深 圳 大 学 实 验 报 告

课 程 名 称：操作系统

实验项目名称：实验 2 处理机调度

学　　院：计算机与软件学院

专　　业：计算机科学与技术

指 导 教 师：阮元

报告人：刘睿辰　学号：2018152051　班级：数计班

实 验 时 间：2021.5.10-2021.5.22

实验报告提交时间：2021.5.22

教务处制

一、 实验目的

1. 加深对进程调度的直观认识；
2. 掌握 Linux 操作系统中调度信息的查看方法；
3. 掌握 Linux 中 CFS 和 RT 调度的 API；
4. 掌握作业调度和进程调度的实现原理；
5. 掌握常用的几种调度算法；
6. 掌握死锁问题产生的必要条件以及预防和避免措施。

二、 实验内容

1. 使用 Linux 或其它 Unix 类操作系统；
2. 学习该操作系统提供的进程、线程创建的函数使用方法；
3. 常用的几种实时调度算法，高优先权优先调度和基于时间片的轮转调度算法；
4. 多处理机环境下的进程调度方式；
5. 死锁的基本概念，预防死锁的方法。

三、 实验环境

1. 操作系统：Windows 10 操作系统、Linux 操作系统；
2. 实验平台：Ubuntu 18.04，Visual Studio Code。

四、 实验步骤及说明

1. 操作部分（第二小题和第三小题调整了顺序）

1） 在一个空闲的单核 Linux 系统上用 nice 命令调整两个进程的优先级（10 分）

要求：使得它们各自使用约 1/5 和 4/5 的 CPU 资源。用 top 和/proc/PID/sched 展示各进程使用的调度策略以及调整前后的优先级变化，用 top 命令展示 CPU 资源分配的效果。

调整前：我们首先看一下调整之前的 CPU 资源分配。我们首先准备一个死循环程序，代码很简单如图 1 所示。

```
#include<stdio.h>
int main()
{
    int k;
    while(1)
    {
        k++;
    }
    return 0;
}
```

图 1. 样例运行程序

然后我们开启两个终端来分别运行该程序。正常来说没有设置优先级，二者的 CPU 分配应该是近似的或者是相等的。我们开启 top 来观察进程的 CPU 资源分配情况，如图 2 所示，这两个 Loop 进程的 CPU 资源分配是相等的。再进行观察，发现尽管 CPU 资源分配是动态的，但是实际上二者的差距还是很小的。

```
进程 USER      PR  NI    VIRT    RES    SHR ◆ %CPU %MEM     TIME+ COMMAND
  2010 ruichen   20   0    4376    856    792 R 28.9  0.0   0:09.90 Loop
  2025 ruichen   20   0    4376    716    652 R 28.9  0.0   0:05.69 Loop
   661 root      20   0  659308  43492  15520 S 11.8  2.2   0:19.22 snapd
  2039 ruichen   20   0  568232  27340  21488 R  7.6  1.4   0:00.32 gnome-software
  2047 ruichen   20   0  323896  18588  14484 R  7.2  0.9   0:00.22 livepatch-notif
  1644 ruichen   20   0 2977920 197936  93484 S  6.9  9.8   0:14.99 gnome-shell
  1505 ruichen   20   0  434204  62216  38400 S  3.6  3.1   0:03.46 Xorg
  1994 ruichen   20   0  839808  47988  34708 S  2.3  2.4   0:01.21 gnome-terminal-
  2037 ruichen   20   0  587832  22224  17552 S  1.3  1.1   0:00.20 update-notifier
```

图 2. 未设置 CPU 资源分配的情况

然后再用/proc/PID/sched 观察两进程的差异，如图 3 所示。

```
Loop (2631, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       2880439.047575
se.vruntime                                  :         71753.760167
se.sum_exec_runtime                          :         19886.197409
se.nr_migrations                             :                    0
nr_switches                                  :                 5451
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                 5451
se.load.weight                               :              1048576
se.runnable_weight                           :              1048576
se.avg.load_sum                              :                47468
se.avg.runnable_load_sum                     :                47468
se.avg.util_sum                              :             17761307
se.avg.load_avg                              :                 1023
se.avg.runnable_load_avg                     :                 1023
se.avg.util_avg                              :                  373
se.avg.last_update_time                      :       2880439047168
policy                                       :                    0
prio                                         :                  120
clock-delta                                  :                   23
mm->numa_scan_seq                            :                    0
numa_pages_migrated                          :                    0
numa_preferred_nid                           :                   -1
total_numa_faults                            :                    0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
```

图 3(a). 未设置 CPU 资源分配—进程 2010

```
Loop (2632, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       2878551.565623
se.vruntime                                  :         70975.842626
se.sum_exec_runtime                          :         19108.148841
se.nr_migrations                             :                    0
nr_switches                                  :                 5255
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                 5255
se.load.weight                               :              1048576
se.runnable_weight                           :              1048576
se.avg.load_sum                              :                47427
se.avg.runnable_load_sum                     :                47427
se.avg.util_sum                              :             15222452
se.avg.load_avg                              :                 1023
se.avg.runnable_load_avg                     :                 1023
se.avg.util_avg                              :                  320
se.avg.last_update_time                      :       2878551565312
policy                                       :                    0
prio                                         :                  120
clock-delta                                  :                   19
mm->numa_scan_seq                            :                    0
numa_pages_migrated                          :                    0
numa_preferred_nid                           :                   -1
total_numa_faults                            :                    0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
```

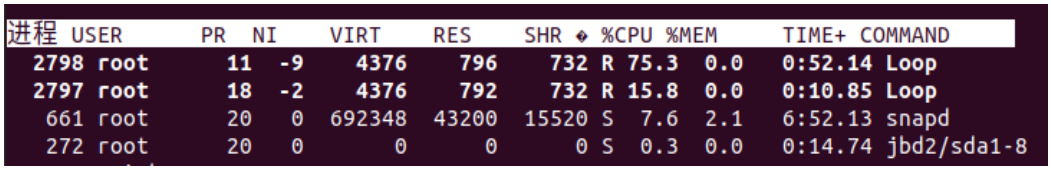
图 3(b). 未设置 CPU 资源分配—进程 2025

对比二者可以发现，两个的启动时间 $se.exec\_start$、虚拟时间推进 $se.vruntime$ 以及 CPU 上获得的$se.sum_exec_runtime$所差不多。由于这两个进程从来没有主动让出 CPU，所以它们的自愿切换次数 $nr\_voluntary\_switches$ 都为 0，进程切换次数 $nr\_switches$ 等于被强制切换的次数 $nr\_involuntary\_switches$。二者的$se.load.weight$也是相等的，为 1048576.

现在我们来设置两个进程的优先级，将前一个进程的优先级置为 0，后一个进程优先级置为 0，然后通过管道

$$nice - n\ 6\ ./Loop|nice - n\ 0\ ./Loop$$

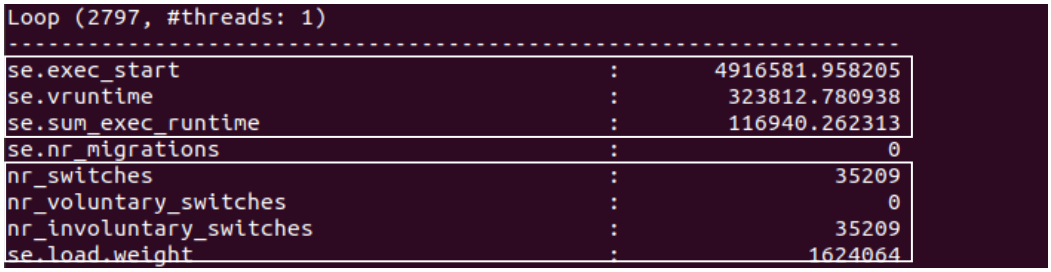
来之从两个程序。由于 NICE 值越大，优先级就越低，所以进程号小一些的进程优先级高一些，进程号大的进程优先级低一些。运行程序之后，从 top 窗口查看 CPU 占用率，如图 4 所示，进程 2797 占用 15.8%的 CPU，约合 1/5；进程 2798 占用 75.3%的 CPU，约合 4/5.



进程 USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2798 root	11	-9	4376	796	732	R	75.3	0.0	0:52.14	Loop
2797 root	18	-2	4376	792	732	R	15.8	0.0	0:10.85	Loop
661 root	20	0	692348	43200	15520	S	7.6	2.1	6:52.13	snapd
272 root	20	0	0	0	0	S	0.3	0.0	0:14.74	jbd2/sda1-8

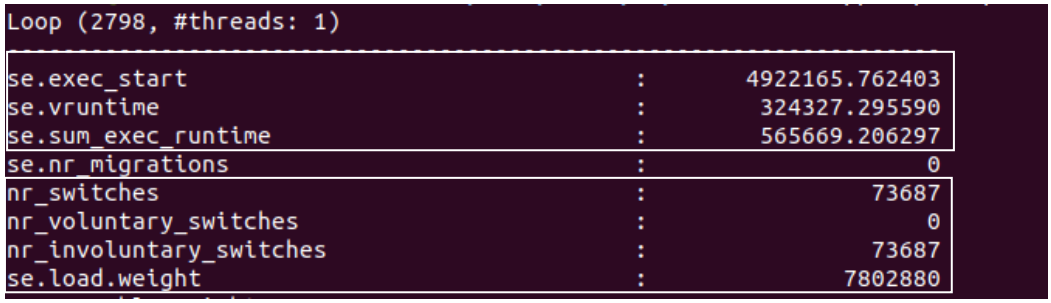
图 4. 设置 CPU 资源分配结果(1)

再来看两个进程的/proc/PID/sched，如图 5 所示。



```
Loop (2797, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :        4916581.958205
se.vruntime                                  :         323812.780938
se.sum_exec_runtime                          :         116940.262313
se.nr_migrations                             :                     0
nr_switches                                  :                 35209
nr_voluntary_switches                        :                     0
nr_involuntary_switches                      :                 35209
se.load.weight                               :               1624064
```

图 5(a). 设置 CPU 资源分配结果(2)



```
Loop (2798, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :        4922165.762403
se.vruntime                                  :         324327.295590
se.sum_exec_runtime                          :         565669.206297
se.nr_migrations                             :                     0
nr_switches                                  :                 73687
nr_voluntary_switches                        :                     0
nr_involuntary_switches                      :                 73687
se.load.weight                               :               7802880
```

图 5(b). 设置 CPU 资源分配结果(3)

可以看到$se.sum_exec_runtime$已经有了明显的差距。进程 2798 优先级更高，因此$se.sum_exec_runtime$也更高（为 565669，远大于进程 2797 的 116940），$se.load.weight$也更高（为 7802880，远大于进程 2797 的 1624064）。由于这两个进程从来没有主动让出 CPU，所以它们的自愿切换次数 $nr\_voluntary\_switches$ 都为 0，进程切换次数 $nr\_switches$ 等于被强制切换的次数 $nr\_involuntary\_switches$。

2） 在一个空闲的单核 Linux 系统运行两个进程，以相同优先级的 RR 实时调度的进程，在上述两个进程结束前运行另一个优先级更高的 FIFO 进程。（15 分）

要求：用 top（某些现象有可能无法观测到，请如实记录）和/proc/PID/sched 展示并记录各进程的调度策略和优先级，展示并记录 FIFO 进程抢占 CPU 的现象，展示并记录两个 RR 进程轮流执行的过程。

在实际实验环境中，按照阅读文档部分设置优先级为 90 和 95 很难查看到结果，所以这里面我们设置成 85 和 86，如图 6 所示。

```
1  ./RR-FIFO-sched 2 85&
2  ./RR-FIFO-sched 2 85&
3  sleep 5s
4  ./RR-FIFO-sched 1 86&
```

图 6. RR-FIFO.sh 脚本

然后参考样例程序 RR－FIFO－sched.c，编译该程序，然后用 chmod a＋x RR－FIFO.sh 将脚本修改为可执行。然后运行该脚本，同时在另一个终端使用 top 查看当前系统进程状态，如图 7 所示。

进程	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2771	root	-86	0	4508	736	672	R	47.8	0.0	0:03.63	RR-FIFO-sched
2770	root	-86	0	4508	816	752	R	47.2	0.0	0:01.89	RR-FIFO-sched
2723	ruichen	20	0	46004	4024	3332	R	0.7	0.2	0:00.36	top
1504	ruichen	20	0	442612	72936	39080	S	0.3	3.6	0:11.38	Xorg

图 7(a). 执行阶段 1

进程	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2773	root	-87	0	4508	704	644	R	94.7	0.0	0:04.74	RR-FIFO-sched
2723	ruichen	20	0	46004	4024	3332	R	0.3	0.2	0:00.37	top
1	root	20	0	160180	9356	6592	S	0.0	0.5	0:02.62	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd

图 7(b). 执行阶段 2

进程	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2770	root	-86	0	4508	816	752	R	36.6	0.0	0:04.05	RR-FIFO-sched
2771	root	-86	0	4508	736	672	R	35.4	0.0	0:05.79	RR-FIFO-sched
1504	ruichen	20	0	442612	72936	39080	S	0.3	3.6	0:11.40	Xorg
1978	ruichen	20	0	845812	58776	43580	S	0.3	2.9	0:03.52	nautilus

图 7(c). 执行阶段 3

在图 7(a)中是第一阶段，进程 2770 和进程 2771 先执行，优先级 85 和 86 在 top 下的数值为-86 和-87，可以看到刚开始有两个 RR 进程同时在运行各自拥有一半的 CPU 资源（47.8%和 47.2%）。当高优先级的 FIFO 进程创建后，几乎所有 CPU 资源都被该进程所占用（94.7%，剩余 5.3%是防止系统无法响应终端命令而保留的），当 FIFO 进程结束后两个 RR 进程又平分 CPU 资源。执行过程非常卡顿，因为此时实时进程执行期间，普通进程仅能保留 5%的资源。

等三个进程都结束后，我们来分析它们保留下来的 sched-pid 文件（即记录下来的/proc/PID/sched）中的信息，如图 8 所示。首先是调度类型，2770 和 2771 的 policy 是 2 对应 RR、优先级 prio 为 14，2773 的 policy 是 1 对应 FIFO、优先级为 13。

由于三个进程都是完成算术运算，因此只有一次自愿调度切换其他都是被动的调度切换。而 RR 是轮循，因此有多达 188 和 176 次被动调度切换，而 FIFO 只被系统打断而发生 23 次被动调度切换。由于 2773 进程比另两个进程优先级高，即使较后开始执行也可以抢到大部分 CPU 资源从而较早结束（16:47:54），而另两个 RR 进程则分

别在后面的 16:48:24 和 16:48:23 时间结束。

```
RR-FIFO-sched (2770, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       2344763.911291
se.vruntime                                  :             0.394917
se.sum_exec_runtime                          :         17440.759763
se.nr_migrations                             :                   0
nr_switches                                  :                 189
nr_voluntary_switches                        :                   1
nr_involuntary_switches                      :                 188
se.load.weight                               :             1048576
se.runnable_weight                           :             1048576
se.avg.load_sum                              :               46891
se.avg.runnable_load_sum                     :               46891
se.avg.util_sum                              :              508928
se.avg.load_avg                              :                1024
se.avg.runnable_load_avg                     :                1024
se.avg.util_avg                              :                  10
se.avg.last_update_time                      :       2291387249664
policy                                       :                   2
prio                                         :                  14
clock-delta                                  :                  10
mm->numa_scan_seq                            :                   0
numa_pages_migrated                          :                   0
numa_preferred_nid                           :                  -1
total_numa_faults                            :                   0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
2021年 05月 20日 星期四 16:48:24 CST
```

图 8(a). sched-2770

```
RR-FIFO-sched (2771, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       2342885.331048
se.vruntime                                  :             0.104690
se.sum_exec_runtime                          :         17428.900884
se.nr_migrations                             :                   0
nr_switches                                  :                 177
nr_voluntary_switches                        :                   1
nr_involuntary_switches                      :                 176
se.load.weight                               :             1048576
se.runnable_weight                           :             1048576
se.avg.load_sum                              :               47696
se.avg.runnable_load_sum                     :               47696
se.avg.util_sum                              :             6292789
se.avg.load_avg                              :                1024
se.avg.runnable_load_avg                     :                1024
se.avg.util_avg                              :                 125
se.avg.last_update_time                      :       2289661069312
policy                                       :                   2
prio                                         :                  14
clock-delta                                  :                  15
mm->numa_scan_seq                            :                   0
numa_pages_migrated                          :                   0
numa_preferred_nid                           :                  -1
total_numa_faults                            :                   0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
2021年 05月 20日 星期四 16:48:23 CST
```

图 8(b). sched-2771

```
RR-FIFO-sched (2773, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :        2314861.291389
se.vruntime                                  :              0.000000
se.sum_exec_runtime                          :          17523.950614
se.nr_migrations                             :                     0
nr_switches                                  :                    24
nr_voluntary_switches                        :                     1
nr_involuntary_switches                      :                    23
se.load.weight                               :               1048576
se.runnable_weight                           :               1048576
se.avg.load_sum                              :                 47740
se.avg.runnable_load_sum                     :                 47740
se.avg.util_sum                              :              24289708
se.avg.load_avg                              :                  1024
se.avg.runnable_load_avg                     :                  1024
se.avg.util_avg                              :                   504
se.avg.last_update_time                      :         2296391923712
policy                                       :                     1
prio                                         :                    13
clock-delta                                  :                    14
mm->numa_scan_seq                            :                     0
numa_pages_migrated                          :                     0
numa_preferred_nid                           :                    -1
total_numa_faults                            :                     0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
2021年 05月 20日 星期四 16:47:54 CST
```

图 8(c). sched-2773

3） 在一个空闲的双核 Linux 系统上启动 P1/P2 一直就绪不阻塞进程。（15 分）

要求：展示并记录/proc/cpuinfo 给出的系统核数，展示并记录 top 命令给出的 CPU 资源分配情况。利用 nice 设置比 p1、p2 优先级低的 p3 不阻塞进程，用 top 查看并记录负载均衡现象，用/proc/PID/sched 展示并记录各进程在处理器核间的迁移次数。

/proc/cpuinfo 查看系统核数，如图 9 所示。

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
stepping        : 10
microcode       : 0xe0
cpu MHz         : 1800.000
cache size      : 6144 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
t syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl
e3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadlin
prefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp f
adx smap clflushopt xsaveopt xsavec xsaves arat md_clear flush
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_stor
bogomips        : 3600.00
clflush size    : 64
cache_alignment : 64
address sizes   : 43 bits physical, 48 bits virtual
power management:
```

图 9(a). 系统核 1 的信息

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
stepping        : 10
microcode       : 0xe0
cpu MHz         : 1800.000
cache size      : 6144 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
t syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl
e3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadlin
prefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp f
adx smap clflushopt xsaveopt xsavec xsaves arat md_clear flush
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_stor
bogomips        : 3600.00
clflush size    : 64
cache_alignment : 64
address sizes   : 43 bits physical, 48 bits virtual
power management:
```

图 9(b). 系统核 2 的信息

然后展示 top 命令，如图 10(a)所示。

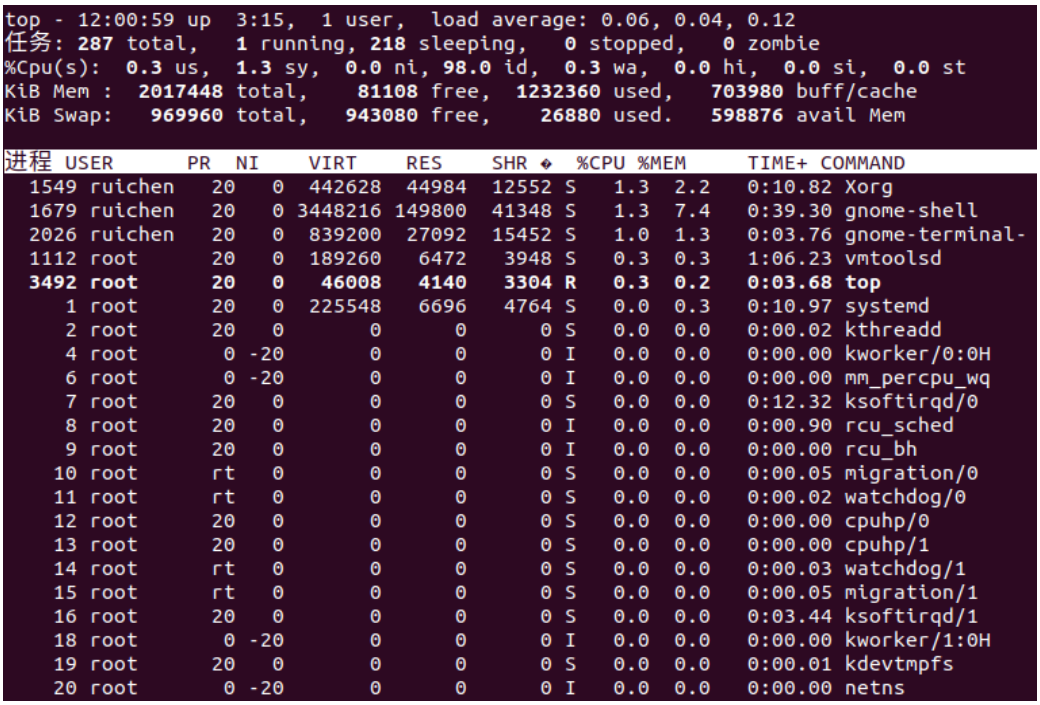


图 10(a). 令查看系统中按 CPU 使用率排序的进程列表

从第一行可以看出，该系统已经启动运行了 3 小时 15 分钟，有 1 个用户登录（同一个帐号多次登录计为多个），平均负载 load average 为 0.06/0.04/0.12。第二行表明任务总 tasks 为 287，就绪 running 进程 1 个，218 个进程处于阻塞状态，处于暂停/被跟踪 stop 状态的进程数为 0，处于僵尸 zombie 状态的进程数为 0。以“%Cpu(s)”标志开头的第三行是 CPU 使用情况的统计，默认是将所有核的利用率一起统计。其中 0.3us 表示 0.3%的时间用于运行用户空间的代码，相应地有 sy 应运行内核代码占用 CPU 时间的百分比、ni 表示低优先级用户态代码占用 CPU 时间的百分比、id 表示空闲（运行 idle 任务进程）CPU 时间的百分比、wa 表示 IO 等待占用 CPU 时间的百分比、hi 表示硬件中断（Hardware IRQ）占用 CPU 时间的百 分比、si 表示软中断（Software Interrupts）占用 CPU 时间的百分比以及和虚拟化有关的 st（steal time）占用 CPU 时间的百分比。

第 4 行和第 5 行是内存相关的统计。后面各列是各个进程的统计信息：PR 和 NI 是优先级和 NICE 值，S 是进程的状态， 它们与调度有关；%CPU 是该进程占用单个 CPU 时间的百分比；TIME 是进程在 CPU 上运行的时间（不计阻塞时间）。VIRT、RES 和 SHR 与调度没有直接关系，分别是进程虚存空间大小、物理内存占用量和共享物理内存大小（按 KB 统计），%MEM 指出该进程占系统物理内存总量的百分比。

如果按数字“1”键将会分别显示各处理器的统计信息，可以不用查看/proc/cpuinfo 也获得处理器个数的信息。在个人系统上执行 top 命令并按“1”键，分别显示 CPU0 和 CPU1 各自的统计信息如图 10(b)所示。

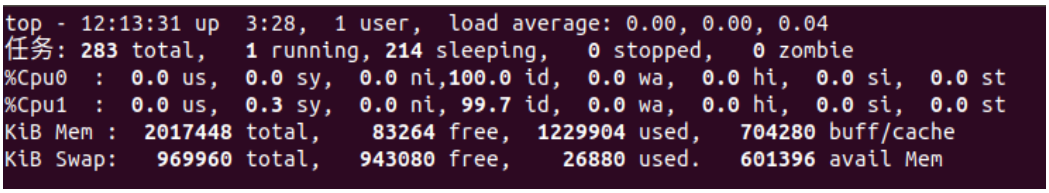


图 10(b). 显示各处理器的统计信息

接下来，利用 nice 设置比 p1、p2 优先级低的 p3 不阻塞进程，用 top 查看并记录负载均衡现象，用/proc/PID/sched 展示并记录各进程在处理器核间的迁移次数。
首先，我们设置图 1 所示的死循环代码，然后编译形成三个可执行文件 p1、p2 和 p3. 然后编写脚本如图 11 所示，前两个 NICE 值设置为 1（优先级较高），最后一个设置为 10（优先级较低），然后运行该脚本，结果如图 12 所示。

```
nice -n 1 ./p1 &
nice -n 1 ./p2 &
nice -n 10 ./p3 &
```

图 11. 三个程序同时运行且优先级不同的脚本

```
进程 USER      PR  NI    VIRT    RES    SHR ◆  %CPU %MEM     TIME+ COMMAND
  3948 root      21   1    4376    716    652 R  95.0  0.0   0:09.99 p1
  3949 root      21   1    4376    708    644 R  88.0  0.0   0:09.45 p2
  3950 root      30  10    4376    840    780 R  12.0  0.0   0:01.26 p3
```

图 12. 脚本运行结果

从图中我们可以看到，三个进程都没有阻塞，但是由于优先级设置的原因，p1、p2 的 CPU 占比是 p3 的 9 倍左右，二者的 CPU 使用率是接近的，体现了负载均衡现象。
此外，我们再来观察/proc/PID/sched 的内容，如图 13 所示。

```
p1 (3948, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :      14791915.275052
se.vruntime                                  :        685134.847951
se.sum_exec_runtime                          :        578782.995078
se.nr_migrations                             :                   13
nr_switches                                  :                22890
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                22890
se.load.weight                               :               839680
se.runnable_weight                           :               839680
se.avg.load_sum                              :                47416
se.avg.runnable_load_sum                     :                47416
se.avg.util_sum                              :             41747463
se.avg.load_avg                              :                  819
se.avg.runnable_load_avg                     :                  819
se.avg.util_avg                              :                  880
se.avg.last_update_time                      :       14791915274240
policy                                       :                    0
prio                                         :                  121
clock-delta                                  :                   15
mm->numa_scan_seq                            :                    0
numa_pages_migrated                          :                    0
numa_preferred_nid                           :                   -1
total_numa_faults                            :                    0
current_node=0, numa_group_id=0
numa_faults node=0 task_private=0 task_shared=0 group_private=0 group_shared=0
```

图 13(a). proc/3948/sched 内容

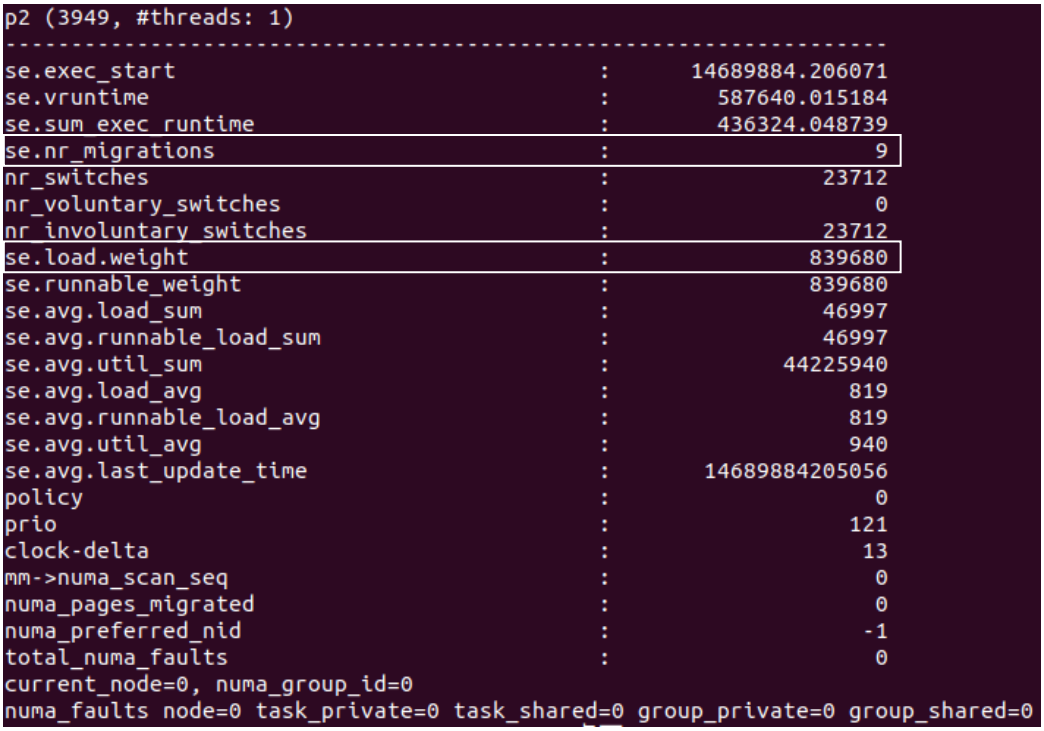


图 13(b). proc/3949/sched 内容

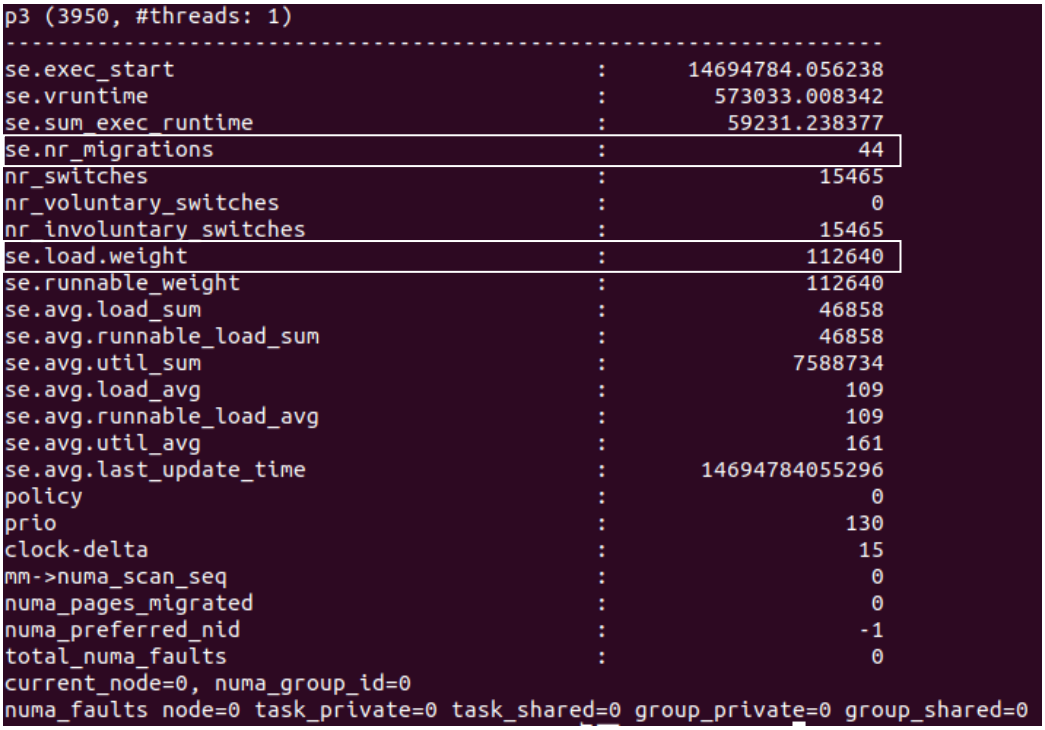


图 13(c). proc/3950/sched 内容

从图中可以看出，图 13(a)和图 13(b)对应的 p1、p2 的负载权重*se.load.weight* 是相同的（均为 839680），体现了负载均衡现象；而图 13(c)对应的 p3 的负载权重为 112640，前后差 8 倍左右，体现了优先级的差异。

再观察各进程迁移次数*se.nr_migrations*，分别为 13(PID=3948)、9(PID=3949)和 44(PID=3950)；可以看出负载较轻的进程被频繁地迁移。

2. 模拟在单处理器多进程操作系统的 CPU 调度

调度算法分为非抢占调度算法和抢占调度算法。

非抢占调度算法：系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成。在本次实验中先来先服务算法、短作业优先算法、高响应比优先算法以及时间片轮转算法都属于非抢占式调度。

抢占式调度算法：系统同样把处理机分配给优先权最高的进程, 使之执行。但在其执行期间，只要又出现了另一个优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。像多级反馈队列调度算法就属于抢占式调度。

（1） 先来先服务算法(first come first serve, FCFS)

FCFS 属于非抢占式调度算法，是相对来说最直观的调度。对于所有作业，FCFS 的思想就是按照时间先后顺序，选择最先来的作业来执行。在就绪队列中，每次都选择队首进程来执行。我们以如表 1 所示的作业来进行说明。

Job	time_start	time_need
A	0	4
B	1	3
C	2	5
D	3	2
E	4	4

表 1. FCFS 所有工作开始时间及服务时间

由于 A 最先到达，随后是 B、C、D 和 E，所以 FCFS 按照这个顺序将所有工作推入队列，然后按照队列顺序来进行执行。时序图如图 14 所示。

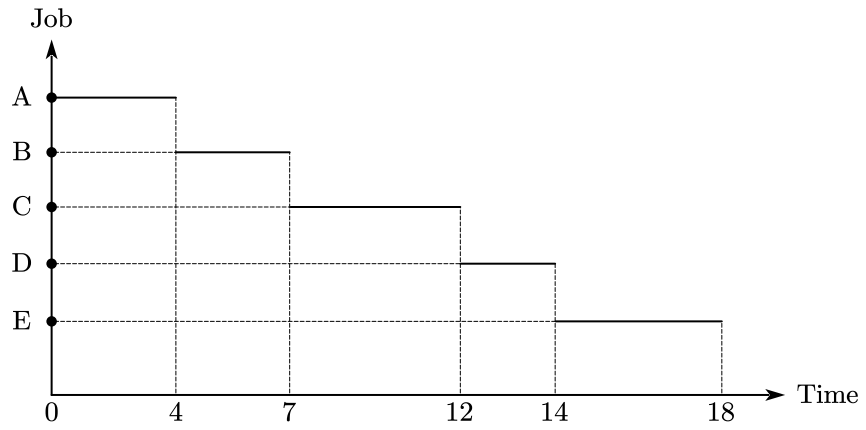


图 14. FCFS 调度时序图

关于周转时间以及带权周转时间，

A 工作的周转时间为 4，带权周转时间为 $\frac{4}{4}=1$；

B 工作的周转时间为$7-1=6$，带权周转时间为$\frac{6}{3}=2$；

C 工作的周转时间为$12-2=10$，带权周转时间为$\frac{10}{5}=2$；

D 工作的周转时间为$14-3=11$，带权周转时间为$\frac{11}{2}=5.5$；

E 工作的周转时间为$18-4=14$，带权周转时间为$\frac{14}{4}=3.5$。

而这部分的代码已经给出，我们运行这部分的代码，结果如图 15 所示，可以看到运行结果与我们的计算是吻合的。

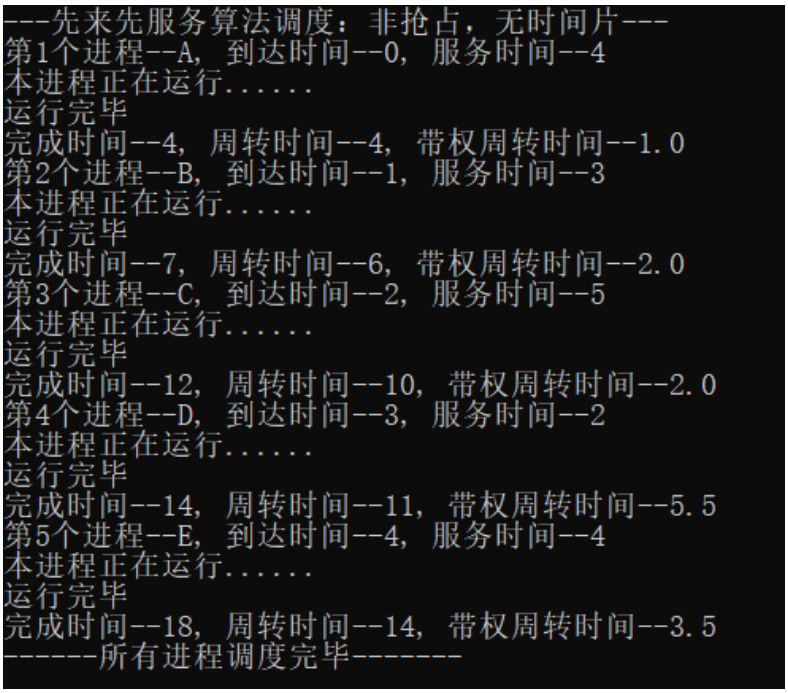


图 15. FCFS 调度运行结果

（2） 短作业优先调度算法(shortest job first, SJF)

短作业优先调度也属于非抢占调度算法。它的主要思想是，优先选择运行时间最短的作业或者进程来执行。很明显，这种方法更适合短作业或者短进程，但是由于这种算法的特殊调度方式而使得长作业/进程长时间不被调度。这种情况下，若一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度，出现所谓“饥饿”现象。

我们先用一个例子来说明 SJF 的工作流程。以如表 2 所示的例子为例，这里面很明显 Job3 的服务时间最少，Job2 的服务时间最长，所以 Job3 先服务，然后是 Job1，Job4，最后是 Job3.

Job	time_start	time_need
Job1	1	9
Job2	1	16
Job3	1	3
Job4	1	11

表 2. SJF 所有工作开始时间及服务时间

根据短作业优先原则，我们绘制调度时序图，如图 16 所示。

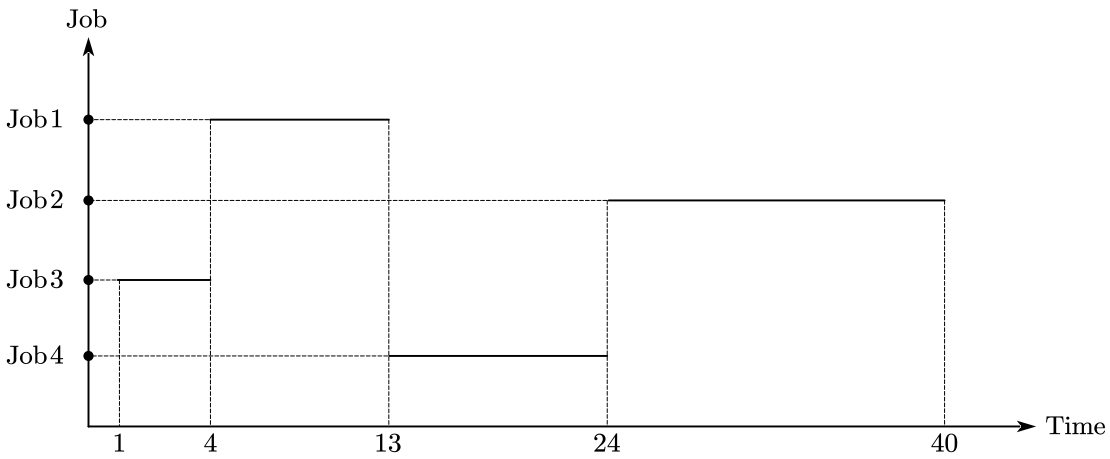


图 16. SJF 调度时序图

关于周转时间以及带权周转时间，

Job3 的周转时间为 3，带权周转时间为 $\frac{3}{3}=1$；

Job1 的周转时间为 $13-1=12$，带权周转时间为 $\frac{12}{9}\approx 1.3$；

Job4 的周转时间为 $24-1=23$，带权周转时间为 $\frac{23}{11}\approx 2.1$；

Job2 的周转时间为 $40-1=39$，带权周转时间为 $\frac{39}{16}\approx 2.4$。

C 语言实现：

根据我们的计算方法，为了寻找第一个进行的进程，先检查到达时间最早的，此时没有其他进程所以只能执行该进程；如果最早到达时间有多个进程，像表 2 所示的情况，那么哪一个进程服务时间最短就执行哪一个进程，将其保存进 ready 数组的头元素中。

所以，我们先将所有进程开始时间保存进 order 数组，然后按照上述方法得到最先执行的进程，代码如图 17 所示。

```
127     for(i=0; i<num; i++)
128     {
129         order[i] = pcbdata[i].time_start;
130     }
131
132     for(i=0; i<num; i++)
133     {
134         if(f >= order[i] && ff > pcbdata[i].time_need)
135         {
136             f = order[i];
137             ff = pcbdata[i].time_need;
138             ready[0] = i;
139         }
140     }
```

图 17. C 语言实现 SJF 算法（1）

然后我们用 N 代表 ready 数组中元素个数。ready 数组实际上保存的是就绪进程编号，如果该数组中没有元素，那就代表所有进程都执行完了。所以我们设置一个循环，循环条件就是 N 不为 0.

此外我们还应注意，在该样例中所有进程都是同时进来的。如果进程并不是同时进来的，那么在一个进程执行结束的时候，我们要将 ready 数组进行重置。因为如果不实时更新的话，新进来的进程就算服务时间最短它也不会进来。此外，为了防止之前已经执行过的进程再被放入 ready 数组，我们可以设置 visited 数组，记录进程是否已经执行过了。ready 数组准备好之后，就可以通过排序来得到当前的最短作业 ready[0].

每次循环都会得到一个被执行的进程，该进程的周转时间、带权周转时间计算如图 18 所示。

```
148         visited[ready[0]] = 1;
149         printf("第%d个进程--%s, ",ct,pcbdata[ready[0]].name);
150         ct++;
151         printf("到达时间--%d, 服务时间--%d\n",order[ready[0]],pcbdata[ready[0]].time_need);
152         printf("本进程正在运行......\n");
153         pcbdata[ready[0]].state = 'E';
154         _sleep(1);
155         printf("运行完毕\n");
156         pcbdata[ready[0]].state = 'F';
157         temp += pcbdata[ready[0]].time_need;
158         time_stay = temp - pcbdata[ready[0]].time_start;
159         time_stay_w = (float)time_stay / pcbdata[ready[0]].time_need;
160         printf("完成时间--%d, 周转时间--%d, 带权周转时间--%.1f\n",temp,time_stay,time_stay_w);
```

图 18. C 语言实现 SJF 算法（2）

然后将所有当前已经到达且没有执行过的进程号保存金 ready 数组，如图 19 所示。

```
161         N = 0;  //重置ready数组
162         for(i=0; i<num; i++)
163         {
164             if(pcbdata[i].time_start < temp && visited[i] == 0)  //当前已存在的所有进程且没有执行过
165             {
166                 ready[N] = i;
167                 N++;
168             }
169         }
```

图 19. C 语言实现 SJF 算法（3）

得到 ready 数组之后，将他们的服务时长进行排序，然后 ready[0]就是下一个要执行的进程。代码如图 20 所示。

```
171         for(i=0; i<N; i++)   //对ready数组进行排序，得到最短作业(bubble sort)
172         {
173             for(j=i+1; j<N; j++)
174             {
175                 if(pcbdata[ready[i]].time_need > pcbdata[ready[j]].time_need)
176                 {
177                     tmp = ready[i];
178                     ready[i] = ready[j];
179                     ready[j] = tmp;
180                 }
181             }
182         }
```

图 20. C 语言实现 SJF 算法（4）

以上就是实现 SJF 算法的 C 语言程序。运行该程序，以我们的样例为例，得到的结果如图 21 所示，这与我们之前的计算是吻合的。

```
---短作业进程优先算法调度：非抢占，无时间片---
第1个进程--Job3，到达时间--1，服务时间--3
本进程正在运行.....
运行完毕
完成时间--4，周转时间--3，带权周转时间--1.0
第2个进程--Job1，到达时间--1，服务时间--9
本进程正在运行.....
运行完毕
完成时间--13，周转时间--12，带权周转时间--1.3
第3个进程--Job4，到达时间--1，服务时间--11
本进程正在运行.....
运行完毕
完成时间--24，周转时间--23，带权周转时间--2.1
第4个进程--Job2，到达时间--1，服务时间--16
本进程正在运行.....
运行完毕
完成时间--40，周转时间--39，带权周转时间--2.4
------所有进程调度完毕-------
```

图 21. SJF 调度运行结果

（3） 高响应比优先调度算法(highest response ratio next, HRRN)

事实，之前的调度算法都属于最高优先权(*highest priority first, HPF*)调度，只不过不同的算法有着不同的优先权设置。例如，FCFS 算法的优先权就是到达时间，SJF 算法的优先权就是服务时间。而 HRRN 调度的最高优先权是响应比。

我们对于响应比的定义：给定等待时间time_wait 以及服务时间time_need，响应比为

$$R_p = \frac{\text{time\_wait}+\text{time\_need}}{\text{time\_need}}$$

这里面由于涉及到了等待时间，所以每次一个进程运行结束之后都需要重新计算每一个未执行的进程的响应比。然后每一个轮次选择一个响应比最高的进程来执行。这里我们依然需要选择一个最先执行的进程。首先检查到达时间最早的进程。如果在最早到达时间有多个进程，那么由于此时响应比都是 1 无法通过这个指标来判断，所以只能默认编号在前的进程先执行。
我们以如表 3 所示的样例来进行 HRRN 调度。

Job	time_start	time_need
P1	10	8
P2	12	12
P3	14	4
P4	16	6

表 3. HRRN 所有工作开始时间及服务时间

首先，P1 进程先进来，所以先执行 P1 进程。所以对于 P1 进程，周转时间为 $18-10=8$，带权周转时间为 $\frac{8}{8}=1$。

P1 执行完毕之后，我们来计算 P2，P3 以及 P4 的响应比

$$R_p^2=\frac{(10+8-12)+12}{12}=1.5$$

$$R_p^3=\frac{(10+8-14)+4}{4}=2$$

$$R_p^4=\frac{(10+8-16)+6}{6}\approx 1.3$$

所以下一个执行的进程就是 P3，周转时间为$22-14=8$，带权周转时间为 $\frac{8}{4}=2$。

同样地，当 P3 执行完毕之后，再计算 P2 和 P4 的响应比

$$R_p^2=\frac{(18+4-12)+12}{12}\approx 1.8$$

$$R_p^4=\frac{(18+4-16)+6}{6}=2$$

所以下一个执行的是 P4，周转时间为$28-16=12$，带权周转时间为 $\frac{12}{6}=2$。

当 P4 执行完毕之后只剩 P2，那么最后执行的就是 P2，其响应比为

$$R_p^2=\frac{(22+6-12)+12}{12}\approx 2.3$$

然后计算其周转时间为$40-12=28$，带权周转时间为 $\frac{28}{12}\approx 2.3$。

调度时序图如图 22 所示。

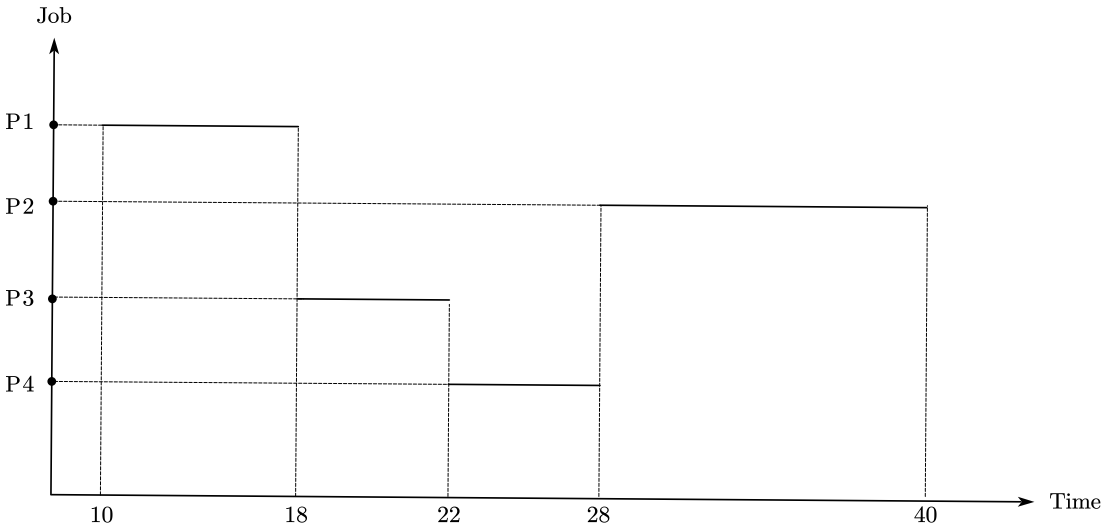


图 22. HRRN 调度时序图

关于 HRRN 调度的 C 语言程序实现，和之前的 SJF 调度类似。只不过在之前，我们每次都更新 ready 数组，然后对 ready 数组针对服务时间来进行排序。在这里面，我们也每次去重新更新 ready 数组，然后对 ready 数组中的进程分别计算他们的响应比。然后再进行排序，取 ready[0]就是当前最高响应比进程。

获取第一个执行的进程编号，代码如图 23 所示。

```
202     for(i=0; i<num; i++)
203     {
204         order[i] = pcbdata[i].time_start;
205     }
206
207     for(i=0; i<num; i++)
208     {
209         if(f > order[i])
210         {
211             f = order[i];
212             ready[0] = i;
213         }
214     }
```

图 23. C 语言实现 HRRN 算法（1）

其余内容与 SJF 算法实现流程类似，计算响应比并得到当前最大响应比的进程编号的代码如图 24 所示。

```
//计算响应比
for(i=0; i<N; i++)
{
    HRF[ready[i]] = (float)(temp - pcbdata[ready[i]].time_start + pcbdata[ready[i]].time_need) / pcbdata[ready[i]].time_need;
    printf("进程%s的响应比为%.1f\n",pcbdata[ready[i]].name,HRF[ready[i]]);
}
float maxi = -1;
int res;
for(i=0; i<N; i++)
{
    if (maxi < HRF[ready[i]])
    {
        maxi = HRF[ready[i]];
        res = ready[i];
    }
}
ready[0] = res;
```

图 24. C 语言实现 HRRN 算法（2）

代码运行结果如图 25 所示，可以看到结果与我们的计算结果是吻合的。

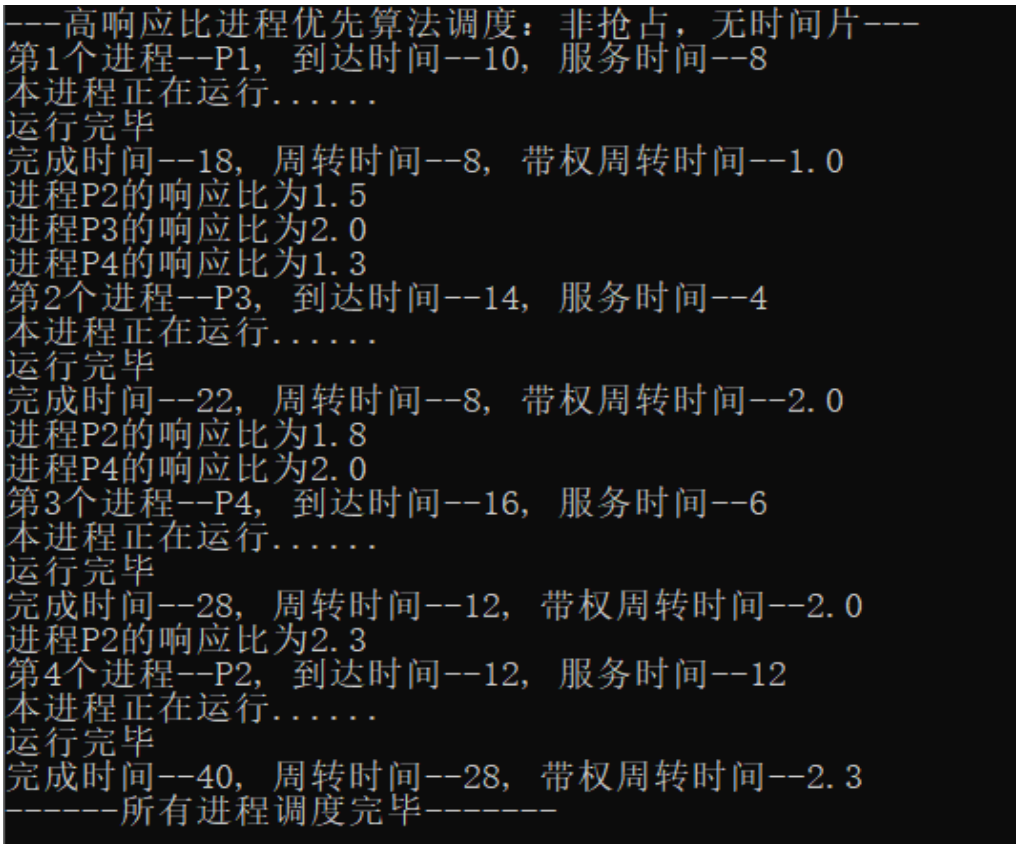


图 25. HRRN 调度运行结果

（4） 轮转法(round robin, RR)

基于时间片的轮转调度算法，将所有就绪进程按照先来先服务的方式组织成队列，然后每次将 CPU 分配给队首进程，该进程仅执行一个时间片。这个时间片长度是规定好的。时间片用完之后，操作系统终止进程，并将此次已经使用完时间片的进程送到就绪队列的末尾，然后再执行队首进程，也就是原来的队首的下一个进程。

我们以内置数据，也就是如表 4 所示数据来进行模拟。

Job	time_start	time_need
A	0	4
B	1	3
C	2	5
D	3	2
E	4	4

表 4. RR 所有工作开始时间及服务时间

从头开始，由于 A 最先到达，所以先执行 A，但并不执行完，只能执行一个时间片长度，这是设置为 2；当$\text{time}=2$的时候，B 和 C 都已经到达，所以按照到达顺序排列。然后 A 将该时间片用完，排在队列的最后即可。该过程如图 26 所示。此时对 A，$\text{time\_left}_{\text{A}}=2$；

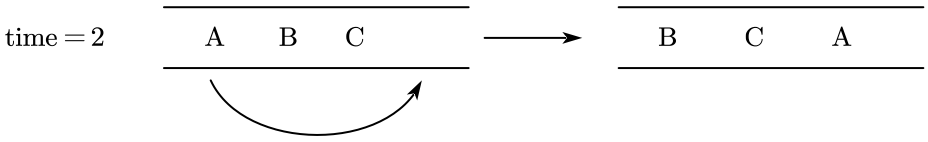


图 26. time=2 的时候就绪队列的变化

接下来执行 B，同样地，执行一个时间片之后$\text{time} = 4$，$\text{time\_left}_{\text{B}} = 1$。此时 D 和 E 都到达了，所以在$\text{time} = 4$时间点，就绪队列发生的变化如图 27 所示。

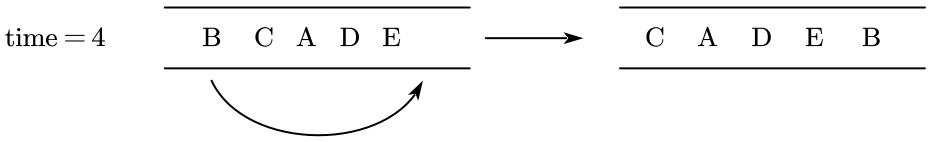


图 27. time=4 的时候就绪队列的变化

接下来执行 C，同样地，执行一个时间片之后$\text{time} = 6$，$\text{time\_left}_{\text{C}} = 3$。就绪队列发生的变化如图 28 所示。

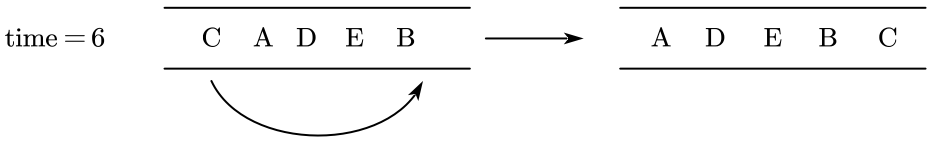


图 28. time=6 的时候就绪队列的变化

接下来执行 A，同样地，执行一个时间片之后$\text{time} = 8$，此时 A 没有剩余时间，已经执行完了。就绪队列发生的变化如图 29 所示。

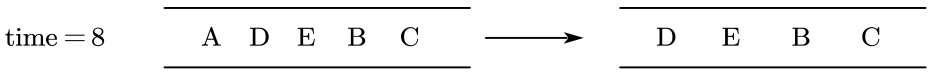


图 29. time=8 的时候就绪队列的变化

接下来执行 D，同样地，执行一个时间片之后$\text{time} = 10$，此时 D 没有剩余时间，已经执行完了。就绪队列发生的变化如图 30 所示。

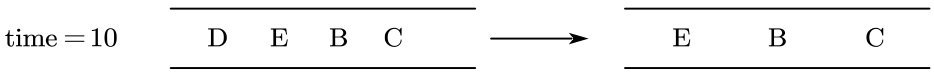


图 30. time=10 的时候就绪队列的变化

接下来执行 E，同样地，执行一个时间片之后$\text{time} = 12$，此时$\text{time_left}_{\text{E}} = 2$，就绪队列发生的变化如图 31 所示。

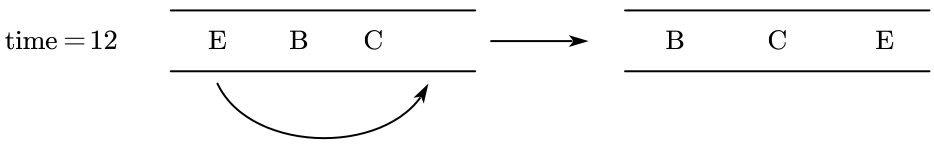


图 31. time=12 的时候就绪队列的变化

接下来执行 B，由于之前计算$\text{time_left}_B = 1$不够一个时间片长度，所以仅仅跑半个时间片就应该更新就绪队列，如图 32 所示。

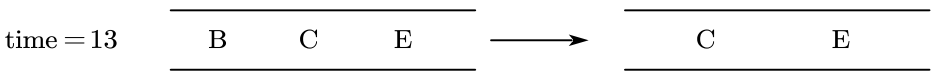


图 32. time=13 的时候就绪队列的变化

接下来执行 C，同样地，执行一个时间片之后$\text{time} = 15$，此时$\text{time_left}_C = 1$，就绪队列发生的变化如图 33 所示。

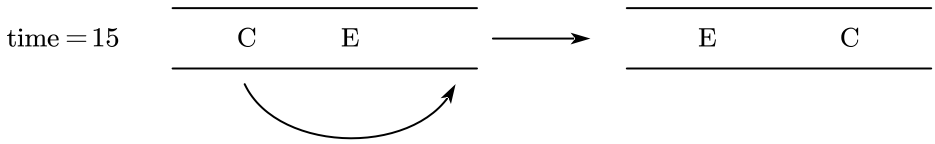


图 33. time=15 的时候就绪队列的变化

接下来执行 E，同样地，执行一个时间片之后$\text{time} = 17$，此时 E 执行完了，就绪队列发生的变化如图 34 所示。

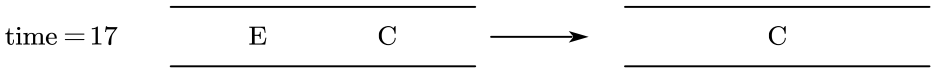


图 34. time=17 的时候就绪队列的变化

最后执行 C，C 还剩 1 个时间，执行完毕之后$\text{time} = 18$，然后队列为空，调度结束。整理上面过程，得到 RR 调度时序图如图 35 所示。

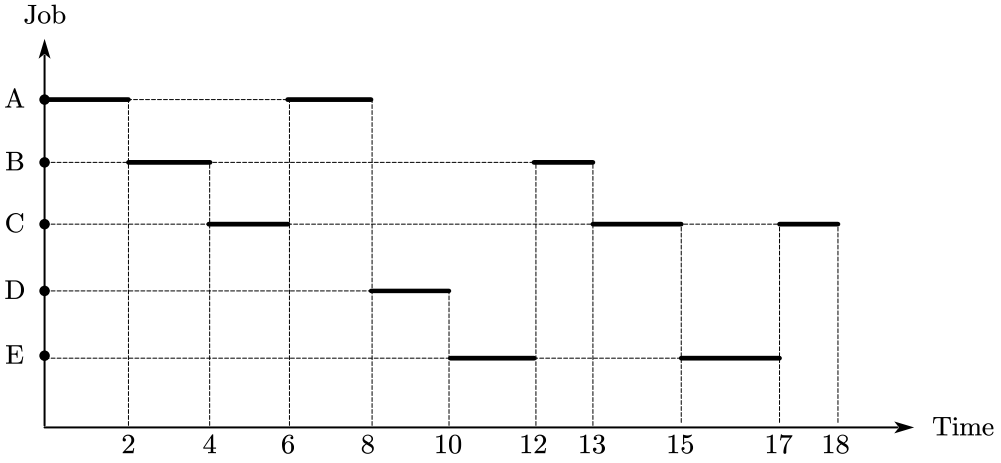


图 35. RR 调度时序图

结合时序图，我们可以计算各个进程的周转时间以及带权周转时间如下

A 的周转时间为 8，带权周转时间为$\frac{8}{4} = 2$；

B 的周转时间为$13-1=12$，带权周转时间为$\frac{12}{3}=4$；

C 的周转时间为$18-2=16$，带权周转时间为$\frac{16}{5}=3.2$；

D 的周转时间为$10-3=7$，带权周转时间为$\frac{7}{2}=3.5$；

E 的周转时间为$17-4=13$，带权周转时间为$\frac{13}{4}=3.25$。

C 语言实现时间片轮转算法：
首先，结合我们之前的说明，易知这里面要对一个队列操作。我们用 struct 结构体来实现一个队列结构，包括队列初始化、队头返回、判空、推入以及弹出操作。
队列定义：如图 36 所示。

```
268  //定义队列结构
269  typedef struct queue
270  {
271      int *list = (int*) malloc(100*sizeof(int));
272      int front;
273      int rear;
274  }Queue;
```

图 36. RR 调度—队列定义

初始化、队头返回操作，如图 37 所示。

```
276  void init (Queue *SQ)
277  {
278      SQ->front = 0;
279      SQ->rear = 0;
280  }
281
282  int front (Queue *SQ)
283  {
284      return SQ->list[0];
285  }
286
```

图 37. RR 调度—队列操作定义（1）

将元素推入队列，如图 38 所示。

```
299  void push (Queue *SQ, int expr)
300  {
301      SQ->list[SQ->rear] = expr;
302      SQ->rear = SQ->rear + 1;
303  }
```

图 38. RR 调度—队列操作定义（2）

队列判空：如图 39 所示。

```
287   int empty (Queue *SQ)
288   {
289       if (SQ->front == SQ->rear && SQ->front == 0)
290       {
291           return 1;
292       }
293       else
294       {
295           return 0;
296       }
297   }
```

图 39. RR 调度—队列操作定义（3）

队列弹出操作，这实际上是一个数组元素前移的过程，并更新头尾游标。如图 40 所示。

```
305   int pop (Queue *SQ)
306   {
307       if (SQ->front == SQ->rear)
308       {
309           return -1;
310       }
311       else
312       {
313           int length = SQ->rear - SQ->front;
314           for(int i = 0; i < length - 1; i++)
315           {
316               SQ->list[i] = SQ->list[i+1];
317           }
318           SQ->rear = SQ->rear - 1;
319           return 0;
320       }
321   }
```

图 40. RR 调度—队列操作定义（4）

队列准备好之后进行初始化，然后我们开始准备 ready 数组的值。首先将所有进程到达时间保存进 order 数组，ready 数组按序号进行保存即可。然后按照到达时间进行排序，ready 数组随之移动。然后将 ready[0]推入队列。
我们设置一个循环，循环条件就是队列不为空。每执行一个进程之后，就要更新该进程还剩下的时间，也就是减去时间片单元长度。这里要对剩下的时间分类来进行处理：

1） 剩下时间等于 0，这就代表该进程刚好执行完毕。此时我们仅仅需要将时间增加一个时间单元，代表时间又增长了一个时间片长度；同时输出提示信息，计算周转时间以及带权周转时间，代码如图 41 所示；

```
if(pcbdata[front(&SQ)].time_left == 0)
{
    int cur1 = cur;
    cur += time_unit;
    printf("在时间片%d->%d区间发生的事情：\n",cur1,cur);
    printf("当前运行的进程：%s\n",pcbdata[front(&SQ)].name);
    printf("进程%d运行结束，其名称为%s，周转时间为%d，带权周转时间为%.1f\n",
        pcbdata[front(&SQ)].id,pcbdata[front(&SQ)].name,
        cur-pcbdata[front(&SQ)].time_start,
        (float)(cur-pcbdata[front(&SQ)].time_start)/pcbdata[front(&SQ)].time_need);
    pcbdata[front(&SQ)].state = 'F';
}
```

图 41. RR 调度—情况（1）

2） 剩下时间小于 0，这就代表该进程在时间片单元中间就停止了。那么此时时间不应再增长一个时间片长度，而是应该增加剩余时间加上时间片单元（这个长度小于时间片单元长度）的长度。然后由于该进程事实上也结束了，所以也要计算该进程的周转时间以及带权周转时间。代码如图 42 所示。

```
else if (pcbdata[front(&SQ)].time_left < 0)
{
    //eg.如果时间片为3，但是只用了一个，此时time_left为-2，而cur仅仅应该+1
    int cur1 = cur;
    cur += (pcbdata[front(&SQ)].time_left + time_unit);
    printf("在时间片%d->%d区间发生的事情：\n",cur1,cur);
    printf("当前运行的进程：%s\n",pcbdata[front(&SQ)].name);
    printf("进程%d运行结束，其名称为%s，周转时间为%d，带权周转时间为%.1f\n",
        pcbdata[front(&SQ)].id,pcbdata[front(&SQ)].name,
        cur-pcbdata[front(&SQ)].time_start,
        (float)(cur-pcbdata[front(&SQ)].time_start)/pcbdata[front(&SQ)].time_need);
    pcbdata[front(&SQ)].state = 'F';
}
```

图 42. RR 调度—情况（2）

3） 剩下时间大于 0，这就代表该进程在时间片单元并没有运行完。这时我们仅仅需要输出提示信息即可。代码如图 43 所示。

```
else  //该进程未运行完毕
{
    int cur1 = cur;
    cur += time_unit;
    printf("在时间片%d->%d区间发生的事情：\n",cur1,cur);
    printf("当前运行的进程：%s\n",pcbdata[front(&SQ)].name);
    printf("该时间片运行的是进程%d,名称为%s,该进程还需服务时间%d\n",
        pcbdata[front(&SQ)].id,
        pcbdata[front(&SQ)].name,
        pcbdata[front(&SQ)].time_left);
    flag = 1;
}
```

图 43. RR 调度—情况（3）

针对各种情况做出判断之后，和之前的算法一样，我们需要将没有进入过队列且当前已经到达的进程加入队列中。代码如图 44 所示。

```
for(i=0; i<num; i++)
{
    if(visited[i] == 0 && pcbdata[i].time_start <= cur)  //没有进入过队列且当前已经进来了的进程
    {
        push(&SQ,i);
        visited[i] = 1;
    }
}
```

图 44. RR 调度--更新就绪队列

此时队列的队首进程已经执行结束，所以为了体现队列结构的变化，我们先将原来的队列输出。如果队首进程没有执行完毕，那么就先将队首元素 pop 出去，然后 push 到队尾即可。如果队首进程已经执行完毕，那么直接 pop 出去而不用再 push 回队尾。运行程序，得到结果如图 45 所示，结果与我们的计算吻合。

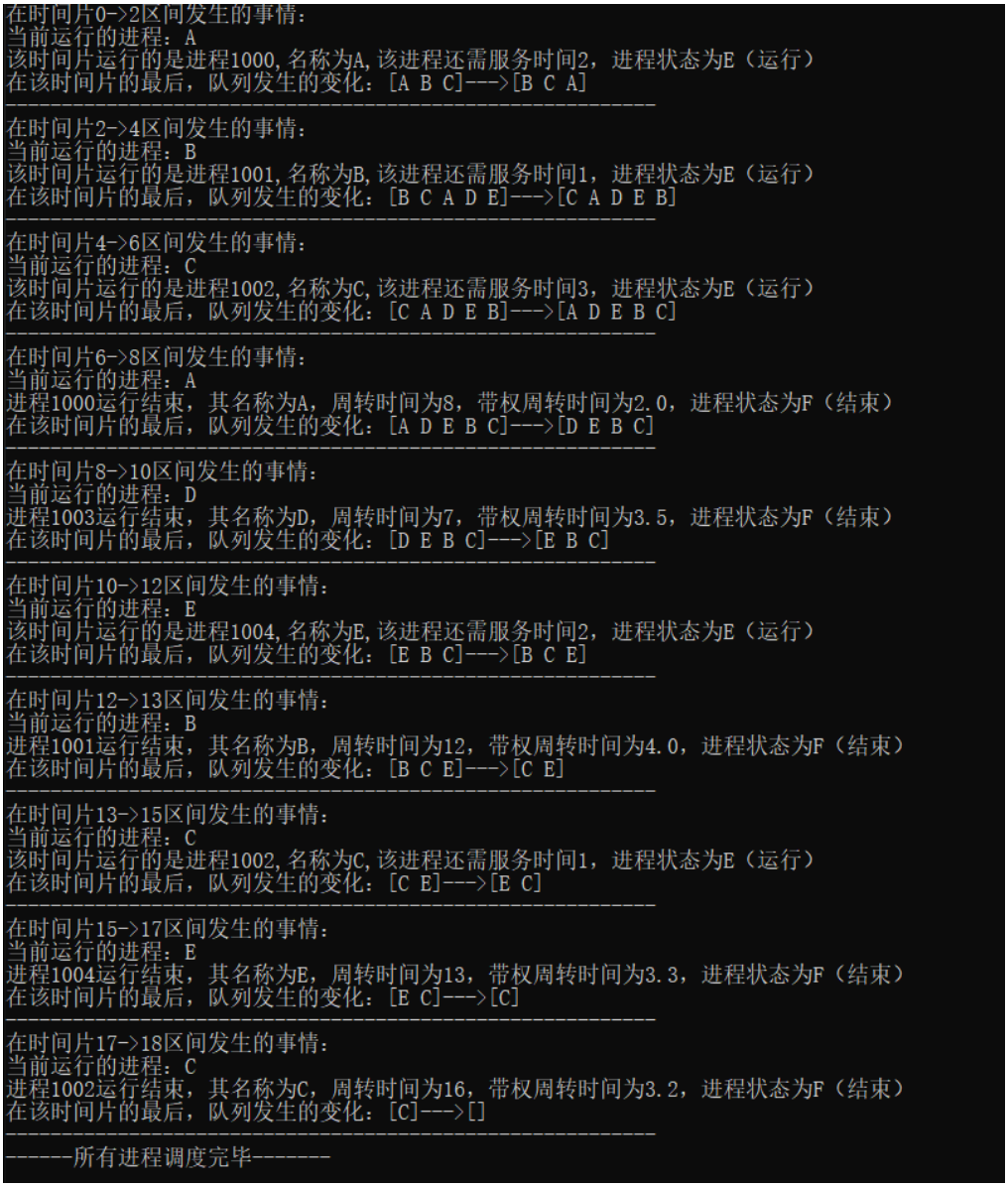


图 45. RR 调度运行结果

3. 附加部分（多级反馈队列算法）

多级反馈队列，第n级队列的优先级比第$n+1$级的队列优先级要高，所以这其实是一种抢占型的调度方式。我们以如下的例子来模拟多级反馈队列算法的运行。

队列：如表 5 所示。

queue	time_unit
I	2
II	4
III	8

表 5. 多级队列时间片设定

再给出各进程的到达时刻以及服务时间，如表 6 所示。

Job	time_start	time_need
A	0	7
B	5	4
C	7	13
D	12	9

表 6. 进程到达时间与服务时间

很明显 A 进程先到达，执行完 2 个时间点之后，A 的第一队列就执行完毕了，此时结果如图 46 所示。

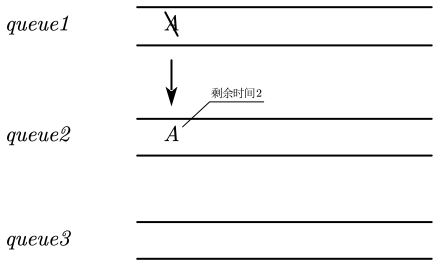


图 46. 时间点 2 队列变化

在 B 没有到达之前，A 在队列 2 中继续执行。所以时间点 5 之前一直是 A 在运行。时间点 5 到达之后，B 进入队列 1，所以此时抢占了 A 的运行，此时 A 剩余时间为 2。B 运行 2 个时间之后，也进入队列 2，此时 B 剩余时间为 2，队列结果如图 47 所示。

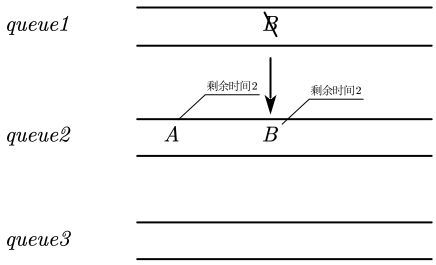


图 47. 时间点 7 队列变化

时间点 7 到达之后，C 进入了第一队列，此时抢占了第二队列的进程，所以 C 也运行 2 个时间点，然后进入第二队列。队列变化如图 48 所示。

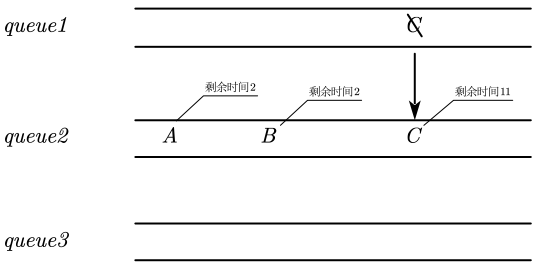


图 48. 时间点 9 队列变化

时间 9 到达之后，由于进程 D 还没到，所以队列 2 现在的进程可以开始执行了。所以在 D 到达之前执行队列 2 的进程，D 到达之后执行队列 D 然后将其放入队列 2；同样地，如果队列 2 的进程跑完队列 2 的时间片还没结束的话，就进入队列 3，同时注意队列 2 的抢占情况。综上所述，时序图如图 49 所示。

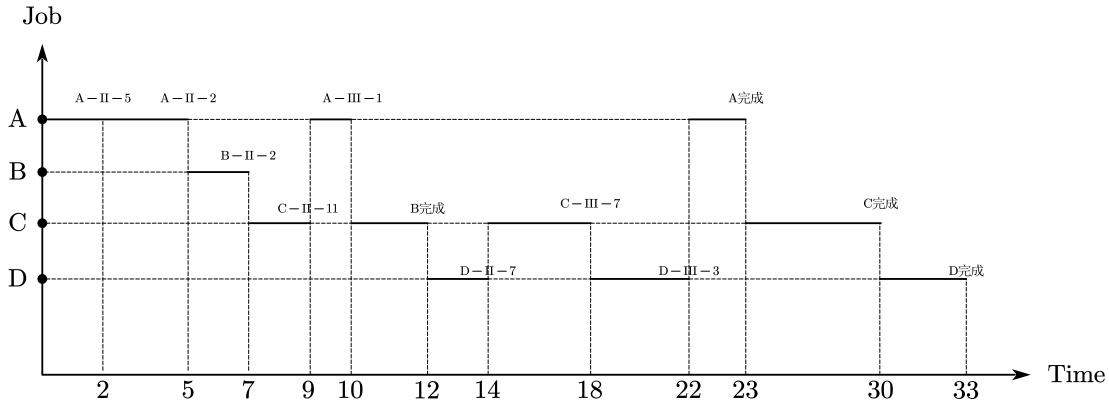


图 49. 多级反馈队列时序图

五、 实验体会

本次实验学习了 Linux 操作系统的调度相关的信息查看的相关操作，了解了优先级、多核、负载均衡等概念并通过实验进行了实际的查看。此外，本次实验详述了非抢占式算法之短作业优先算法、高响应比优先算法以及时间片轮转算法并通过 C 语言进行实现。关于拓展部分我们详述了多级反馈队列轮转算法。

本次实验内容较多，但是通过真正编程来实现算法可以加深对以上这些算法的理解。

六、 实验附件

1. CPU 调度算法 C 语言实现（FCFS、SJF、HRF、Timeslice）scheduling.cpp；

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：