这份文件是 OS_homework_1 by 计科 210X 甘晴 void 202108010XXX
文档设置了目录，可以通过目录快速跳转至答案部分。

目录

第四章

4.1 用以下标志运行程序：./process-run.py -l 5:100,5:100。CPU 利用率（CPU 使用时间的百分比）应该是多少？为什么你知道这一点？利用 -c 标记查看你的答案是否正确。

```
wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 5:100,5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Process 1
  cpu
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 5:100,5:100 -c
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2        RUN:cpu         READY             1
  3        RUN:cpu         READY             1
  4        RUN:cpu         READY             1
  5        RUN:cpu         READY             1
  6           DONE       RUN:cpu             1
  7           DONE       RUN:cpu             1
  8           DONE       RUN:cpu             1
  9           DONE       RUN:cpu             1
 10           DONE       RUN:cpu             1
```

两进程都只使用 cpu。每一时刻都使用到 cpu，故 cpu 利用率为 100%。

使用-c 查看发现符合预期

4.2 现在用这些标志运行： ./process-run.py -l 4:100,1:0 这些标志指定了一个包含 4 条指令的进程(都要使用 CPU),并且只是简单地发出 IO 并等待它完成。完成这两个进程需要多长时间?利用 c 检查你的答案是否正确。

```
wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 4:100,1:0
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu

Process 1
  io
  io_done

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 4:100,1:0 -c
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2        RUN:cpu         READY             1
  3        RUN:cpu         READY             1
  4        RUN:cpu         READY             1
  5           DONE        RUN:io             1
  6           DONE       BLOCKED                           1
  7           DONE       BLOCKED                           1
  8           DONE       BLOCKED                           1
  9           DONE       BLOCKED                           1
 10           DONE       BLOCKED                           1
 11*          DONE   RUN:io_done             1
```

进程完成需要的时间与 I/O 等待完成需要的时间有关。若设这个时间为 t0，则两个进程需要的时间 t=4+t0+2。(表中假设 t0 为 5 个单位时间，两进程所需时间为 11)

4.3 现在交换进程的顺序：./process-run.py -l 1:0,4:100 现在发生了什么？交换顺序是否重要？为什么？同样，用 -c 看看你的答案是否正确。

```
wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 1:0,4:100
Produce a trace of what would happen when you run these processes:
Process 0
  io
  io_done

Process 1
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 1:0,4:100 -c
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7*   RUN:io_done          DONE             1
```

两个进程与第二题一致，只是把顺序交换了一下。但是可以看到，由于进程 1 的 I/O 使其进入阻塞状态，进程 2 补齐了这个空闲的时间，而导致总体的时间缩短了。

由此可见，合理地交换顺序十分重要，这样可以大大提高 cpu 利用率和效率。

使用-c 可以发现答案正确。

4.4 现在探索另一些标志。一个重要的标志是 -S，它决定了当进程发出 IO 时系统如何反应。将标志设置为 SWITCH_ON_END，在进程进行 I/O 操作时,系统将

不会切换到另一个进程,而是等待进程完成。当你运行以下两个进程时，会发生什么情况？一个执行 I/O，另一个执行 CPU 工作。（-l 1:0,4:100 -c -S SWITCH_ON_END）

```
wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 1:0,4:100 -c -S SWITCH_ON_END
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED         READY                           1
  3        BLOCKED         READY                           1
  4        BLOCKED         READY                           1
  5        BLOCKED         READY                           1
  6        BLOCKED         READY                           1
  7*   RUN:io_done         READY             1
  8           DONE       RUN:cpu             1
  9           DONE       RUN:cpu             1
 10           DONE       RUN:cpu             1
 11           DONE       RUN:cpu             1
```

使用标记阻止切换。这种情况下系统将停止并等待 I/O 操作结束，再进行下一个进程。这种方式的时间与第二题的时间相同，cpu 利用率很低。

4.5 现在,运行相同的进程，但切换行为设置，在等待 IO 时切换到另一个进程（-l 1:0,4:100 c-S- SWITCH_ON_IO）现在会发生什么？利用 -c 来确认你的答案是否正确。

```
wolf@wolf-VirtualBox:~/OS-homework/cpu-intro$ ./process-run.py -l 1:0,4:100 -c -S SWITCH_ON_IO
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7*   RUN:io_done          DONE             1
```

使用标记允许进行 I/O 操作时切换。这种情况下系统在进入 I/O 操作时将切换执行别的进程，以最大限度利用 cpu。这种方式与第三题的时间相同，cpu 利用率较高。

第五章

5.1 编写一个调用 fork()的程序。在调用之前,让主进程访问一个变量(例如 x)并将其值设置为某个值(例如 100)。子进程中的变量有什么值?当子进程和父进程都改变 x 的值时,变量会发生什么?

在主进程中访问变量 x 并设为 100，子进程中变量 x 保持不变为 100。

（这里对父进程 sleep 是为了防止命令行提示符混杂在结果中）

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int x=100;
    int rc = fork();
    if (rc < 0)
    {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {
        // child (new process)
        printf("child (pid:%d), x=%d\n", (int)getpid(),x);
    }
    else
    {
        // parent (original process)
        printf("parent of %d (pid:%d), x=%d\n",
               rc, (int)getpid(),x);
        sleep(1);
    }
    return 0;
}
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.1 5.1.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.1
parent of 2593 (pid:2592), x=100
child (pid:2593), x=100
wolf@wolf-VirtualBox:~/OS-work/5$
```

当子进程和父进程都改变 x 的值时，变量互不影响，各自改变。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
        int x=100;
    int rc = fork();
    if (rc < 0)
    {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {
        // child (new process)
        x=50;
        printf("child (pid:%d), x=%d\n", (int)getpid(),x);
    }
    else
    {
        // parent (original process)
        x=150;
        printf("parent of %d (pid:%d), x=%d\n",
               rc, (int)getpid(),x);
                sleep(1);
    }
    return 0;
}
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.1 5.1.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.1
parent of 2150 (pid:2149), x=150
child (pid:2150), x=50
wolf@wolf-VirtualBox:~/OS-work/5$
```

5.2 编写一个打开文件的程序(使用 open 系统调用),然后调用 fork 创建一个新进程。子进程和父进程都可以访问 open()返回的文件描述符吗?当它们并发(即同时)写入文件时,会发生什么?

使用如下 c 代码打开文件并执行操作。

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

int main(){
        int rc=fork();
        int fd=open("./5.2.txt",O_RDWR);
        if (rc<0){
                fprintf(stderr,"fork failed!\n");
                exit(1);
        }
        else if(rc==0){
                printf("Child process,fd:%d\n",fd);
                char s1[]="Child process";
                write(fd,s1,sizeof(s1));
        }
        else{
                printf("Parent process,fd:%d\n",fd);
                char s2[]="Parent process";
                write(fd,s2,sizeof(s2));
                sleep(1);
        }
        close(fd);
        return 0;
}
```

调用 fork()创建新进程并使子进程与父进程访问 open()返回的文件描述符。可见子进程与父进程都可以访问 open()返回的文件描述符。

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.2 5.2.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.2
Parent process,fd:3
Child process,fd:3
wolf@wolf-VirtualBox:~/OS-work/5$
```

将它们并发地写入文件后，发现文件中被较后执行的子进程写入覆盖。

5.4 编写一个调用 fork()的程序,然后调用某种形式的 exec()来运行程序"/bin/ls"看看是否可以尝试 exec 的所有变体,包括 execl()、execle()、execlp()、execv()、execvp()和 execve(),为什么同样的基本调用会有这么多变种？

首先使用 man exec 指令查看 exec 族函数的函数原型

```
EXEC(3)                    Linux Programmer's Manual                    EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                   char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE
```

参数可以单独传入，也可以按数组的方式传入，并可以传入环境变量，路径，可执行文件名。

总结：

（1）带 l 的 exec 函数：execl、execlp、execle，表示后边的参数以可变参数的形式给出且都以一个空指针结束。

（2）带 p 的 exec 函数：execlp、execvp，表示第一个参数 path 不用输入完整路径，只要给出命令名即可，它会在环境变量 PATH 当中查找命令.

（3）不带 l 的 exec 函数：execv、execvp 表示命令所需的参数以 char *arg[] 形式给出且 arg 最后一个元素必须是 NULL 。

（4）带 e 的 exec 函数：execle 表示，将环境变量传递给需要替换的进程。

我们这里以 excl 为例，尝试运行程序 ls。

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

int main(){
        int rc=fork();
        if (rc<0){
                fprintf(stderr,"fork failed!\n");
                exit(1);
        }
        else if(rc==0){
                printf("execl\n");
                execl("/bin/ls","ls",NULL);
        }
        else{
                int wc=wait(NULL);
        }
        return 0;
}
```

下面是结果。

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
5.1  5.1.c  5.1.c~  5.2  5.2.c  5.2.c~  5.2.txt  5.2.txt~  5.4  5.4.c  5.4.c~
wolf@wolf-VirtualBox:~/OS-work/5$ 
```

当然也可以尝试其他的结果，我们这里分次尝试了以下代码的所有共 6 个语句。

```c
int main(){
        int rc=fork();
        if (rc<0){
                fprintf(stderr,"fork failed!\n");
                exit(1);
        }
        else if(rc==0){
                printf("execl\n");

                const char* arg;
                char *const argv[] = {"ls","-l",NULL};
                char * const envp[] = {"", "", NULL};
                //execl
                execl("/bin/ls" , arg , NULL);

                //execlp
                execlp("ls" , arg , NULL);

                //execle
                execle("/bin/ls" , arg , NULL , envp);

                //execv
                execv("/bin/ls", argv);

                //execvp
                execvp("ls", argv);

                //execvpe
                execvpe("ls" , argv , envp);

        }
        else{
                int wc=wait(NULL);
        }
        return 0;
}
```

分别得到结果如下：

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
5.1  5.1.c  5.1.c~  5.2  5.2.c  5.2.c~  5.2.txt  5.2.txt~  5.4  5.4.c  5.4.c~
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
5.1  5.1.c  5.1.c~  5.2  5.2.c  5.2.c~  5.2.txt  5.2.txt~  5.4  5.4.c  5.4.c~
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
5.1  5.1.c  5.1.c~  5.2  5.2.c  5.2.c~  5.2.txt  5.2.txt~  5.4  5.4.c  5.4.c~
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
总用量 52
-rwxrwxr-x 1 wolf wolf 7378  4月 12 01:29 5.1
-rw-rw-r-- 1 wolf wolf  554  4月 12 00:56 5.1.c
-rw-rw-r-- 1 wolf wolf  554  4月 12 00:56 5.1.c~
-rwxrwxr-x 1 wolf wolf 7498  4月 12 01:29 5.2
-rw-rw-r-- 1 wolf wolf  456  4月 12 01:53 5.2.c
-rw-rw-r-- 1 wolf wolf  456  4月 12 01:28 5.2.c~
-rw-rw-r-- 1 wolf wolf   15  4月 12 01:29 5.2.txt
-rw-rw-r-- 1 wolf wolf    0  4月 12 01:29 5.2.txt~
-rwxrwxr-x 1 wolf wolf 7452  4月 12 02:28 5.4
-rw-rw-r-- 1 wolf wolf  620  4月 12 02:28 5.4.c
-rw-rw-r-- 1 wolf wolf  618  4月 12 02:27 5.4.c~
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
总用量 52
-rwxrwxr-x 1 wolf wolf 7378  4月 12 01:29 5.1
-rw-rw-r-- 1 wolf wolf  554  4月 12 00:56 5.1.c
-rw-rw-r-- 1 wolf wolf  554  4月 12 00:56 5.1.c~
-rwxrwxr-x 1 wolf wolf 7498  4月 12 01:29 5.2
-rw-rw-r-- 1 wolf wolf  456  4月 12 01:53 5.2.c
-rw-rw-r-- 1 wolf wolf  456  4月 12 01:28 5.2.c~
-rw-rw-r-- 1 wolf wolf   15  4月 12 01:29 5.2.txt
-rw-rw-r-- 1 wolf wolf    0  4月 12 01:29 5.2.txt~
-rwxrwxr-x 1 wolf wolf 7452  4月 12 02:28 5.4
-rw-rw-r-- 1 wolf wolf  620  4月 12 02:28 5.4.c
-rw-rw-r-- 1 wolf wolf  618  4月 12 02:27 5.4.c~
```

```
wolf@wolf-VirtualBox:~/OS-work/5$ gcc -o 5.4 5.4.c
wolf@wolf-VirtualBox:~/OS-work/5$ ./5.4
execl
total 52
-rwxrwxr-x 1 wolf wolf 7378 Apr 12 01:29 5.1
-rw-rw-r-- 1 wolf wolf  554 Apr 12 00:56 5.1.c
-rw-rw-r-- 1 wolf wolf  554 Apr 12 00:56 5.1.c~
-rwxrwxr-x 1 wolf wolf 7498 Apr 12 01:29 5.2
-rw-rw-r-- 1 wolf wolf  456 Apr 12 01:53 5.2.c
-rw-rw-r-- 1 wolf wolf  456 Apr 12 01:28 5.2.c~
-rw-rw-r-- 1 wolf wolf   15 Apr 12 01:29 5.2.txt
-rw-rw-r-- 1 wolf wolf    0 Apr 12 01:29 5.2.txt~
-rwxrwxr-x 1 wolf wolf 7377 Apr 12 02:28 5.4
-rw-rw-r-- 1 wolf wolf  624 Apr 12 02:28 5.4.c
-rw-rw-r-- 1 wolf wolf  622 Apr 12 02:28 5.4.c~
```

为什么同样的基本调用会有这么多变种？多种 exec()调用的参数传递方式、传递参数不同，方便以不同的形式使用，实现更多功能。

第七章

7.1 使用 SJF 和 FIFO 调度程序运行长度为 200 的 3 个作业时，计算响应时间和周转时间。

SJF 与 FIFO 均采取如下方式运行三个进程

SJF/FIFO		
200	200	200
A	B	C

可总结出如下表格

SJF/FIFO	响应时间	周转时间
A	0	200
B	200	400
C	400	600
average	200	400

可见，对于 SJF 与 FIFO，其平均响应时间为 200，平均周转时间为 400。

7.2 现在做同样的事情，但有不同长度的作业，即 100、200 和 300。
SJF 与 FIFO 均采取如下方式运行三个进程

SJF/FIFO		
100	200	300
A	B	C

可总结出如下表格

SJF/FIFO	响应时间	周转时间
A	0	100
B	100	300
C	300	600
average	133.3	333.3

可见，对于 SJF 与 FIFO，其平均响应时间为 133.3，平均周转时间为 333.3。

7.3 现在做同样的事情，但采用 RR 调度程序，时间片为 1。
采用 RR 调度，时间片为 1。

RR	响应时间	周转时间
A	0	598
B	1	599
C	2	600
average	1	599

可见，对于 RR，其平均响应时间为 1，平均周转时间为 599。

7.4 对于什么类型的工作负载，SJF 提供与 FIFO 相同的周转时间？
答：
（1）作业到达时间不一致时。
（2）作业到达时间一致时，列表中执行顺序按作业长度非严格递增。
（3）部分作业到达时间一致时，到达时间一致的部分满足条件（2）

7.5 对于什么类型的工作负载和量子长度，SJF 与 RR 提供相同的响应时间？
答：每个工作的时长相同且等于量子长度（时间切片）。

7.6 随着工作长度的增加，SJF 的响应时间会怎样？你能使用模拟程序来展示趋势吗？
答：

（1）若所有工作长度都增加，则除了最短任务的响应时间不变，其余任务的响应时间均增加。
（2）若只有部分工作长度增加，则长度比该部分工作长的工作，其响应时间增加。

模拟程序略。

7.7 随着量子长度的增加,RR 的响应时间会怎样?你能写出一个方程,计算给定 N 个工作时,最坏情况的响应时间吗?
答：

假设量子长度为 t0，则平均响应时间为
[0*t0+1*t0+2*t0+……+(n-1)*t0]/n=(n-1)*t0/2
最坏情况的响应时间即是最后一个工作的响应时间，为(n-1)*t0

第八章
8.1 只用两个工作和两个队列运行几个随机生成的问题。针对每个工作计算 MLFQ 的执行记录。限制每项作业的长度并关闭 I/O，让你的生活更轻松。

根据题目意思，查阅 Readme 文件，使用以下指令
./mlfq.py -n 2 -j 2 -m 100 -M 0
其中-n 为队列个数，-j 为工作个数，-m 为每项作业长度上限，-M 为 I/O 频率，由于关闭 I/O，其频率为 0。
（1）尝试第一个问题

```
wolf@wolf-VirtualBox:~/OS-work/8$ ./mlfq.py -n 2 -j 2 -m 100 -M 0
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  84 - ioFreq   0
  Job  1: startTime   0 - runTime  42 - ioFreq   0

Compute the execution trace for the given workloads.
If you would like, also compute the response and turnaround
times for each of the jobs.

Use the -c flag to get the exact results when you are finished.
```

关于工作的参数记录如下：
工作 0，时间 84

工作 1，时间 42，

队列 0，队列 1：时间切片 10

重新加入最高优先级队列的时间 S（巫毒常量）：0

（由于这里只有两个工作且无 I/O，重新加入最高优先级队列的意义不大）

计算执行记录如下，

时间	队列 0	队列 1	CPU
0-10	AB		A
10-20	B	A	B
20-30		AB	A
30-40		AB	B
40-50		AB	A
50-60		AB	B
60-70		AB	A
70-80		AB	B
80-90		AB	A
90-92		AB	B
92-126		AB	A

可见平均响应时间为 5，平均周转时间为 109

使用-c 可以查看每个单位时间的运算情况，与上表一致。

使用-s 指令可以更换种子，我这里用了-s 9，得到如下

（2）尝试第二个问题

```
wolf@wolf-VirtualBox:~/OS-work/8$ ./mlfq.py -s 9 -n 2 -j 2 -m 100 -M 0
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  46 - ioFreq   0
  Job  1: startTime   0 - runTime  14 - ioFreq   0

Compute the execution trace for the given workloads.
If you would like, also compute the response and turnaround
times for each of the jobs.

Use the -c flag to get the exact results when you are finished.
```

分析同第一个问题，略去不讲，只说计算执行记录如下表

时间	队列 0	队列 1	CPU
0-10	AB		A
10-20	B	A	B
20-30		AB	A
30-44		AB	B
40-66		AB	A

可见平均响应时间为 5，平均周转时间为 55

使用-c 可以查看每个单位时间的运算情况，与上表一致。

（3）尝试其他问题，可以通过指令运行，这里就不再赘述。

8.3 将如何配置调度程序参数，像轮转调度程序那样工作？

答：当工作在同一队列时，进行轮转调度工作，因此只需要将 mlfq 调度的队列数设置为 1，这里考虑重新加入最高优先级队列的时间 S（巫毒常量）意义不大，所以不需考虑。

综上答案为./mlfq.py -n 1

8.5 给定一个系统，其最高队列中的时间片长度为 10ms，你需要如何频繁地将工作推回到最高优先级级别（带有-B 标志），以保证一个长时间运行（并可能饥饿）的工作得到至少 5%的 CPU。

答：10ms/5%=200ms，故 boost 的频率至少为 200ms 才能使该工作至少得到 5%的 CPU。

第九章

9.1 计算 3 个工作在随机种子为 1、2 和 3 时的模拟解。

（1）随机种子 seed=1

```
wolf@wolf-VirtualBox:~/OS-work/9$ ./lottery.py -s 1
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

Here is the job list, with the run time of each job:
  Job 0 ( length = 1, tickets = 84 )
  Job 1 ( length = 7, tickets = 25 )
  Job 2 ( length = 4, tickets = 44 )


Here is the set of random numbers you will need (at most):
Random 651593
Random 788724
Random 93859
Random 28347
Random 835765
Random 432767
Random 762280
Random 2106
Random 445387
Random 721540
Random 228762
Random 945271
```

计算模拟解过程如下：

随机种子：1　份额：84,25,44　工作长度：1,7,4

时间	job0	job1	job2	随机数	总份额	模	运行
1	0	0	1	651593	153	119	job2
2	1(完成)	0	1	788724	153	9	job0
3		1	1	93859	69	19	job1
4		1	2	28347	69	57	job2
5		1	3	835765	69	37	job2
6		1	4(完成)	432767	69	68	job2
7		2		762280	25	5	job1
8		3		2106	25	6	job1
9		4		445387	25	12	job1
10		5		721540	25	15	job1
11		6		228762	25	12	job1
12		7(完成)		945271	25	21	job1

（2）随机种子 seed=2

```
wolf@wolf-VirtualBox:~/OS-work/9$ ./lottery.py -s 2
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 2

Here is the job list, with the run time of each job:
  Job 0 ( length = 9, tickets = 94 )
  Job 1 ( length = 8, tickets = 73 )
  Job 2 ( length = 6, tickets = 30 )


Here is the set of random numbers you will need (at most):
Random 605944
Random 606802
Random 581204
Random 158383
Random 430670
Random 393532
Random 723012
Random 994820
Random 949396
Random 544177
Random 444854
Random 268241
Random 35924
Random 27444
Random 464894
Random 318465
Random 380015
Random 891790
Random 525753
Random 560510
Random 236123
Random 23858
Random 325143
```

计算模拟解过程如下：

随机种子：2　份额：94,73,30　工作长度：9,8,6

时间	job0	job1	job2	随机数	总份额	模	运行
1	0	0	1	605944	197	169	job2
2	1	0	1	606802	197	42	job0
3	2	0	1	581204	197	54	job0
4	2	0	2	158383	197	192	job2
5	3	0	2	430670	197	28	job0
6	3	1	2	393532	197	123	job1
7	4	1	2	723012	197	22	job0
8	4	1	3	994820	197	167	job2
9	5	1	3	949396	197	53	job0
10	6	1	3	544177	197	63	job0
11	7	1	3	444854	197	28	job0
12	7	2	3	268241	197	124	job1
13	8	2	3	35924	197	70	job0
14	9(完成)	2	3	27444	197	61	job0
15		3	3	464894	103	55	job1
16		3	4	318465	103	92	job2
17		4	4	380015	103	48	job1
18		5	4	891790	103	16	job1
19		6	4	525753	103	41	job1
20		6	5	560510	103	87	job2
21		7	5	236123	103	47	job1
22		8(完成)	5	23858	103	65	job1
23			6(完成)	325143	30	3	job2

（3）随机种子 seed=3

```
wolf@wolf-VirtualBox:~/OS-work/9$ ./lottery.py -s 3
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 3

Here is the job list, with the run time of each job:
  Job 0 ( length = 2, tickets = 54 )
  Job 1 ( length = 3, tickets = 60 )
  Job 2 ( length = 6, tickets = 6 )


Here is the set of random numbers you will need (at most):
Random 13168
Random 837469
Random 259354
Random 234331
Random 995645
Random 470263
Random 836462
Random 476353
Random 639068
Random 150616
Random 634861
```

计算模拟解过程如下：

随机种子：3　份额：54,60,6　工作长度：2,3,6

时间	job0	job1	job2	随机数	总份额	模	运行
1	0	1	0	13168	120	88	job1
2	0	2	0	837469	120	109	job1
3	1	2	0	259354	120	34	job0
4	1	3(完成)	0	234331	120	91	job1
5	2(完成)		0	995645	60	5	job0
6			1	470263	6	1	job1
7			2	836462	6	2	job1
8			3	476353	6	1	job2
9			4	639068	6	2	job2
10			5	150616	6	4	job2
11			6(完成)	634861	6	1	job2

9.2 现在运行两个具体的工作：每个长度为 10，但是一个（工作 0）只有一张彩票，另一个（工作 1）有 100 张（-l 10:1,10:100）。
彩票数量如此不平衡时会发生什么？在工作 1 完成之前，工作 0 是否会运行？多久？一般来说，这种彩票不平衡对彩票调度的行为有什么影响？

使用 ./lottery.py -l 10:1,10:100 进行模拟。

```
wolf@wolf-VirtualBox:~/OS-work/9$ ./lottery.py -l 10:1,10:100
ARG jlist 10:1,10:100
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 0

Here is the job list, with the run time of each job:
  Job 0 ( length = 10, tickets = 1 )
  Job 1 ( length = 10, tickets = 100 )


Here is the set of random numbers you will need (at most):
Random 844422
Random 757955
Random 420572
Random 258917
Random 511275
Random 404934
Random 783799
Random 303313
Random 476597
Random 583382
Random 908113
Random 504687
Random 281838
Random 755804
Random 618369
Random 250506
Random 909747
Random 982786
Random 810218
Random 902166
```

计算模拟解过程如下：

份额：1,100　工作长度：10

时间	job0	job1	随机数	总份额	模	运行
1	0	1	844422	101	62	job1
2	0	2	757955	101	51	job1
3	0	3	420572	101	8	job1
4	0	4	258917	101	54	job1
5	0	5	511275	101	13	job1
6	0	6	404934	101	25	job1
7	0	7	783799	101	39	job1
8	0	8	303313	101	10	job1
9	0	9	476597	101	79	job1
10	0	10(完成)	583382	101	6	job1
11	1		908113	1	0	job0
12	2		504687	1	0	job0
13	3		281838	1	0	job0
14	4		755804	1	0	job0
15	5		618369	1	0	job0
16	6		250506	1	0	job0
17	7		909747	1	0	job0
18	8		982786	1	0	job0
19	9		810218	1	0	job0
20	10(完成)		902166	1	0	job0

这里工作 0 与工作 1 的彩票数量差距过大了，导致在工作 1 完成之前工作 0 占用 CPU 几乎是不可能的。从模拟结果上看，在工作 1 完成之前，工作 0 没有运行。

在这种情况下，持有份额小的工作响应时间与周转时间非常长，且基本上占用不了 CPU。极有可能会“饿死”。

9.3 如果运行两个长度为 100 的工作，都有 100 张彩票（-l 100:100,100:100），调度程序有多不公平？运行一些不同的随机种子来确定（概率上的）答案。不公平性取决于一项工作比另一项工作早完成多少。

使用./lottery.py -l 100:100,100:100 来进行模拟。使用-s 来设定种子。

种子编号	工作 0 完成时间	工作 1 完成时间	提前完成时间
0	192	200	8
1	200	196	4
2	200	190	10
3	196	200	4
4	200	199	1
5	200	181	19
6	200	193	7
7	200	185	15
8	200	191	9
9	200	192	8
10	197	200	3
11	196	200	4
12	200	189	11
Average	198.5	193.5	7.9

提前完成的工作所用时间平均比后完成的快了 8 左右，对比工作长度只能保证基本接近公平，假设以提前完成时间比工作长度为指标，不公平度约为 7.9%。