```
void return_forks(int i){
    state[i] = THINKING;
    //通知邻居吃完了
    test((i+4)%5);
    test((i+1)%5);
}
```

test 函数：

```
void test(int i){
    // 当哲学家处于饥饿状态且他的邻居们不处于吃饭状态时 此哲学家可以吃饭
    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]);
    }

}
```

main 函数:

```
int main(){
    for(int i = 0; i < NUM_PH; i++){
        pthread_cond_init(&self[i], NULL);
        pthread_mutex_init(&mutex[i], NULL);
    }

    int id[NUM_PH];
    pthread_t tid[NUM_PH];
    pthread_attr_t attr[NUM_PH];

    for(int j = 0; j < NUM_PH; j++){
        id[j] = j;
        pthread_attr_init(&attr[j]);
        pthread_create(&tid[j], &attr[j], philo, &id[j]);
    }
    for(int k = 0; k < NUM_PH; k++){
        pthread_join(tid[k], NULL);
    }
```

```
    return 0;
}
```

6.实验结果打印

```
pyx@DESKTOP-ETHO4RO:~/os-assignment1$ ./dph
The philosopher 0 is eating...
The philosopher 2 is eating...
The philosopher 2 is thinking...
The philosopher 1 is eating...
The philosopher 4 is eating...
The philosopher 0 is thinking...
The philosopher 4 is thinking...
The philosopher 1 is thinking...
The philosopher 3 is eating...
The philosopher 0 is eating...
The philosopher 3 is thinking...
The philosopher 2 is eating...
The philosopher 0 is thinking...
The philosopher 4 is eating...
The philosopher 2 is thinking...
The philosopher 0 is eating...
The philosopher 4 is thinking...
The philosopher 3 is eating...
The philosopher 0 is thinking...
The philosopher 1 is eating...
The philosopher 3 is thinking...
The philosopher 4 is eating...
The philosopher 1 is thinking...
The philosopher 4 is thinking...
The philosopher 0 is eating...
The philosopher 2 is eating...
```

Problem 3：MIT S.081 experiment

实验一

1.实验内容及要求

阅读 MIT 6.S081 项目介绍，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。

2.实验环境

本次实验的系统环境为 XV6，Xv6 是由麻省理工学院(MIT)为操作系统工程的课程（代号 6.828）,开发的一个教学目的的操作系统。Xv6 是在 x86 处理器上(x 即指 x86)用 ANSI 标准 C 重新实现的 Unix 第六版(Unix V6，通常直接被称为 V6)。

3.实验过程

安装并启动 xv6：

```
pyx@DESKTOP-ETHO4RO:~/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-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

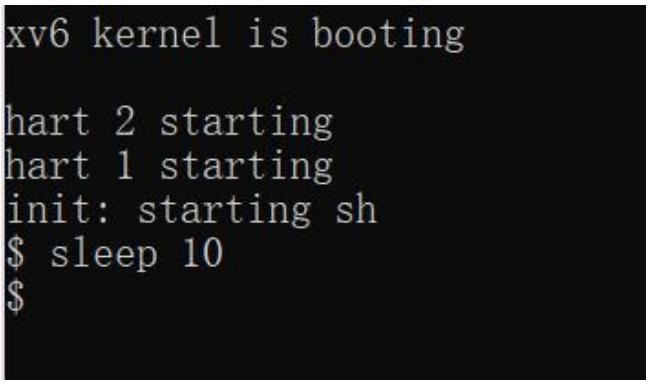
sleep.c 代码：

```c
#include <kernel/types.h>
#include <user/user.h>

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

    int x = atoi(argv[1]);
    sleep(x);
    exit(0);
}
```

sleep 运行截图：



运行./grade-lab-util sleep 对 sleep.c 进行验证：

```
pyx@DESKTOP-ETH04RO:~/xv6-riscv/xv6-labs-2020$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.6s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

实验二

1.实验内容及要求

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。

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。

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。

2.实验背景

调度：任何操作系统都可能碰到进程数多于处理器数的情况，这样就需要考虑如何分享处理器资源。理想的做法是让分享机制对进程透明。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。调度策略必须满足几个

相互冲突的目标:快速的进程响应时间、良好的后台作业吞吐量、避免进程饥饿、协调低优先级和高优先级进程的需求，等等。

Round-Robin: 在循环调度算法中，操作系统定义了一个时间量(片)。所有进程都将以循环方式执行。每个进程将获得 CPU 一小段时间(称为时间量)，然后回到就绪队列等待下一轮。

3.实验过程

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:

打印结果如图：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sleep 20

context_sp : 0x0000003fffffdef0
context_ra : 0x000000008000204a
process memory size(Byte) : 0x0000000000003000
pid = 1 state = sleep  process_name = init
virtual address of kernel stack : 0x0000003fffffd000
1 sleep  init
context_sp : 0x0000003fffffbef0
context_ra : 0x000000008000204a
process memory size(Byte) : 0x0000000000004000
pid = 2 state = sleep  process_name = sh
virtual address of kernel stack : 0x0000003fffffb000
2 sleep  sh
context_sp : 0x0000003fffff9f20
context_ra : 0x000000008000204a
process memory size(Byte) : 0x0000000000003000
pid = 3 state = sleep  process_name = sleep
virtual address of kernel stack : 0x0000003fffff9000
3 sleep  sleep
$
```

b.逐行对 swtch.S，scheduler(void)，sched(void)，yield(void)等函数的核心部分进行解释，写出对 xv6 中进程调度框架的理解:

scheduler（void）解析：

首先看看 scheduler 函数的头部注释：

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
```

通过直译注释，我们可以知道每个 CPU 都有相同的 scheduler，每个 cpu 在初始化自身之后就调用 scheduler，并且 scheduler 调度器永远不返回，它不断的循环，循环的内容是：

选择一个进程来运行；

用 swtch 函数来开始运行被选择的进程；

最终该进程同样通过调用 swtch 函数将 cpu 控制权交还给 scheduler。

对比 RR 算法的定义，xv6 的 scheduler 大致框架与 RR 类似，但没有提到就绪队列这一说，实际的代码实现复现就绪队列：

```
1void
2scheduler(void)
3{
4  struct proc *p;
5  struct cpu *c = mycpu();
6
7  c->proc = 0;
8  for(;;){ //无限循环
9    // Avoid deadlock by ensuring that devices can interrupt.
10    intr_on();
11
12    for(p = proc; p < &proc[NPROC]; p++) {
13      acquire(&p->lock);//上锁保户
14      if(p->state == RUNNABLE) {
15        // Switch to chosen process.  It is the process's job
16        // to release its lock and then reacquire it
17        // before jumping back to us.
18        p->state = RUNNING;//将进程状态修改为RUNNING
19        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
20        swtch(&c->context, &p->context);//上下文切换
21        // Process is done running for now.
22        // It should have changed its p->state before coming back.
23        c->proc = 0;
24      }
25      release(&p->lock);
26    }
27  }
28}
```

可以看到第 8 行至第 27 行就是注释中所提到的无限循环 loop，这使得 scheduler 永远不会返回。而第 12 行到第 26 行则是第二层循环。进入循环后，调度器获取了&p->lock，这可以防止其他 cpu 对进程指针表中进程状态作出修改，避免了数据冲突，调度器按顺序遍历进程表中的进程，当找到一个进程的状态(state)为可运行(RUNNABLE)时，就在第 18 行到第 19 行进行以下操作：

1.将进程的状态改为 RUNNING

2.将本 cpu 当前运行的进程指针改为该被选中的进程指针

完成上述操作后，就调用 swtch 函数，切换上下文。知道该进程因为某些原因(例如完成一个时间片的运行)调用 swtch 函数切换回 scheduler 的上下文。当进程 p 运行完成后，这时 scheduler 将从第 23 行开始运行，在将当前 cpu 上的运行进程置为空后，进入下一次循环。

可以看出来 xv6 系统实现的所谓 RR 算法虽然满足了轮询的要求，但是并没有按照 FCFS 的原则来排就绪队列。而是按照固定的顺序(即进程在进程表中的顺序)来排序。

xv6 系统的整个进程调度系统涉及到的函数包括但不限于：sleep，wakeup，wait，yield，sched，swtch，scheduler。为了更好的理解 xv6 系统的调度实现，我们首先需要对整个调度过程中使用到的函数及其作用有一个概括性的了解。

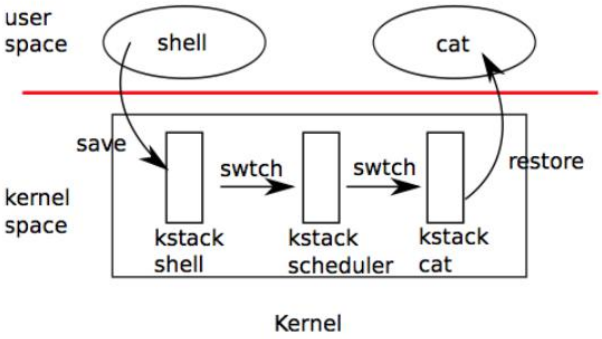


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

通过这张图我们能了解到，每次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，而是存在一个中间的协调者。我们知道在多进程的操作系统中，通过系统实现时的特殊技巧使得每一个进程都拥有完整的一套寄存器，内存栈空间，并且仿佛独占了一个 cpu。那么对应的，当我们在切换进程时，原进程占有的资源也应该是释放，问题是应该由谁来释放这些资源。若用最直接的想法，即在切换两个进程时直接切换他们的上下文，中间不存在任何中间函数。那么释放资源的任务只能交给被切换出 cpu 的进程本身。但一个进程是无法将自己占用的资源全部释放的，否则它连执行释放资源本身所需的资源也无法提供。那么这里就必须存在一个第三者来完成这一工作。现在我们再来看前文的这张图，就能理解为什么一个切换进程的操作需要经过三个中间节点了。

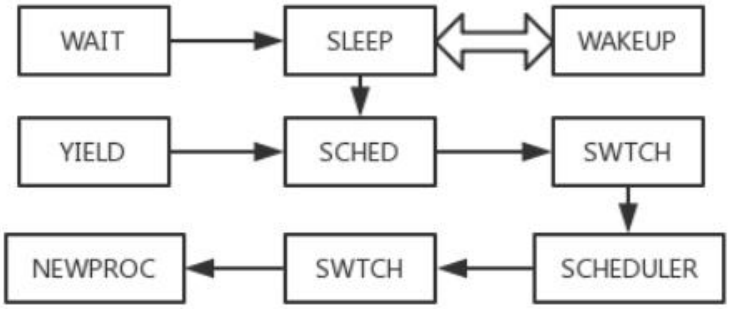
swtch.S 解析：

首先我们需要了解 xv6 调度中的一个最底层的关键函数 swtch：

```
1   # Context switch
2   #
3   #   void swtch(struct context *old, struct context *new);
4   #
5   # Save current registers in old. Load from new.
6
7
8   .globl swtch
9   swtch:
10      sd ra, 0(a0)
11      sd sp, 8(a0)
12      sd s0, 16(a0)
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
25      ld ra, 0(a1)
26      ld sp, 8(a1)
27      ld s0, 16(a1)
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)
39
40      ret
```

swtch 函数的实现语言是 RISCV-V 汇编语言，通过阅读它的头部注释我们能知道它的作用正是我们前文屡次提到的上下文交换。所谓上下文交换，就是在进程切换时保存上一个进程的寄存器等信息，然后载入新进程的寄存器等信息。可以看到第 10 到 23 行代码保存了上一进程的寄存器的信息，第 25 到 38 行载入了新进程的寄存器的信息。其中，a0 指的是返回的 swtch 函数的第一个参数，a1 指的是返回的 swtch 函数的第二个参数，ra 指的是返回的

跳转的地址，sp 指的是堆栈指针，s0 指的是保存 register 0，以此类推。



可以看到一个进程要交出 cpu 的控制权由两种方式，一个是通过调用 sleep 进入休眠状态，sleep 完成准备工作后将调用 sched。而 sched 将调用 swtch。而另一种方式是调用进程调用 yield。而 yield 也将调用 sched，之后的过程与上一种形式相同。

sched(void)分析：

首先我们阅读 sched 的头部注释：

```
// Switch to scheduler.  Must hold only p->lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->noff, but that would
// break in the few places where a lock is held but
// there's no process.
```

注释内容告诉我们，在进入 scheduler 之前必须支持有 ptable.lock 一个锁，并且 proc 的状态不能是 RUNNING，并且已经储存了当前进程的 intena，简要的来说就是判断在上锁之前系统中断是否已经被阻止，这对于进程释放所有锁后是否允许系统中断是很重要的判断依据，而切换进程后这个值将会被新进程的 intena 顶替，所以在此应该先储存 intena，再进入 scheduler。从 scheduler 回来后，再将 intena 赋值给 mycpu()->intena。保证数据不被损坏。

我们再来看具体的实现：

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  // 是否获取到了进程表锁
  if(!holding(&p->lock))
    panic("sched p->lock");
  // 是否执行过 pushcli
  if(mycpu()->noff != 1)
```

```
    panic("sched locks");
  // 执行的程序应该处于结束或睡眠状态
  if(p->state == RUNNING)
    panic("sched running");
  // 判断中断是否可以关闭
  if(intr_get())
    panic("sched interruptible");
  intena = mycpu()->intena;
  // 上下文切换至scheduler
  swtch(&p->context, &mycpu()->context);
  mycpu()->intena = intena;
}
```

我们可以看到 sched 的实现基本上就是检查了注释中提到的几个条件是否符合，并储存了 intena，然后直接调用了 swtch 函数将当前进程的上下文与 scheduler 的上下文进行切换。在 scheduler 再次选择当前进程来占有 cpu 时，scheduler 将调用 swtch 恢复该进程的上下文，即运行最后一行代码，重新给 intena 赋值。

yield(void)分析：

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  // 获取进程表锁
  struct proc *p = myproc();
  acquire(&p->lock);
 // 将进程状态设为可运行，以便下一次便利时可以被唤醒
  p->state = RUNNABLE;
  // 执行sched函数，准备将CPU切换到scheduler context
  sched();
  // 释放进程表锁
  release(&p->lock);
}
```

从 yield 函数简单的头部注释我们可以知道，yield 的作用是结束当前时间片的运行，交出 cpu 的使用权。通过对 sched 的解析我们知道进入 sched 时必须持有 ptable.lock，并且将当前进程的状态从 RUNNING 改为其他状态。

xv6 调度算法流程：

当 CPU 启动之后，执行 scheduler 函数，无限循环。在每个周期里，从固定进程表中找到一个 RUNNABLE 的进程 p，用 swtch 函数切换为进程的上下文，此时开始执行进程 p。当 p 进程运行结束时，调用 yield 函数使当前 CPU 上的进程 p 放弃 CPU，然后进入 sched 函数

切换到调度程序，此时切换为 CPU 的上下文，开始下一循环。

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:

CFS（完全公平调度算法）不跟踪进程的睡眠时间，也不企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。假设 runqueue 中有 n 个进程，当前进程运行了 10ms。在“完全理想的多任务处理器”中，10ms 应该平分给 n 个进程(不考虑各个进程的 nice 值)，因此当前进程应得的时间是(10/n)ms，但是它却运行了 10ms。所以 CFS 将惩罚当前进程，使其它进程能够在下次调度时尽可能取代当前进程。最终实现所有进程的公平调度。CFS 没有将任务维护在链表式的运行队列中，而是对每个 CPU 维护一个以时间为顺序的红黑树。红黑树可以始终保持平衡，这意味着树上没有路径比任何其他路径长两倍以上。由于红黑树是二叉树，查找操作的时间复杂度为 O(log n)。但是除了最左侧查找以外，很难执行其他查找，并且最左侧的节点指针始终被缓存。对于大多数操作（插入、删除、查找等），红黑树的执行时间为 O(log n)，而以前的调度程序通过具有固定优先级的优先级数组使用 O(1)。O(log n) 行为具有可测量的延迟，但是对于较大的任务数无关紧要。Molnar 在尝试这种树方法时，首先对这一点进行了测试。红黑树可通过内部存储实现，即不需要使用外部分配即可对数据结构进行维护。要实现平衡，CFS 使用“虚拟运行时”表示某个任务的时间量。任务的虚拟运行时越小，意味着任务被允许访问服务器的时间越短，其对处理器的需求越高。CFS 还包含睡眠公平概念以便确保那些目前没有运行的任务（例如，等待 I/O）在其最终需要时获得相当份额的处理器。

由 b 中 scheduler(void)分析可以看出来 xv6 系统的调度算法大致框架与 RR 算法类似，RR 算法指的是系统将所有的进程按照 FCFS 算法排成一个就绪队列，然后设定一个时间片的大小，当当前进程运行完一个时间片大小的时间时，就产生一次时间中断，把 cpu 的使用权交还给调度器 scheduler。scheduler 拿到使用权后，在就绪队列里找到队首进程，将 cpu 的使用权分配给该进程。当该进程在 cpu 上运行完一个时间片后，再度重复上述操作。这就是 RR 算法的基本思想。RR 调度算法中每个进程的优先级是平等的，它们的执行顺序仅仅与任务提交的先后有关，并且每个进程执行完一个时间片后都会让出 cpu 的使用权，因此即便先提交了一个非常耗时的任务，后续提交的任务也能够在可以预期的时间内获取 cpu 的计算资源。然而 xv6 系统实现的所谓 RR 算法虽然满足了轮询的要求，但是并没有按照 FCFS 的原则来排就绪队列。而是按照固定的顺序(即进程在进程表中的顺序)来排序。举例来说，当表中第 a 个进程正在运行(RUNNING)时，第 a+2 个进程处于就绪(RUNNABLE)状态，那

么该进程就必须等待第 a 个进程交出 cpu 的控制权。但如果在等待的过程中第 a+1 个进程也完成了必要的工作从而处于就绪状态(RUNNABLE)，那么接下来调度器将不会把 cpu 的控制权优先交给先进入就绪队列的第 a+2 个进程，而是交给后来的第 a+1 个进程。这个例子说明了 xv6 中就绪队列的排列方式不遵守 FCFS 的原则，而是按照进程表中的固定顺序。总的来说，这样的偏差并不会引起饥饿的出现，该实现能够保证每个处于就绪队列中的进程能在可以预期的时间内获取 cpu 的使用权。然而，此调度算法性能很大程度上取决于一个时间片的大小，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。

与 CFS 算法对比，xv6 系统当一个进程占用 CPU 时，其它进程就必须等待，这就产生了不公平；CFS 把所有进程都维护在一颗红黑树上，以 vruntime 为权值，每次运行 vruntime 值最小的进程，xv6 系统中所有进程的优先级是平等的，它们的执行顺序仅仅与任务提交的先后有关，并且每个进程执行完一个时间片后都会让出 cpu 的使用权，这样会导致系统进程无法优先处理，同时需要实时性的进程被拖慢，运行时间长的进程将会被频繁地切换上下文。

根据上述分析我们可以从以下几点进行改进：

a)为每个进程添加优先级属性

可以在 proc.h 中定义相关字段 priority 记录进程优先级。可以规定 priority 的数值越大，进程的优先级越高，并且 priority 的数值范围是 0 到 10。采用的办法是使用与缺省 scheduler 相似的方式，即遍历 ptable。但与原 scheduler 不同的是，原来的调度器在找到第一个 RUNNABLE 进程后，就确定它为目标进程，并开始切换虚拟内存空间等一系列操作。我们为了实现基于 priority 的调度，在进入 ptable 前设置一个名为 priority 的 flag，其值为-1，进入 ptable 后，对于找到的每个 RUNNABLE 进程，都取其 priority 值与当前的 flag 进行比较，如果该进程的值较大，则选择该进程指针作为待定的目标指针，并将 flag 更新为该指针的 priority 值。若该进程的值等于当前的 flag 值，那么就比较两者的 p->start，由于两者的状态都为 RUNNABLE，因此 p->start 的值就代表了双方进入队列的时间点。当该进程的 p->start 小于当前待定的目标进程的 start 时，就将该进程选为新的待定进程。值得注意的是，在遍历 ptable 退出内层循环后，我们需要判断 priority 是否还是-1，若还是-1 说明 ptable 中没有可以运行的进程。若不是-1，那么我们只需要仿造缺省的 scheduler 的办法完成准备工作，并把 cpu 的使用权移交给目标进程即可。以上就是基于 priority 实现的优先级调度，在优先级相同时采取 FCFS 原则。

b)禁用时钟中断

我们要实现的是一个非抢占式的优先级调度，因此单靠替换 scheduler 只能解决调度时

的优先级选择问题。如果不禁用时间中断，我们实现的调度算法就更像是一个特殊的抢占式优先级算法，因此我们需要禁用时间中断。