**Q1**

***“***

Run a few randomly-generated problems with just two jobs andtwo queues; compute the MLFO execution trace for each. Makeyour life easier by limiting the length of each job and turning off/os.

可以用如下指令：

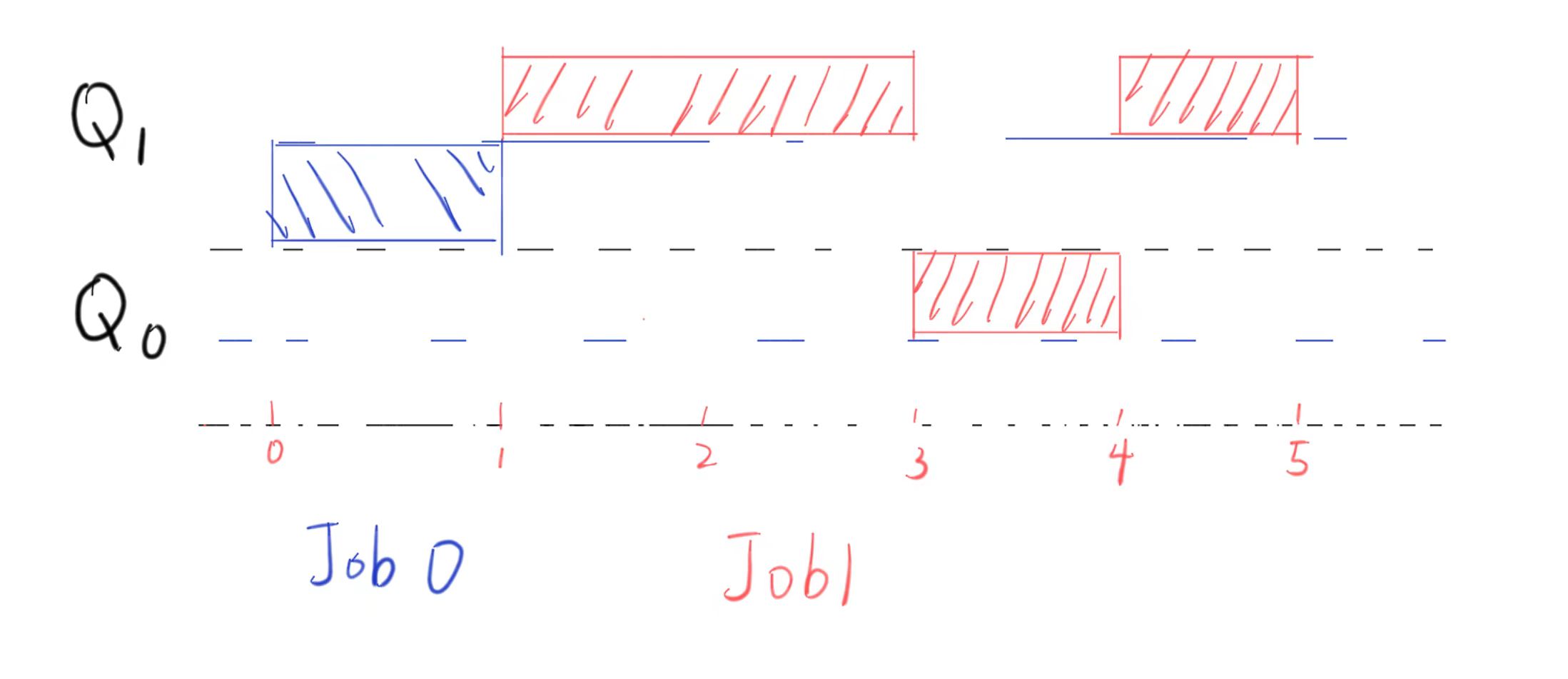
***“***

*python3 mlfq.py -j 2 -n 2 -q 2 -m 10 -M 0 -s 123 -B 4 -c*

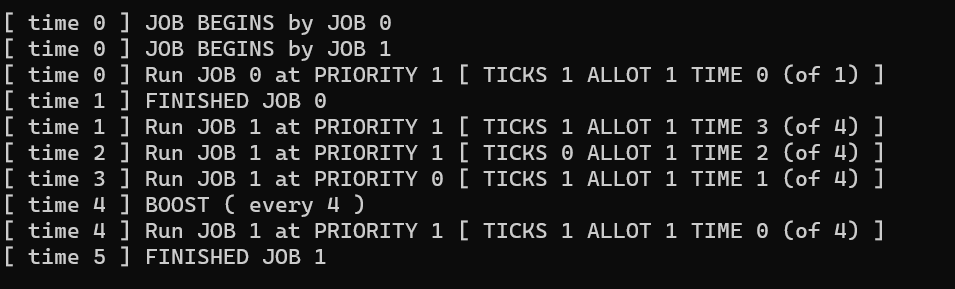
一些参数的说明如下：

* -j 2：生成 2 个作业。
* -n 2：设置 2 个队列。
* -q 2：每个队列的时间片长度设置为 2 毫秒。
* -m 10：每个作业的最大运行时间为 10 毫秒。
* -M 0：每个作业的最大 I/O 频率为 0 毫秒。（没有I/O）
* -s 123：设置随机种子为 123。
* -B 4：每隔 4 毫秒，系统会自动将所有作业的优先级提升回最高优先级队列。
* -c：计算并打印出作业调度的执行轨迹和统计信息。

每个任务开始执行（生成/加入队列）的时间不可以随机，唯一能够指定生成时间的-l没有随机生成的功能。



**图一 手绘任务轨迹**



**图二 程序运行实例**

**Q2**

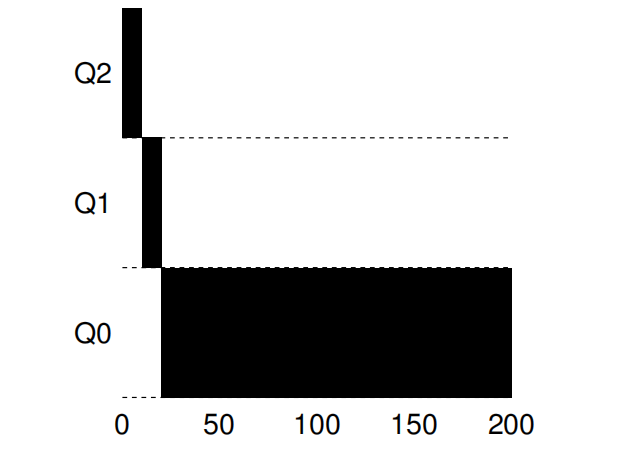
***“***

How would you run the scheduler to reproduce each of the exam-ples in the chapter?

* 无boost 单任务 三队列 时间片10ms 无4a4b规则 任务1从0时刻开始，运行200ms，无io操作

***“***

*python3 mlfq.py -n 3 -q 10 -j 1 -l 0,200,0 -B 0 -S -i 5 -c*



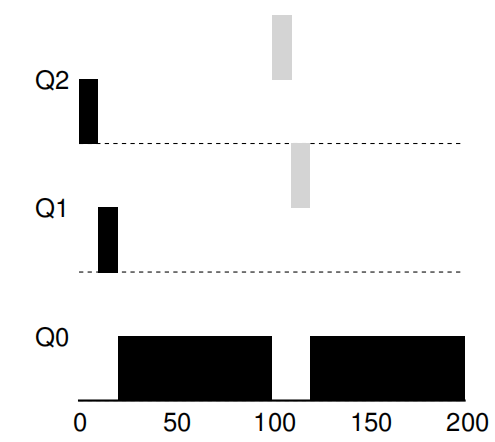
**图三 教材给图**

这个程序太简单了，就不截图了。

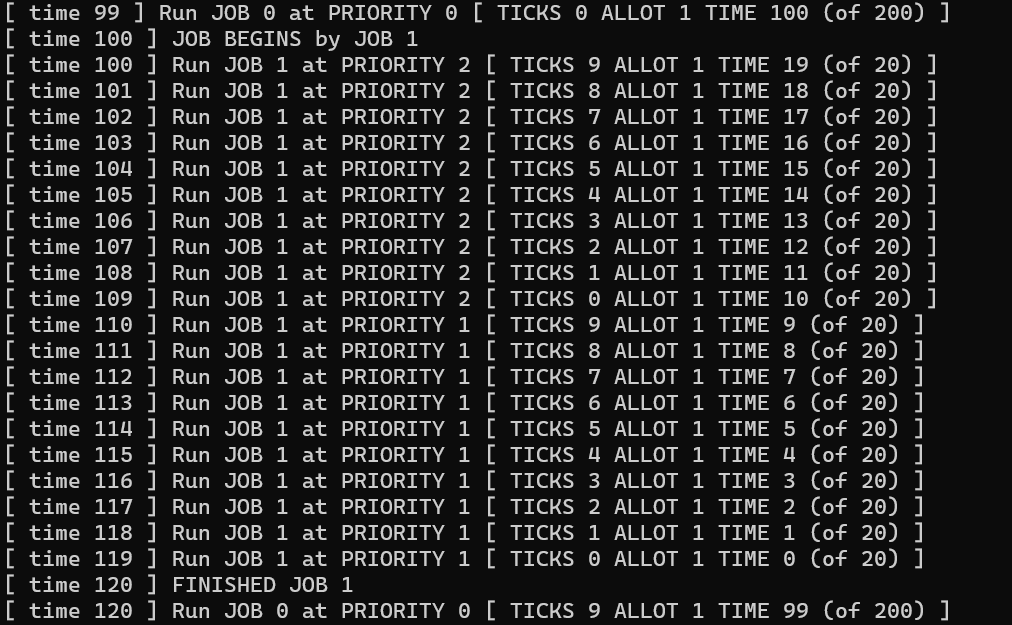
* 无boost 双任务 三队列 时间片10ms 无4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从100时刻开始，运行20ms，无io操作

***“***

*python3 mlfq.py -n 3 -q 10 -j 2 -l 0,200,0:100,20,0 -B 0 -S -i 5 -c*



**图四 教材给图**

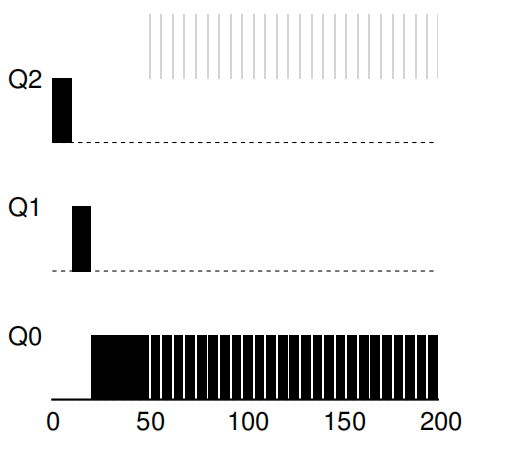


**图五 程序运行实例**

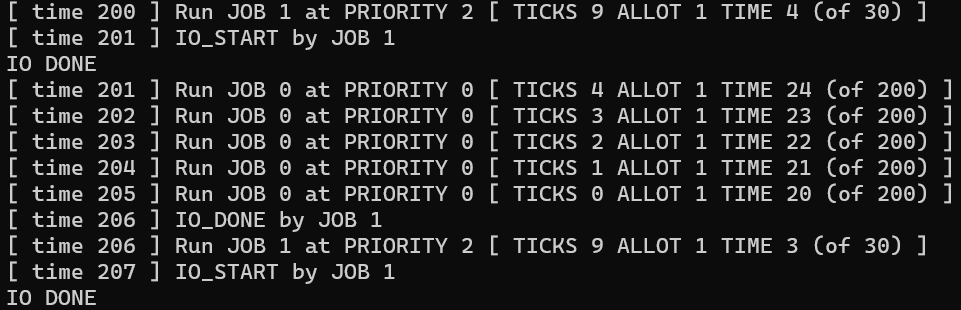
* 无boost 双任务 三队列 时间片10ms 有4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从50时刻开始，运行30ms，io频率为1ms

***“***

*python3 mlfq.py -n 3 -q 10 -j 2 -l 0,200,0:50,30,1 -B 0 -S -i 5 -c*



**图六 教材给图**

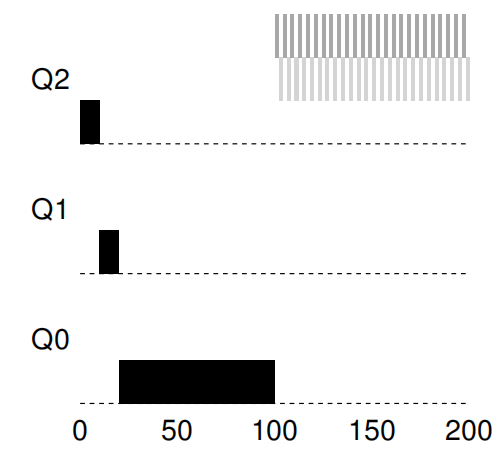


**图七 程序运行实例**

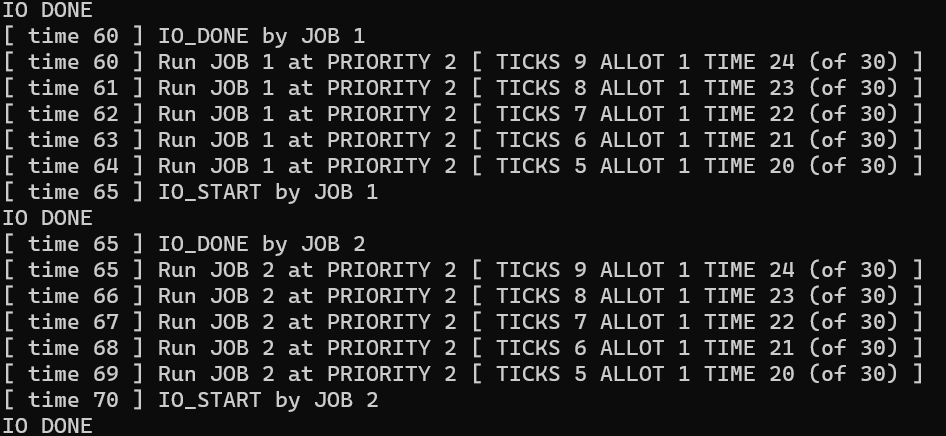
* 无boost 三任务 三队列 时间片10ms 有4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从50时刻开始，运行30ms，io频率为5ms，任务三从51时刻开始，运行30ms，io频率为5ms ， 系统io速度（-i）为5ms（这个是默认的）

***“***

*python3 mlfq.py -n 3 -q 10 -j 3 -l 0,200,0:50,30,1:51,30,1 -B 0 -S -c*



**图八 教材给图**

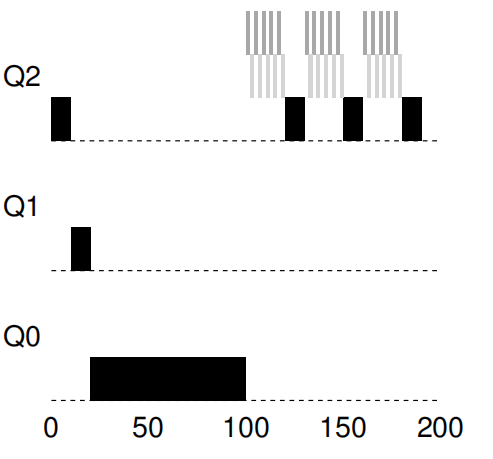


**图九 程序运行实例**

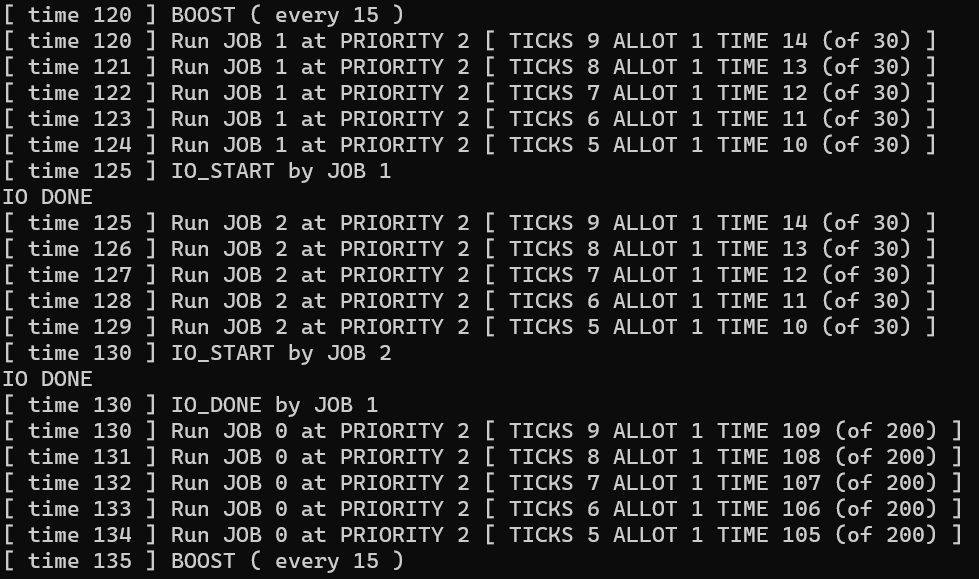
* 有boost，15ms一次 三任务 三队列 时间片10ms 有4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从50时刻开始，运行30ms，io频率为5ms，任务三从51时刻开始，运行30ms，io频率为5ms， 系统io速度（-i）为5ms

***“***

*python3 mlfq.py -n 3 -q 10 -j 3 -l 0,200,0:50,30,5:51,30,5 -B 15 -S -c*



**10 教材给图**

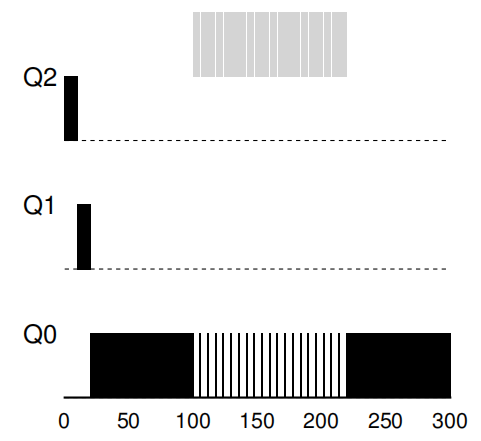


**11 程序运行实例**

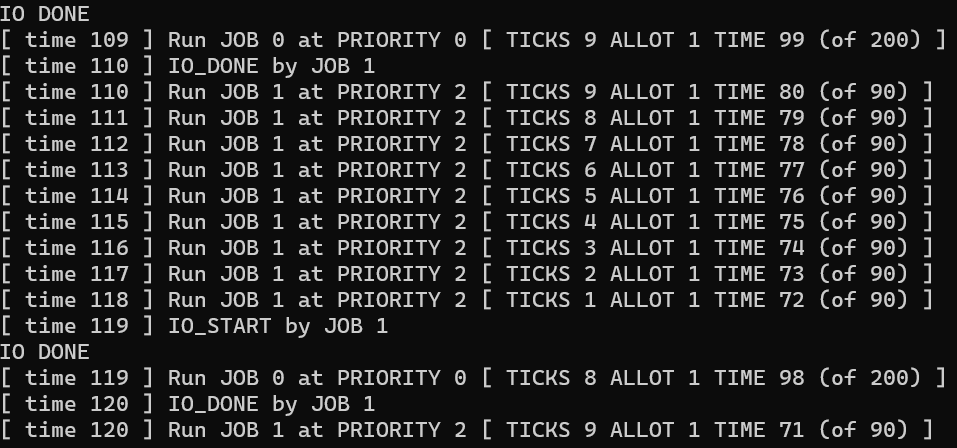
* 无boost 双任务 三队列 时间片10ms 有4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从100时刻开始，运行90ms，io频率为9ms， 系统io速度（-i）为1ms

***“***

*python3 mlfq.py -n 3 -q 10 -i 1 -l "0,200,0:100,90,9" -S -c*



**12 教材给图**

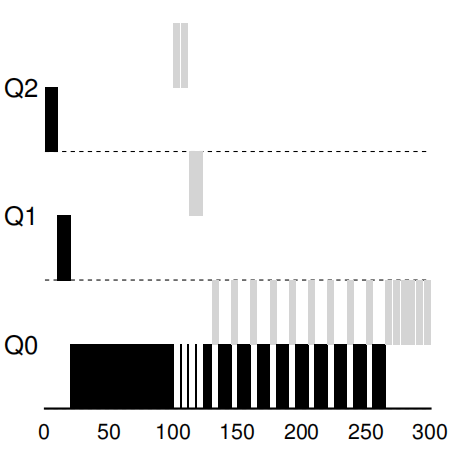


**13 程序运行实例**

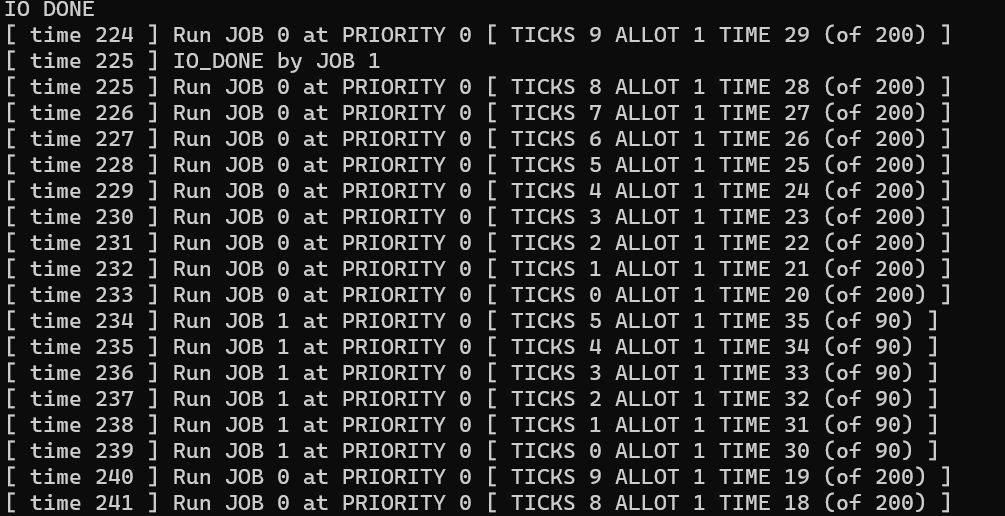
* 无boost 双任务 三队列 时间片10ms 无4a4b规则 任务1从0时刻开始，运行200ms，无io操作，任务2从100时刻开始，运行90ms，io频率为9ms， 系统io速度（-i）为1ms

***“***

*python3 mlfq.py -n 3 -q 10 -i 1 -l "0,200,0:100,90,9" -c*



**14 教材给图**



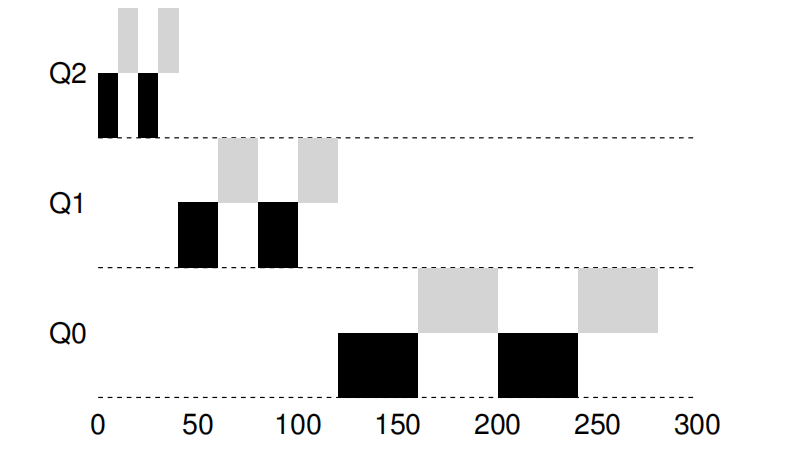
**15 程序运行实例**

* 无boost 双任务 三队列 队列的时间片分别为10ms，20ms，40ms，任务1运行70ms，任务2运行70ms，都从0时刻开始。

这个也很简单，就不运行了。

***“***

*python3 mlfq.py -n 3 -q 10 -q2 20 -q3 40 -i 1 -l "0,70,0:0,70,0" -c*



**16 教材给图**

**Q3**

***“***

How would you configure the scheduler parameters to behave justlike a round-robin scheduler?

单一队列，把-q设置为1。

**Q4**

***“***

Craft a workload with two jobs and scheduler parameters so that4.one job takes advantage of the older Rules 4a and 4b (turned onwith the -s flag) to game the scheduler and obtain 99% of the CPUover a particular time interyal.

设计一下的话，

三队列双任务 无boost io速度1 时间片100ms，使用4a4b规则

任务1 0时刻开始 有 io 运行500ms io间隔99ms

任务2 0时刻开始 无io 运行500ms

需要等到任务2运行到一次时间片降低优先级，才可以到达99％cpu资源

具体的说，是：

199到任务1结束，99％cpu资源，下面是指令：

***“***

*python3 mlfq.py -n 3 -q 100 -i 1 -S -l 0,500,99:0,500,0 -c*

**Q5**

***“***

Given a system with a quantum length of 10 ms in its highest queuehow often would you have to boost jobs back to the highest prioritylevel (with the -B flag)in order to guarantee that a single longrunning (and potentially-starving) job gets at least 5% of the CPu?

由于如果说所有任务轮转一次的时间超过boost，那么boost就没有任何意义，所以有如下假设：

*假设：所有任务轮转一次的时间不会超过boost*

那么在一次boost内，这个任务至少会运行一次。

又因为：boost越大他能获得的CPU资源就越少，取最坏情况，使用5%时

5%=10ms/boost

解出来就是200ms。

**Q6**

***“***

One question that arises in scheduling is which end of a queue toadd a job that just finished I/O; the -I flag changes this behaviorfor this scheduling simulator. Play around with some workloadsand see if you can see the effect of this flag.

下面是我设计的两个任务：

假设说两个任务，两个队列。一个要长时间使用cpu，一个是io密集形任务

第一个任务运行50ms，无io；

第二个任务运行50ms，io间隔1ms；

io等待时间2ms，时间片5ms

***“***

*python3 mlfq.py -j 2 -q 5 -n 2 -I -i 2 -l "0,50,0:0,50,1" -c*

有-I标志：

***“***

IO DONE

[ time 48 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 16 (of 50) ]

[ time 49 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 15 (of 50) ]

[ time 50 ] IO\_DONE by JOB 1

[ time 50 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 34 (of 50) ]

[ time 51 ] IO\_START by JOB 1

IO DONE

以上是一个调度JOB0和JOB1时的循环，可以看到，在JOB1等待IO的2ms内，JOB0会占用CPU，在IO等待完毕之后，JOB会继续运行1ms再等待IO

无-I标志：

***“***

IO DONE

[ time 32 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 24 (of 50) ]

[ time 33 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 23 (of 50) ]

[ time 34 ] IO\_DONE by JOB 1

[ time 34 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 22 (of 50) ]

[ time 35 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 21 (of 50) ]

[ time 36 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 20 (of 50) ]

[ time 37 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 42 (of 50) ]

[ time 38 ] IO\_START by JOB 1

IO DONE

以上是一个调度JOB0和JOB1时的循环，可以看到，JOB1是在时间片的末尾才能够运行一次，并且一个时间片按理来说只能运行五次，这里却因为JOB1在TICKS 2时收到了IO\_DONE，延长了时间片运行了JOB1。

之后修改过数据运行一次，如果说io间隔2ms，他会把时间片延长2ms。

总结：  
 有-I，在有IO的进程请求IO并等待时会立刻执行其他程序，等待完毕后会立刻切换回来

无-I，在有IO的进程等待完毕后不会马上切换回来，而是等时间片轮转完毕后，再把时间交给有IO的进程，直到这个进程下一次请求IO或者是时间片用光，再开始下一个时间片。