

实现抢占式优先级调度算法

一、实验目的

了解 Linux 内核的进程调度算法，在此基础上实现具有优先级的抢占式进程调度算法，通过对调度算法的修改，掌握内核中此部分的代码结构，加深对进程调度算法的理解和进程管理的理解。

二、实验内容

修改内核现有的进程调度算法，为每个进程添加一个优先级，按照优先级高低进行调度。高优先级的进程在时间片没用完的情况下，优先被运行；当进程优先级相同时且时间片都没用完时，选择剩余时间片多的进程；当所有进程都没有时间片时，需要重新为进程分配时间片。

- 在 `task_struct` 中添加优先级并初始化该变量；
- 修改 `do_timer()` 函数，在每次发生时钟中断时，执行进程调度算法；
- 修改进程调度算法，为抢占式优先级调度算法；
- 添加系统调用，允许设置进程的优先级；
- 编译内核，并在 `qemu` 上运行；
- 编写测试程序，利用系统调用修改进程的优先级，测试调度算法的正确性。

三、实验指导

3.1 实验环境以及环境设置

实验环境：

Ubuntu18.04、Linux 0.11 内核、qemu 模拟器。

环境设置：

(1) 下载 qemu；

(2) 下载 Linux 0.11 内核，make 编译。可能存在编译工具链不一致或者 asm 代码格式的错误，需要下载相应工具链以及修改文件内容；

(3) 下载根文件系统映像，本实验指导采用的是 hdc-0.11.img (Harddisk image)；

(4) 查看是否能在 qemu 上运行编译好的 Image。

建议在上述步骤均能正确完成后，再进行本次实验。

3.2 在 task_struct 中添加优先级并初始化该变量

task_struct 在 include/linux/sched.h 中定义，添加一个新的成员变量 add_priority，代表进程的优先级。如下所示：

```
1. struct task_struct {  
2. ...  
3.     /* new defined priority by myself */  
4.     unsigned short add_priority;  
5. };
```

Linux 中所有进程都是进程 0 的子进程，即所有进程都是通过 fork 得到，因此我们在 fork 时初始化此变量，默认初始化为 0。

```
1. /*set the priority of the new task struct*/  
2.     p->add_priority = 0;
```

3.3 修改 do_timer()函数

该函数在 kernel/sched.c 中，其解释如下：

do_timer()函数则根据特权级对当前进程运行时间作累计。如果 CPL=0,则表示进程运行在内核态时被中断，因此内核就会把进程的内核态运行时间统计值 stime 增 1,否则把进程用户态运行时间统计值增 1。如果软盘处理程序 floppy.c 在操作过程中添加过定时器，则对定时器链表进行处理。若某个定时器时间到(递减后等于 0)，则调用该定时器的处理函数。然后对当前进程运行时间进行处理，把当前进程运行时间片减 1。时间片是一个进程在被切换掉之前所能持续运行的 CPU 时间，其单位是上面定义的嘀嗒数。如果进程时间片值递减后还于 0，表示其时间片还没有用完，于是就退出 do_timer()继续运行当前进程。如果此时进程时间片已经递减为 0，表示该进程已经用完了此次使用 CPU 的时间片，于是程序就会根据被中断程序的级别来确定进一步处理的方法。若被中断的当前进程是工作在用户态的(特权级别大于 0)，则 do_timer()就会调用调度序 schedule()切换到其他进程去运行。如果被中断的当前进程工作在内核态，也即在内核程序中运行时被中断，则 do_timer()会立刻退出。因此这样的处理方式决定了 Linux 系统的进程在内核态运行时不会被调度程序切换。（摘自《Linux 内核完全注释》）

请修改为在用户态的进程每次发生时钟中断时，执行进程调度算法，而不是等当前时间片用完了才去执行调度算法，因此实现抢占式调度。

3.4 修改进程调度算法

进程调度算法 schedule()函数在 kernel/sched.c 中。

3.4.1 原算法实现：

系统最多同时有 NR_TASKS（64）个进程，在 schedule()中，遍历所有的进程，找到剩余时间片最多的进程，于是运行这个进程。当所有进程的时间片都用完后，重新根据 priority（注意不是我们定义的 add_priority，原本代码中含有的变量）分配时间片。

3.4.2 新算法实现：

(1) 定义优先级的个数

在本实验指导中，定义了 5 个优先级，从低到高分别为 0、1、2、3、4。首先我们在这个文件的开始的位置，定义 TNOP (the number of priorities) 为 5。

```
1. #define TNOP 5
```

(2) 算法设计

如前面所提到的，已经为每个进程添加一个优先级，按照优先级高低进行调度。所有进程默认优先级为 0，可以通过系统调用设置进程的优先级。

算法设计为：高优先级的进程在时间片没用完的情况下，它优先被运行；当进程优先级相同且时间片都没用完时，选择剩余时间片多的进程；当所有进程都没有时间片时，需要重新为进程分配时间片。

举几个例子：

例 1：当存在一个优先级为 1 且时间片不为零的进程 A，其他进程优先级都为 0，那么优先运行 A。

例 2：如果存在两个优先级为 1 的进程 A 和 B，且 A 的时间片大于 B 的时间片，其他进程优先级都为 0，那么优先运行 A，等 A 时间片用完之后，再运行 B，然后再运行其他进程。

例 3：如果所有进程优先级都为 0，按照时间片的多少顺序运行，时间片越多越优先运行。

```
1. void schedule(void)
2. {
3.     ...
4.     /* this is the scheduler proper: */
5.
6.     while (1) {
7.
```

```

8.          // insert your code
9.
10.         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
11.             if (*p)
12.                 (*p)->counter = ((*p)->counter >> 1) +
13.                 (*p)->priority;
14.         }
15.         switch_to(next);
16. }

```

(3) 算法结果体现

当下一个要执行的进程的优先级不为 0 时，打印这个进程的信息，包括进程 pid、优先级 add_priority、剩余时间片 counter。

```

1. printk("pid: %ld, prio:%u, time: %ld\n", task[next]->pid,
   task[next]->add_priority, task[next]->counter);

```

3.5 添加系统调用

本次实验的添加系统调用和上次实验的过程不尽相同，但系统调用的本质还是一致的，下面简要介绍添加系统调用的过程。

(1) 定义 sys_setpriority() 函数

设置进程号为 pid 的进程的优先级为 prio。在 /kernel/sys.c 中添加函数如下：

```

1. int sys_setpriority(int pid, unsigned short prio)
2. {
3.
4.     if(prio < 5){
5.         if(!pid)
6.             return -1;

```

```
7.          // insert your code
8.      }
9.
10.     // return
11. }
```

（2）声明函数并添加系统调用号

在 include/unistd.h 中添加如下：

```
#define __NR_setpriority    74

int setpriority(int pid, unsigned short prio);
```

在 include/linux/sys.h 中添加如下：

```
extern int sys_setpriority();
```

并且在 sys_call_table 最后添加 sys_setpriority。

在 kernel/system_call.s 里面修改如下：

```
nr_system_calls = 75
```

3.6 编译并运行

（1）编译内核

\$ make clean

\$make

（2）运行

```
qemu-system-i386 -m 32 -boot a -fda Image -hda hdc-0.11.img
```

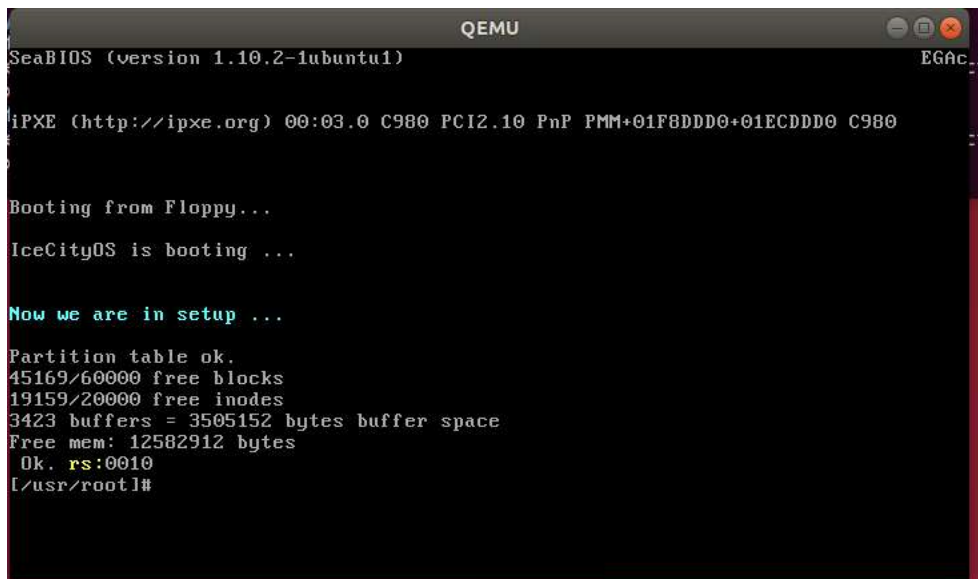


图 3.1 运行结果

3.7 测试程序

(1) 新建文件

之后在运行的 0.11 内核中，新建一个文件夹，并在此文件夹下新建一个 c 文件。

```
[/usr/root]# mkdir test
```

```
[/usr/root]# touch a.c
```

(2) 编写测试程序

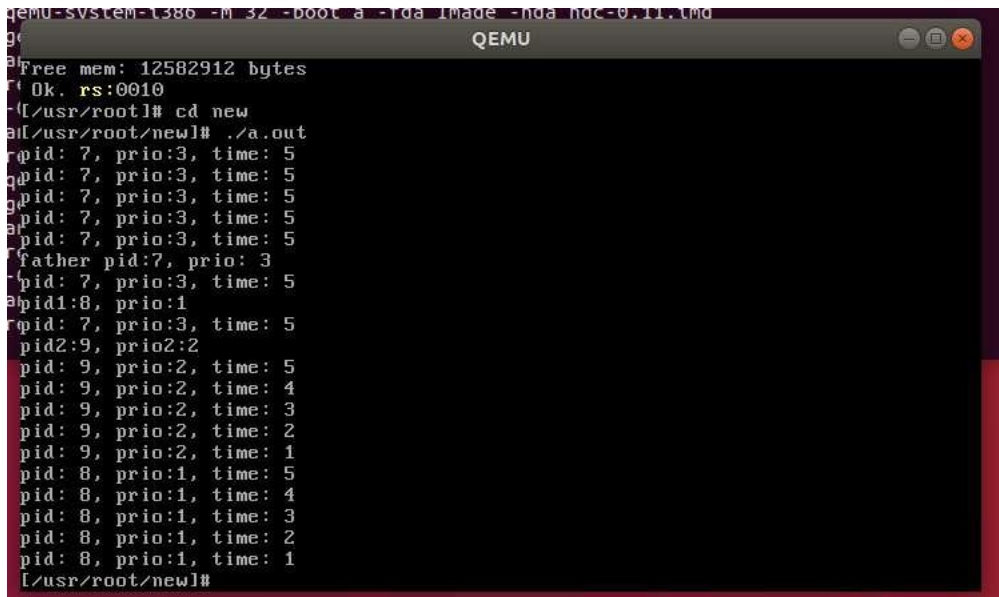
要进行系统调用，需要在文件中添加如下代码，并用 gcc 编译后运行：

```
1. #define __LIBRARY__
2. #include <unistd.h>
3. _syscall2(int, setpriority, int, pid, unsigned short, pri
o)
4. int main(){
5.     setpriority(getpid(), 3); // then the priority is
set to 3
6.     //your code
```

7. }

(3) 测试程序要求

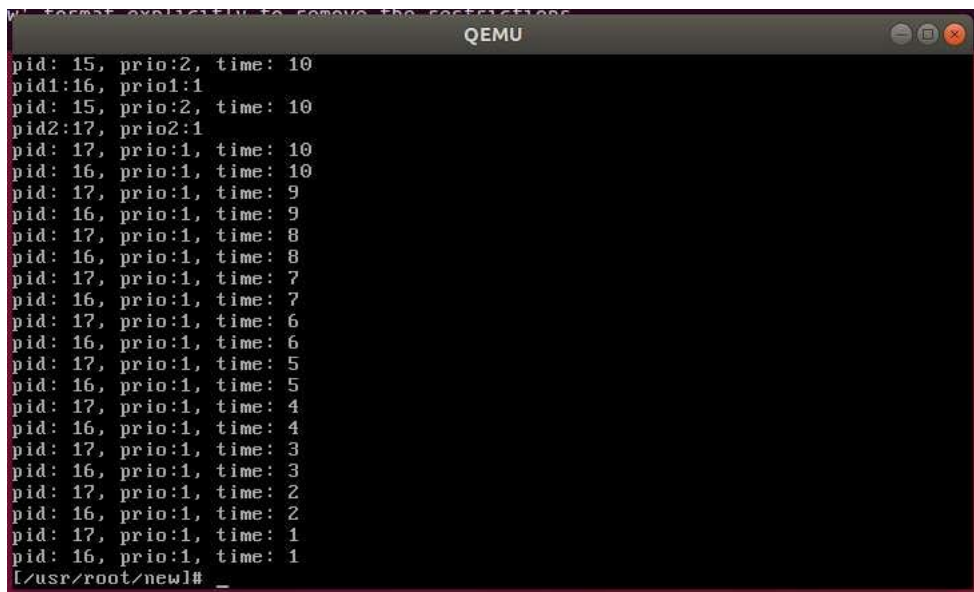
<1> 首先将该进程的优先级设置成 3，然后 fork 出两个进程（可以是两个死循环的进程），分别将两个子进程的优先级设置为 1 和 2，实现结果如下（该测试将每个进程初始时间片设置为 5）：



```
Free mem: 12582912 bytes
Ok. rs:0010
[/usr/root]# cd new
[/usr/root/new]# ./a.out
pid: 7, prio:3, time: 5
pid: 7, prio:3, time: 5
pid: 7, prio:3, time: 5
pid: 7, prio:3, time: 5
pid: 7, prio:3, time: 5
father pid:7, prio: 3
pid: 7, prio:3, time: 5
pid1:8, prio:1
pid: 7, prio:3, time: 5
pid2:9, prio:2
pid: 9, prio:2, time: 5
pid: 9, prio:2, time: 4
pid: 9, prio:2, time: 3
pid: 9, prio:2, time: 2
pid: 9, prio:2, time: 1
pid: 8, prio:1, time: 5
pid: 8, prio:1, time: 4
pid: 8, prio:1, time: 3
pid: 8, prio:1, time: 2
pid: 8, prio:1, time: 1
[/usr/root/new]#
```

图 3.2 测试程序 1 输出

<2> 和上述步骤类似，先将该进程的优先级设置成 2，然后 fork 出两个进程（可以是两个死循环的进程），分别将两个子进程的优先级设置为 1 和 1，实现结果如下（该测试将每个进程初始时间片设置为 10）：



```
pid: 15, prio:2, time: 10
pid1:16, prio:1
pid: 15, prio:2, time: 10
pid2:17, prio:1
pid: 17, prio:1, time: 10
pid: 16, prio:1, time: 10
pid: 17, prio:1, time: 9
pid: 16, prio:1, time: 9
pid: 17, prio:1, time: 8
pid: 16, prio:1, time: 8
pid: 17, prio:1, time: 7
pid: 16, prio:1, time: 7
pid: 17, prio:1, time: 6
pid: 16, prio:1, time: 6
pid: 17, prio:1, time: 5
pid: 16, prio:1, time: 5
pid: 17, prio:1, time: 4
pid: 16, prio:1, time: 4
pid: 17, prio:1, time: 3
pid: 16, prio:1, time: 3
pid: 17, prio:1, time: 2
pid: 16, prio:1, time: 2
pid: 17, prio:1, time: 1
pid: 16, prio:1, time: 1
[/usr/root/new]#
```


图 3.3 测试程序 2 输出

建议：Linux 0.11 运行起来容易死机，在编写代码和测试的时候尽量不要修改进程初始时间片，初始值为 15，当时间片越短越容易卡死。在实验基本完成后，为了方便截图可以修改时间片。

四、实验提交

- (1) 需上传两个测试结果的截图，并上传编写过或修改过的全部代码。
- (2) 分析两个测试程序的输出。
- (3) 不设置父进程的优先级（即优先级为 0），分别将两个进程的优先级设置为 1 和 2，测试其结果，上传运行结果截图并分析其原因。
- (4) 分析该算法的优劣。如果你设计调度算法，将如何设计？
- (5) 简单谈谈你在实验中遇到的困难。