操作系统

第三章 进程与处理机管理

文艳军

计算机学院

第2讲:进程的创建（回顾）

一. fork系统调用的使用

二. fork系统调用的实现

1. PCB的申请与初始化
2. 内存拷贝
3. 修改GDT

三. 独学&讨论：fork.c

第3讲:进程调度的时机与算法

- 进程调度：内核决定将CPU分配给某个就绪进程的过程
  - 选择一个就绪进程
    - 调度算法（策略）
  - 进程切换
    - 进程执行现场的切换

进程调度算法

① 先进先出调度算法（FIFO）

② 时间片轮转调度算法

③ 优先级调度算法

④ 短进程优先调度算法

⑤ 最短剩余时间优先调度算法

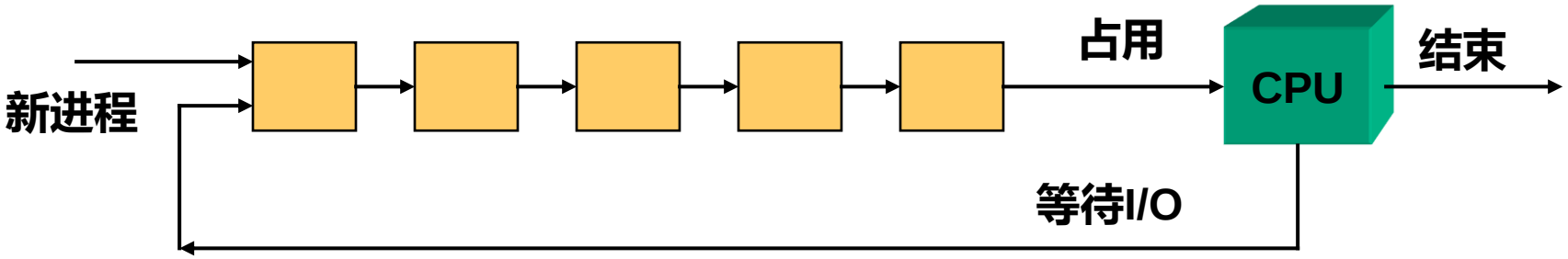
⑥ 最高响应比优先法

⑦ 多级反馈队列调度算法

目录

一. 先进先出调度算法（FIFO）

- 按照进程就绪的先后次序来调度进程（非剥夺调度）



优点:实现简单，公平
缺点:短进程的响应时间比较长，不适合交互环境

进程调度的时机

① 当前进程主动放弃处理机时（非剥夺调度）

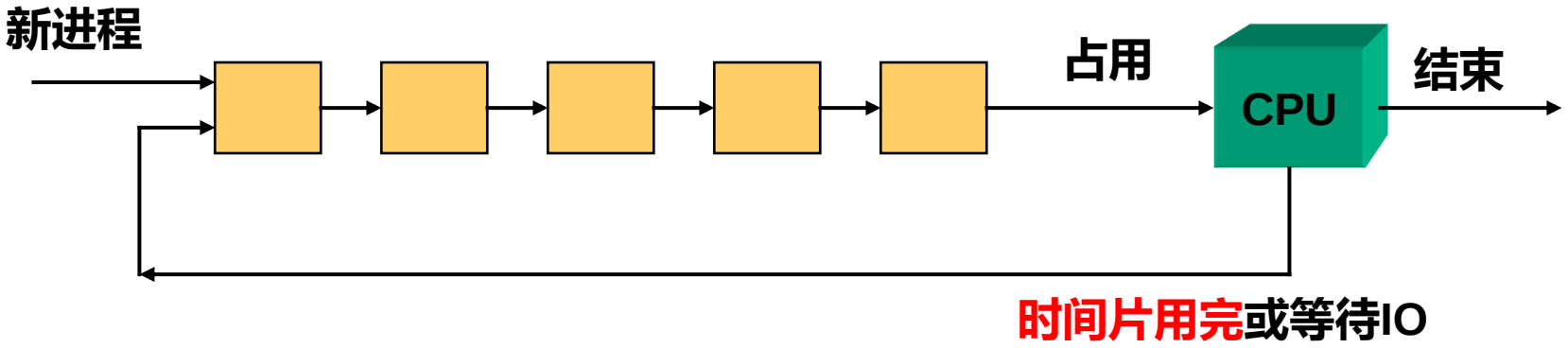
可能事件：

- 进程结束（ exit ）
- 进程被阻塞或挂起（ read, pause, sleep_on, ... ）
- “自愿放弃”系统调用（yield）

目录

二. 时间片轮转调度算法

- 赋给每个进程一个CPU时间片，进程轮流占用CPU，当时间片用完时，释放CPU。



- 保证每个进程有均等机会运行，同时改善了短进程的响应时间，提高了系统吞吐量。
- 常用的调度算法，特别适合分时交互环境。

进程调度的时机

② 当时间片用完（剥夺调度）

可能事件：

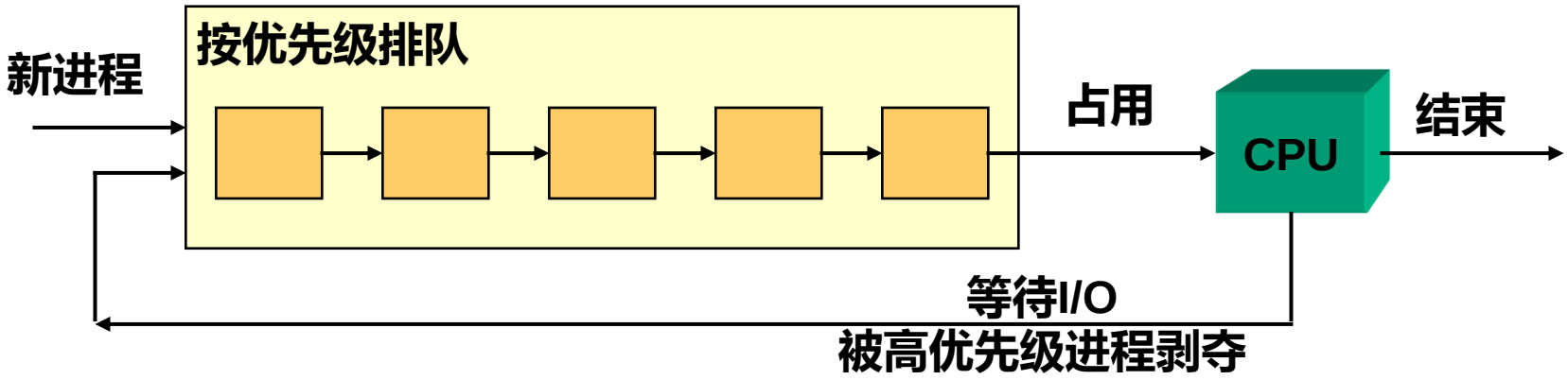
- 时钟中断（do_timer ）

目录

三. 优先级调度算法

优先级可静态也可动态

- 赋给每个进程一个优先级，优先级最高的进程占用CPU。（非剥夺或剥夺）



- 调度灵活，适应多种调度需求；
- 低优先级进程可能被“饿死”；
- 优先级剥夺调度导致调度开销增大。

进程调度的时机

③ 更高优先级的就绪进程出现时（剥夺调度）

可能事件：

- 中断

目录

Linux 0.11 的进程调度算法

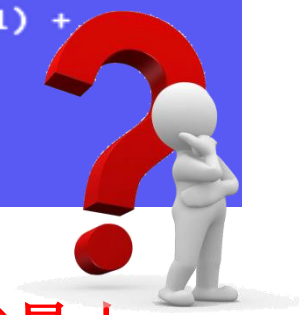
```
(gdb) list 105, 143
105     void schedule(void)
106     {
107             int i,next,c;
108             struct task_struct ** p;
109
110     /* check alarm, wake up any interruptible tasks that have got a signal */
111
112             for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p) {
                                if ((*p)->alarm && (*p)->alarm < jiffies) {
                                                (*p)->signal |= (1<<(SIGALRM-1));
                                                (*p)->alarm = 0;
                                        }
                                if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                                (*p)->state==TASK_INTERRUPTIBLE)
                                        (*p)->state=TASK_RUNNING;
                        }

    /* this is the scheduler proper: */

                while (1) {
                        c = -1;
                        next = 0;
                        i = NR_TASKS;
                        p = &task[NR_TASKS];
                        while (--i) {
                                if (!*--p)
                                        continue;
                                if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                                        c = (*p)->counter, next = i;
                        }
                        if (c) break;
                        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                                if (*p)
                                        (*p)->counter = ((*p)->counter >> 1) +
                                                        (*p)->priority;
                }
                switch_to(next);
        }
(gdb)
```

可中断睡眠状态

进程调度

```
123     /* this is the scheduler proper: */
124
125             while (1) {
126                     c = -1;
127                     next = 0;
128                     i = NR_TASKS;
129                     p = &task[NR_TASKS];
130                     while (--i) {
131                             if (!*--p)
132                                     continue;
133                             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
134                                     c = (*p)->counter, next = i;
135                     }
136                     if (c) break;
137                     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
138                             if (*p)
139                                     (*p)->counter = ((*p)->counter >> 1) +
140                                                     (*p)->priority;
141             }
142             switch_to(next);
143     }
(gdb)
```

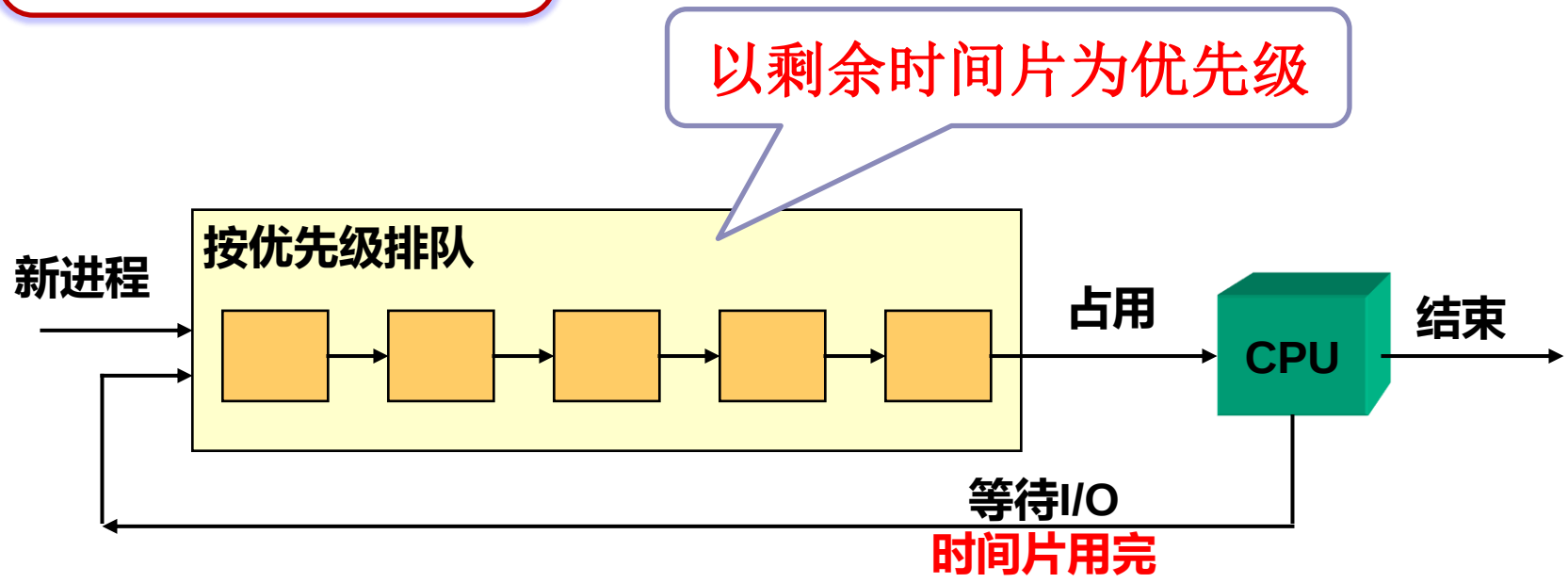
优先级与时间片相结合的调度算法



- 选择所有就绪进程（0号进程除外）中**counter值最大**的那一个；如果没有一个就绪进程，就选择**0号进程**；如果所有就绪进程的**counter值都为0**，则给所有进程重新添加时间片，然后重新选择。

四. Linux 0.11的进程调度算法

优先级与时间片相结合的调度算法



四. Linux 0.11的进程调度算法

- 演示5:

第1、2次进程调度

  - 位置：schedule
  - 数据：调用栈

目录

五. 调度时机之延迟调度

④ 系统调用返回前

可能事件：

- 系统调用返回（ret_from_sys_call ）

（如果事件发生时不适合立刻调度）

五. 调度时机之延迟调度

- 演示(续):

第3次进程调度

  - 位置：schedule
  - 数据：调用栈

五. 调度时机之延迟调度

- 执行一个pause系统调用可能引发几次调度？

五. 调度时机之延迟调度

- 执行一个pause系统调用可能引发几次调度？

```
pause()
   int 0x80
      system_call()
         sys_pause()
            schedule()    //主动放弃
         [schedule()]    //可能引发延迟调度
         ret_from_sys_call()
            iret
```

六. 独学&讨论

- 学习《Linux内核完全剖析》：
  - P303：schedule()
  - P288：_system_call()
- 问题：
  - 哪些情况会引发延迟调度？

小结

一. 先进先出调度算法（FIFO）

二. 时间片轮转调度算法

三. 优先级调度算法

四. Linux 0.11的进程调度算法

五. 调度时机之延迟调度