2021级期末考试复习版本

《操作系统原理》

第6章 处理机调度（进程调度）

教师：邹德清，李珍，苏曙光

华中科技大学网安学院

2023年10月-2024年01月

第6章 进程调度

- 主要内容
  - 进程调度概念
  - 进程调度算法
  - Linux进程调度
- 教学重点
  - 典型调度算法
  - Linux调度机制

6.1 进程调度概念

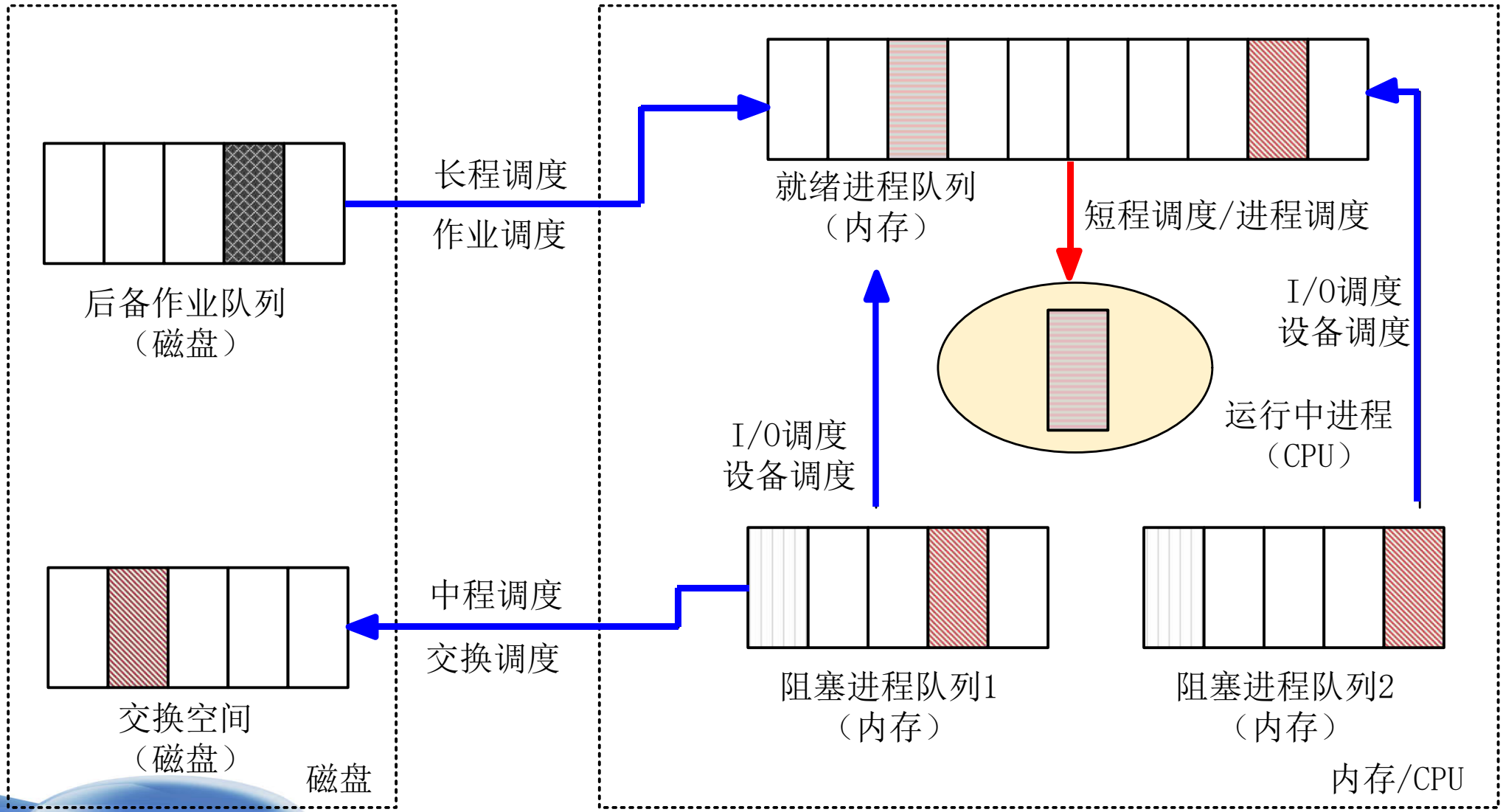
调度定义与分类

- 调度定义
  - schedule

队列

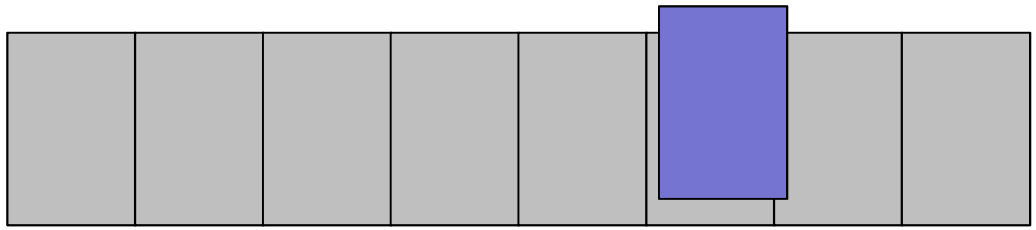
  - 在队列中按某种策略选择最合适的对象(执行相应操作)。
- 调度分类
  - 长程调度（宏观调度/作业调度）【作业：磁盘→内存】
  - 中程调度（交换调度）【进程：就绪(内存)→交换空间】
  - 短程调度（进程调度）【进程：就绪(内存)→CPU】
  - I/O调度（设备调度）【进程：阻塞(设备)→就绪】

调度定义与分类


长程调度
作业调度
后备作业队列
（磁盘）
就绪进程队列
（内存）
短程调度/进程调度
I/O调度
设备调度
运行中进程
（CPU）
I/O调度
设备调度
中程调度
交换调度
交换空间
（磁盘）
磁盘
阻塞进程队列1
（内存）
阻塞进程队列2
（内存）
内存/CPU

调度定义与分类

- 进程调度/短程调度
  - 在合适的时候以一定策略选择一个就绪进程运行.
  - 调度的策略？
    - 调度的目标？
    - 调度的时机？



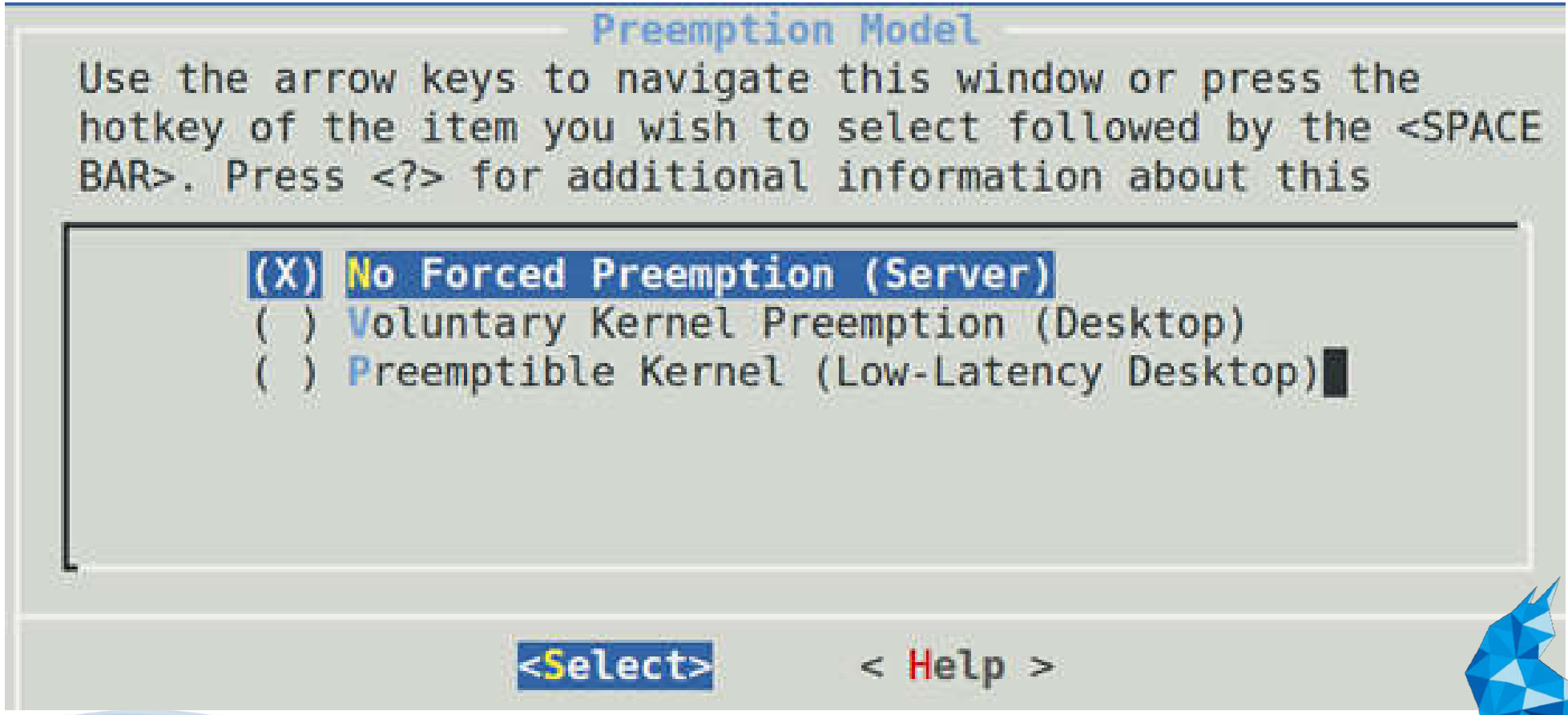
就绪进程队列

调度定义与分类

- 进程调度的目标
  - 1.响应速度尽可能快
  - 2.进程处理的时间尽可能短
  - 3.系统吞吐量尽可能大
  - 4.资源利用率尽可能高
  - 5.对所有进程要公平
  - 6.避免饥饿
  - 7.避免死锁
  - 上述部分原则之间存在自相矛盾！

调度定义与分类

- 进程调度的目标
  - 例：Linux内核配置 > Preemption Model



KYLIN
银河麒麟

调度定义与分类

- 进程调度的目标（两个量化的衡量指标）
  - 周转时间/平均周转时间
  - 带权周转时间/平均带权周转时间

调度定义与分类

- 周转时间
  - 进程(或作业)提交给计算机到完成所花费的时间

    周转时间 t = 完成时间 t_c – 提交时间 t_s
    - t_s——进程的提交时间（Start）
    - t_c——进程的完成时间（Complete ）
  - 意义
    - 说明进程在系统中停留时间的长短。
- 平均周转时间
  - $t = （t_1 + t_2 + \ldots + t_n）/ n$
  - 所有进程的周转时间的平均
  - 平均周转时间越短意味着：平均停留时间越短，系统吞吐量越大，资源利用率越高。

调度定义与分类

- 带权周转时间w
  - w = 周转时间t ÷ 进程运行时间(进程大小) t_r

    $= t / t_r$
  - t：进程的周转时间
  - t_r：进程的运行时间（run）
  - 意义：进程在系统中的相对停留时间。
- 平均带权周转时间
  - $w = （w_1 + w_2 + \ldots + w_n） \div n$
  - 所有进程的带权周转时间的平均

6.2 进程调度算法

进程调度算法

- 典型调度算法
  - 1. 先来先服务调度（First Come First Serve）
  - 2. 短作业优先调度算法（Short Job First）
  - 3. 响应比高者优先调度算法
  - 4. 优先数调度算法
  - 5. 循环轮转调度法（ROUND-ROBIN）
  - 6. 可变时间片轮转调度法
  - 7. 多重时间片循环调度法

1. 先来先服务调度（First Come First Serve）

- 算法
  - 按照作业进入系统的时间先后次序来挑选作业。先进入系统的作业优先被运行。
- 特点
  - 只考虑作业等候时间，不考虑作业大小(运行时间)。
  - 晚来的作业会等待较长时间
  - 不利于晚来但是很短的作业。

2. 短作业优先调度算法（Short Job First）

- 算法
  - 参考运行时间，选取时间最短的作业投入运行。
- 特点/缺点
  - 忽视了作业等待时间
  - 早来的长作业会长时间等待(资源“饥饿”)

3. 响应比高者优先调度算法

- 响应比定义
  - ■作业的响应时间和与运行时间的比值
  - □响应比 = 响应时间/运行时间
    = （等待时间 + 运行时间）/运行时间
    = 1 + 等待时间 / 运行时间
- 算法 = 加权周转时间（即时的）
  - ●调度作业时计算作业列表中每个作业的响应比，选择响应比最高的作业优先投入运行。

3. 响应比高者优先调度算法

- 特点
  - 响应比 = 1 + 等待时间 / 运行时间
  - 有利于短作业
  - 有利于等候已久的作业
  - 兼顾长作业
- 应用
  - 每次调度时重新计算和比较剩余作业的响应比

4. 优先数调度算法

- 算法
  - 根据进程优先数，把CPU分配给最高的进程。
  - 进程优先数 = 静态优先数 +动态优先数
- 静态优先数
  - 进程创建时确定，在整个进程运行期间不再改变。
- 动态优先数
  - 动态优先数在进程运行期间可以改变。

4. 优先数调度算法

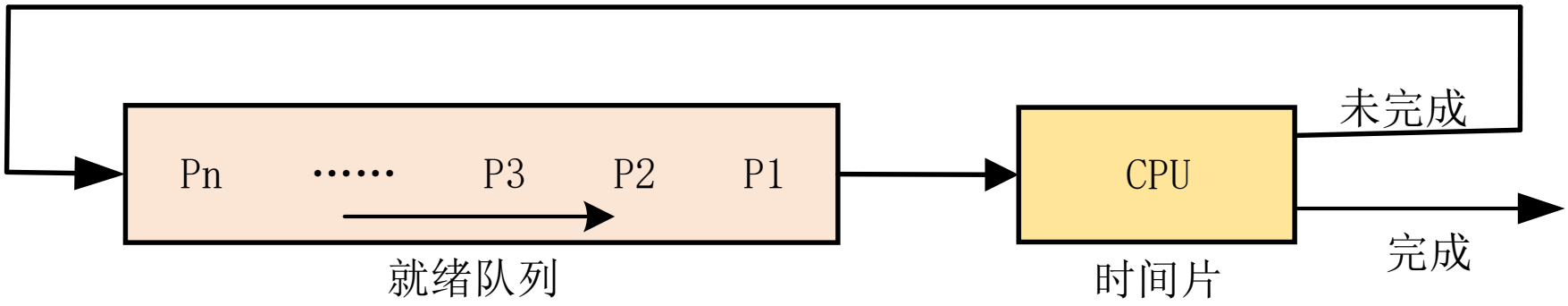
- 静态优先数的确定
  - ✓ 基于进程所需的资源多少
  - ✓ 基于程序运行时间的长短
  - ✓ 基于进程的类型[IO/CPU，前台/后台，核心/用户]
- 动态优先数的确定
  - ✓ 当进程使用CPU超过一定时长时；
  - ✓ 当进程等待时间超过一定时长时；
  - ✓ 当进行I/O操作后；

4. 优先数调度算法

- **LINUX进程优先级（task_struct成员变量）**
  - ■priority/静态优先数
    - ◆$ nice # ↙
      - □priority = priority – #
      - □ #:[-20～19]
        - ▲普通用户：自己进程，[0, 19]
        - ▲root用户：任何进程，[-20, 19]
  - ■counter/动态优先数
    - ◆动态优先数（实际用于比较的优先数，结果）
    - ◆counter初值 = priority
    - ◆可变化，可调整

5. 循环轮转调度法（ROUND-ROBIN）

- 算法
  - 把所有就绪进程按先进先出的原则排成队列。新来进程加到队列末尾。
  - 进程以时间片q为单位轮流使用CPU。刚刚运行了一个时间片的进程排到队列末尾，等候下一轮调度。
  - 队列逻辑上是环形的。



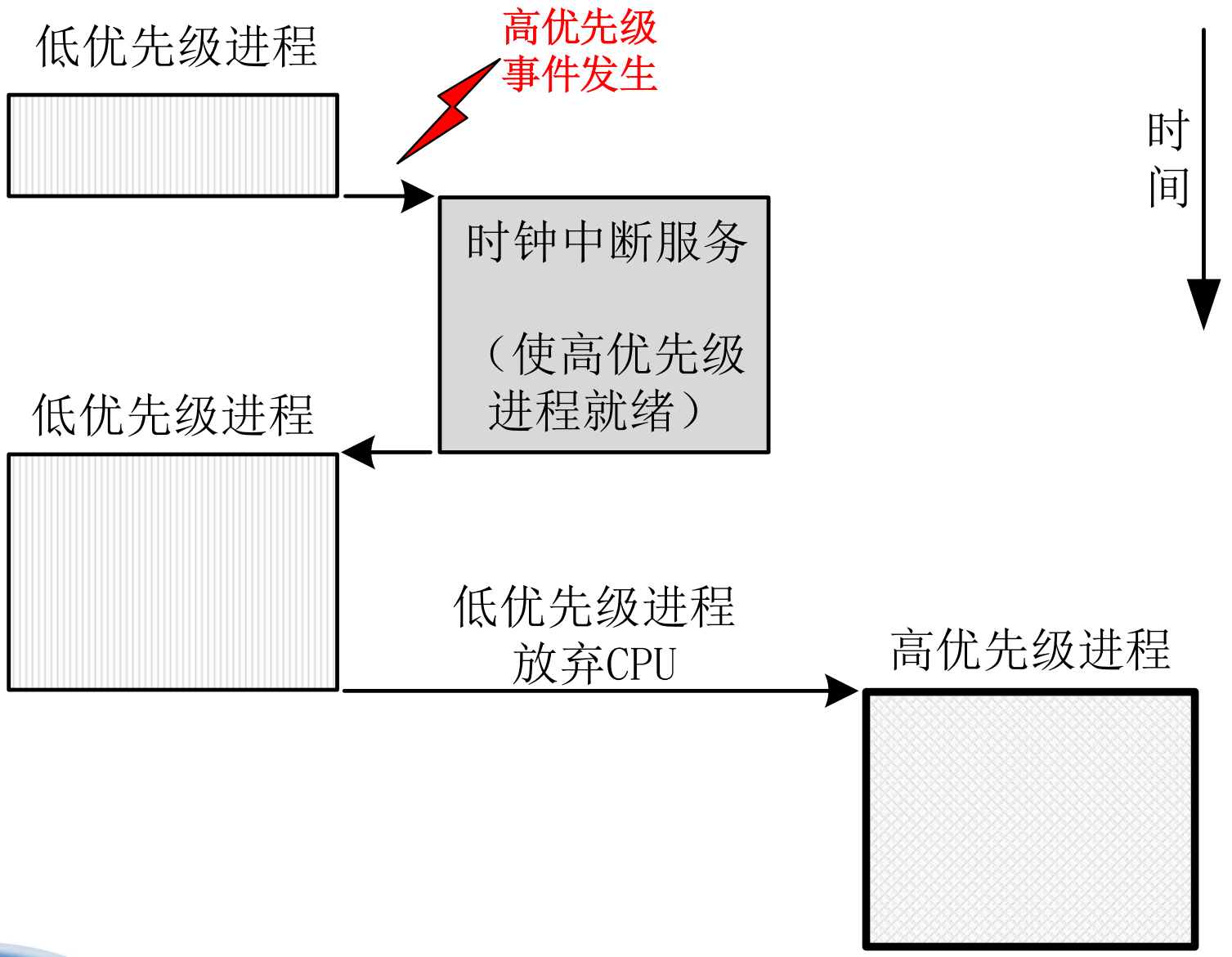
5. 循环轮转调度法（ROUND-ROBIN）

- 优点
  - 公平性：每个就绪进程有平等机会获得CPU
  - 交互性：每个进程等待(N-1)* q的时间就可以重新获得CPU
- 时间片q的大小
  - 如果q太大
    - 交互性差
      - 甚至退化为FCFS调度算法。
  - 如果q太小
    - 进程切换频繁，系统开销增加。
- 改进
  - 时间片的大小可变(可变时间片轮转调度法)
  - 组织多个就绪队列（多重时间片循环轮转）

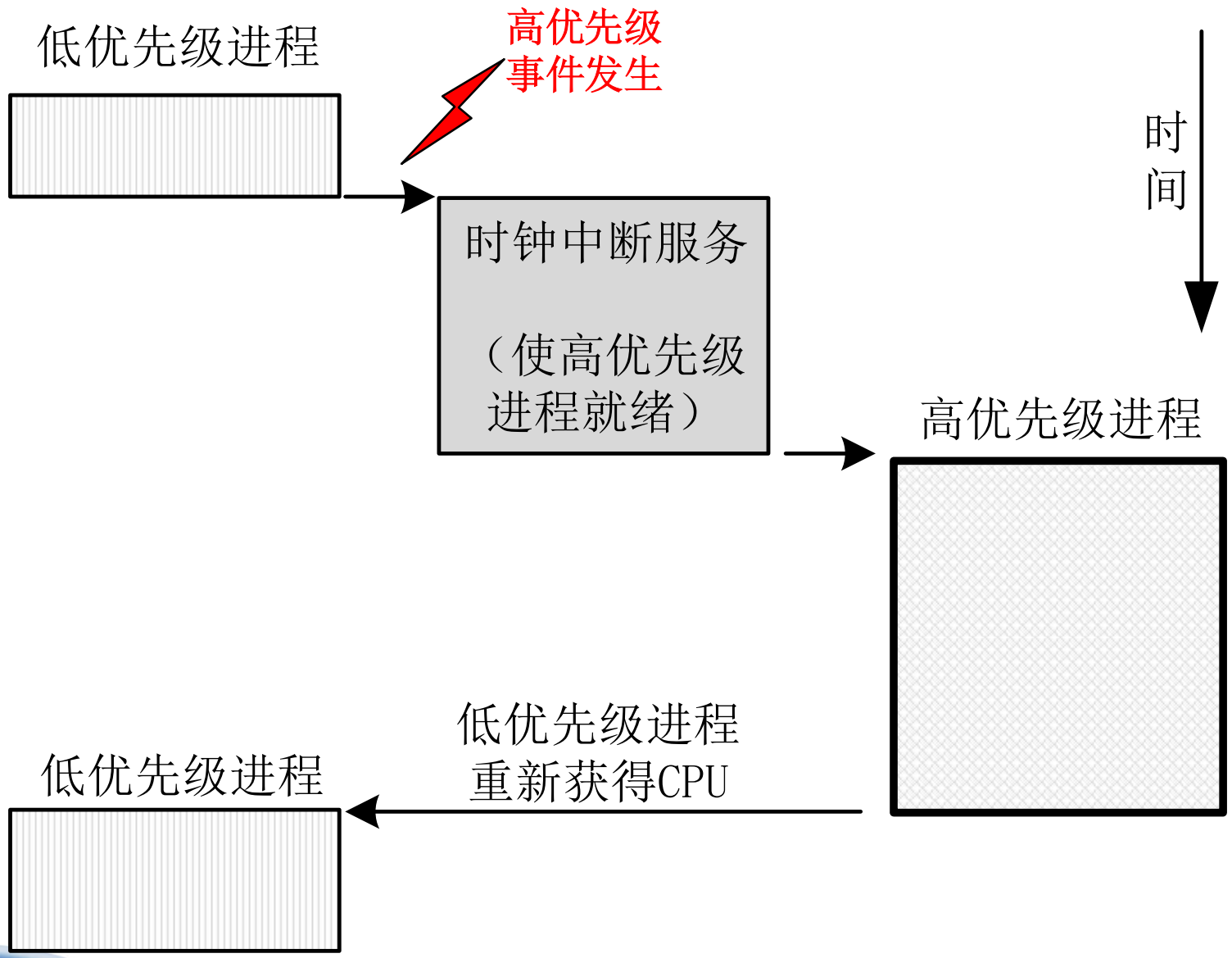
调度方式

- 定义
  - 当一进程正在CPU上运行时，若有更高优先级的进程进入就绪，系统如何对待新进程（分配CPU）？
- 非抢占方式
  - 让正在运行的进程继续运行，直到该进程完成或发生某事件而进入“完成”或“阻塞”状态时，才把CPU分配给新来的更高优先级的进程。
- 抢占方式
  - 让正在运行的进程立即暂停，立即把CPU分配给新来的优先级更高的进程。

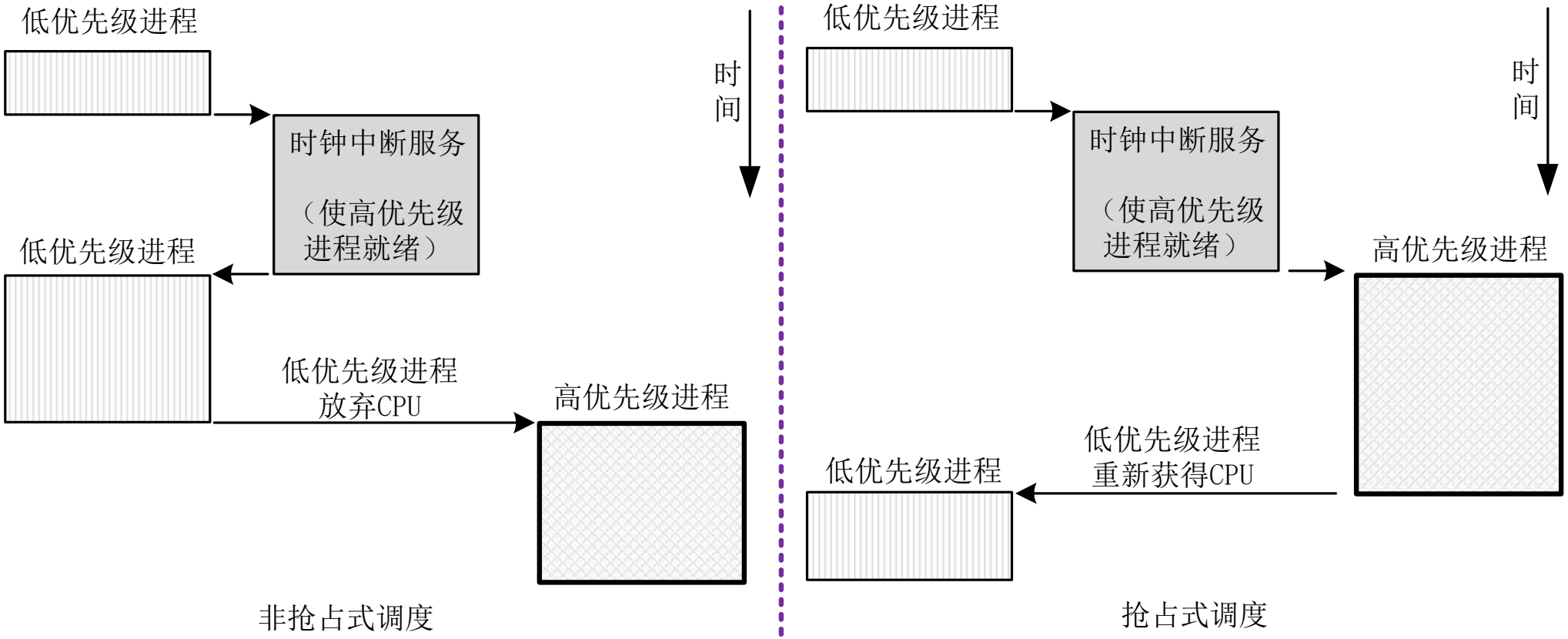
调度方式 ->非抢占方式


低优先级进程
高优先级
事件发生
时
间
时钟中断服务
（使高优先级
进程就绪）
低优先级进程
低优先级进程
放弃CPU
高优先级进程

调度方式 ->抢占方式


低优先级进程
高优先级
事件发生
时
间
时钟中断服务
（使高优先级
进程就绪）
高优先级进程
低优先级进程
重新获得CPU
低优先级进程

非抢占方式与抢占方式的对比



6.3 Linux进程调度

Linux进程调度

- 基本特点
  - 1）基于优先级调度；
  - 2）支持普通进程，也支持实时进程；
    - 3）实时进程优先于普通进程；
  - 4）普通进程公平使用CPU时间。

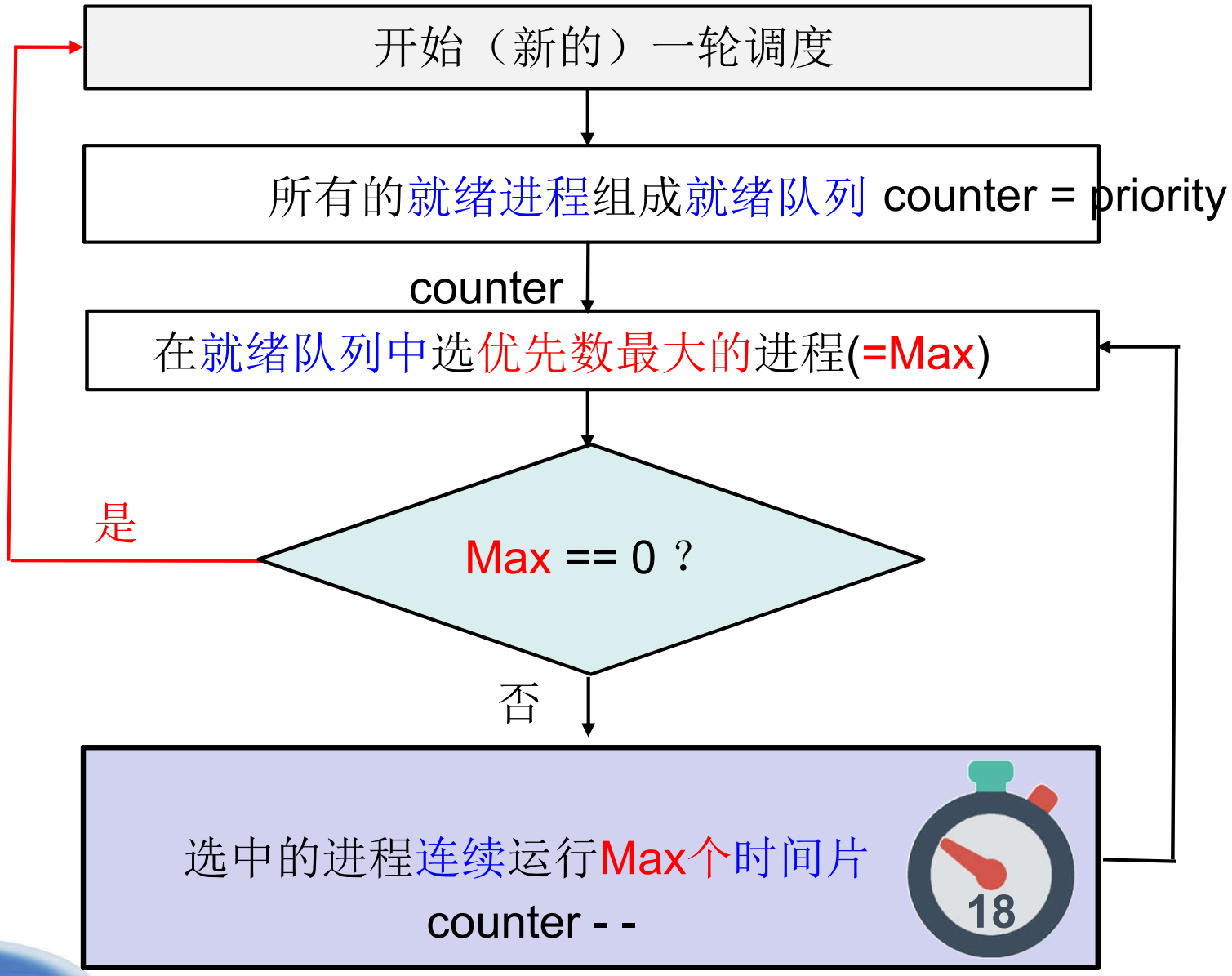
Linux进程调度

- **LINUX进程优先级（task_struct成员变量）**
  - priority/静态优先数
    - $ nice # ↙
    - priority = priority – #
    - #:[-20～19]
      - 普通用户：自己进程，[0, 19]
      - root用户：任何进程，[-20, 19]
  - counter/动态优先数
    - 用于实际比较的优先数
    - 初值 = priority
    - 可变化，可调整

LINUX进程优先级（task_struct成员变量）

- counter
  - 进程在当前一轮调度中还能连续运行的时间片数量
    - counter越大，优先级越高，可获得越多CPU时间
  - 新一轮调度开始时
    - counter = priority
  - 时钟中断服务程序
    - counter - -
  - 特定情形
    - counter = counter + △
  - 所有进程的counter都减到0后
    - 重新开始新一轮调度

LINUX进程调度基本原理（优先数调度）



LINUX进程调度基本原理（优先数调度）

- task_struct
  - rt_priority
    - 实时进程特有的优先级：rt_priority+1000
  - policy
    - 进程的调度策略,用来区分实时进程和普通进程
    - SCHED_OTHER(0)||SCHED_FIFO(1)||SCHED_RR(2)

调度策略（task_struct）

- **task_struct** -> policy指明进程使用何种调度策略。

```
/*
 * Scheduling policies
 */
#define SCHED_OTHER     0
#define SCHED_FIFO      1
#define SCHED_RR        2

/*
 * This is an additional bit set when we want to
 * yield the CPU for one re-schedule..
 */
#define SCHED_YIELD     0x10
```

普通的分时进程

先进先出的实时进程

基于优先级轮转的实时进程

当一个进程自动放弃运行时设置

KYLIN
银河麒麟

调度函数（ schedule函数）

- **schedule( )函数**
  - 在可运行队列中查找最高优先数进程并把CPU切换给它。
- 调用时机
  - 直接调度
    - 时钟中断 | do_timer( )
    - 当资源无法满足时阻塞进程
    - sleep_on( )
  - 间接调度/松散调度
    - 进程从内核态返回到用户态前
      - 检查need_resched == 1 ?

时钟中断：do_timer()函数

```
1  void do_timer(long cpl)
2  {
3      extern int beepcount;
4      extern void sysbeepstop(void);
5      if (beepcount)
6          if (!--beepcount)
7              sysbeepstop();
8
9      if (cpl)
10         current->utime++;
11     else
12         current->stime++;
13
```

时钟中断：do_timer()函数

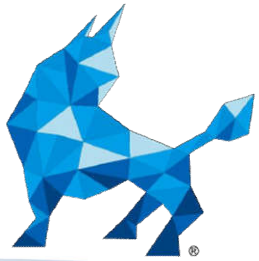
```
14      if (next_timer) {
15          next_timer->jiffies--;
16          while (next_timer && next_timer->jiffies <= 0) {
17              void (*fn)(void);
18
19              fn = next_timer->fn;
20              next_timer->fn = NULL;
21              next_timer = next_timer->next;
22              (fn)();
23          }
24      }
25      if (current_DOR & 0xf0)
26          do_floppy_timer();
27      if ((--current->counter)>0) return;
28      current->counter=0;
29      if (!cpl) return;
30      schedule();
31  }
```

调度函数（ schedule函数）

- 工作流程
  - 第一步：选择进程
    - 扫描可运行队列，选择一个合适进程
    - 优先级最高 | counter最大 | goodness( )
  - 第二步：切换进程
    - 当前进程 → 新进程
      - 进程的上下文切换

调度函数（ schedule函数）

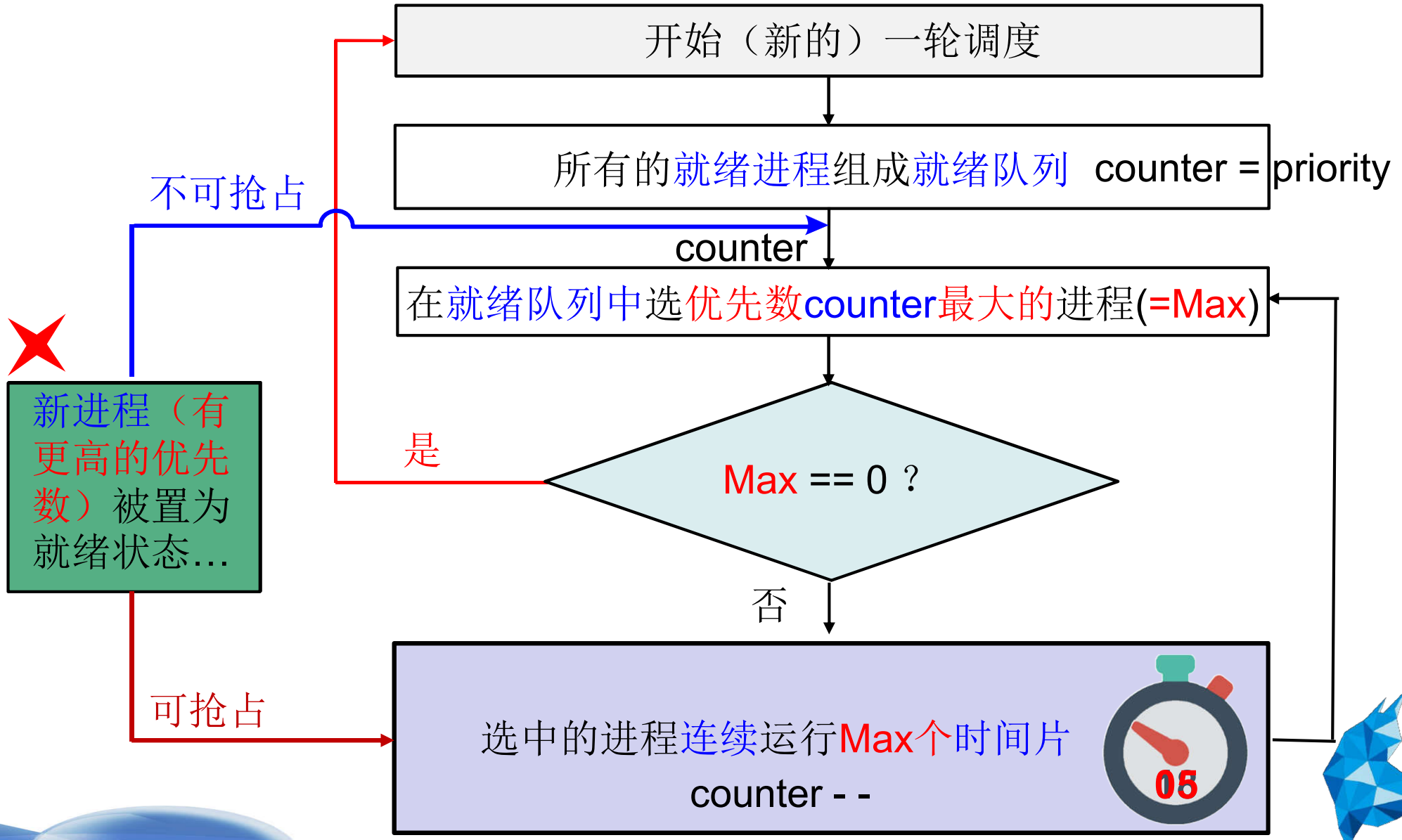
```
1  void schedule(void)
2  {
3      int i,next,c;
4      struct task_struct ** p;
5      for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
6          if (*p) {
7              if ((*p)->alarm && (*p)->alarm < jiffies) {
8                      (*p)->signal |= (1<<(SIGALRM-1));
9                      (*p)->alarm = 0;
10                 }
11             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
12             (*p)->state==TASK_INTERRUPTIBLE)
13                 (*p)->state=TASK_RUNNING;
14         }
15     while (1) {
16         c = -1;
17         next = 0;
18         i = NR_TASKS;
19         p = &task[NR_TASKS];
20         while (--i) {
21             if (!*--p)
22                 continue;
23             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
24                 c = (*p)->counter, next = i;
25         }
26         if (c) break;
27         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
28             if (*p)
29                 (*p)->counter = ((*p)->counter >> 1) +
30                         (*p)->priority;
31     }
32     switch_to(next);
33 }
```



第二步：切换进程

```
1  #define switch_to(n) {\
2  struct {long a,b;} __tmp; \
3  __asm__("cmpl %%ecx,_current\n\t" \
4      "je 1f\n\t" \
5      "movw %%dx,%1\n\t" \
6      "xchgl %%ecx,_current\n\t" \
7      "ljmp %0\n\t" \
8      "cmpl %%ecx,_last_task_used_math\n\t" \
9      "jne 1f\n\t" \
10     "clts\n" \
11     "1:" \
12     ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
13     "d" (_TSS(n)),"c" ((long) task[n])); \
14 }
```

调度的方式：不可抢占/可抢占



KYLIN
银河麒麟