操作系统

第3章 CPU调度

朱小军

南京航空航天大学计算机科学与技术学院

2025年春

目录

什么是“调度”?

- 在计算机领域的含义，大部分对应英文schedule
- schedule的其他翻译（作为名词）
  - 时间安排表
    - 进度安排：ahead of/behind schedule
  - train schedule：列车时刻表
  - class schedule：课表
- 作为动词的schedule，就是生成名词schedule的动作
  - 制作一份XXX的时间安排表
  - 制作一份CPU的时间安排表（CPU调度）
  - TW翻译为“排程”

CPU调度

- 调度的层次
  - 高级调度（作业调度）：外存中的哪些作业进入内存？
  - 中级调度（内存调度）：暂时不用的进程调至外存（挂起）
  - 低级调度（进程调度/线程调度）：从就绪队列选择一个进程分配处理器，进入运行态[本课程重点]
- 低级调度的对象
  - 进程（不支持内核级线程的OS）
  - 线程（支持内核级线程的OS）
  - 在此页之后，CPU调度、进程调度、线程调度等同

究竟选择哪个进程运行？

- 假设
  - CPU只有一个
  - 处于就绪队列的进程有三个

进程编号	所需时间
P_1	24
P_2	3
P_3	3

目标是什么？

调度的目标

- CPU利用率
  - CPU有效工作时间/任务完成的总时间
- 吞吐量
  - 单位时间内完成进程的数量
- 平均周转时间
  - 周转时间：从进程提交到进程结束
- 平均带权周转时间
  - 带权周转时间：周转时间除以服务时间
- 响应时间
  - 提交请求到产生第1响应的时间
- 可能同时优化所有目标吗？

目录

调度器的实现有三个关键点

- 选
  - 选谁？就绪队列中选择1个（对应调度算法）
  - 万一就绪队列没有进程怎么办？
    - 闲逛进程（空闲进程，Idle Process，一般PID为0），处于就绪态或运行态（内核态运行，不会切入用户态），占内存低、执行空指令或节能指令让CPU进入低功耗状态
- 切
  - 上下文切换，让CPU去执行被选择的进程
  - 切换逻辑：调度器->进程1->调度器->进程2
- 回
  - 什么时候回到调度器？调度时机

“回”：调度时机

- 根据是否“当前执行进程主动发起”，分为
  - 非抢占式调度，又称协作调度：主动放弃
  - 抢占式调度：已分配给某进程、转而分配给其他进程

- 非抢占式调度（主动放弃）的调度时机
  - 自愿放弃CPU：如通过sched_yield系统调用，会进入就绪态
  - 进程主动退出(exit)
  - 发起阻塞式系统调用（如read, write, wait, 等）

- 抢占式调度的典型调度时机
  - 时钟中断处理中切换，内核从进程内核栈切换至调度程序
  - 新的高优先级进程进入就绪队列（高优先级新创建、从阻塞态进入就绪态）

案例剖析：xv6的“选”“切”“回”

调度程序在内核中的位置（xv6）

- 操作系统主程序

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
      …
kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}


static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

xv6的scheduler函数

```
void
scheduler(void)
{
  struct proc *p;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

选

选择PCB列表中下一个处于就绪状态的进程

切

切换到被选择进程的内核上下文

切换至被选择进程的页表

回？

切回来时，更换内核页表

xv6的scheduler函数

```
void
scheduler(void)
{
  struct proc *p;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

选

选择PCB列表中下一个处于就绪状态的进程

切

切换到被选择进程的内核上下文

切换至被选择进程的页表

回？

切回来时，更换内核页表

- GTD段表切换：TSS中设置进程的内核栈（软、硬中断时要用）
- 页表切换：进程映像更改（内核部分不变，用户空间存在）

```
void
scheduler(void)
{

...
    proc = p;
    switchuvm(p);

    p->state = RUNNING;

    swtch(&cpu->scheduler, proc->context);



    switchkvm();
  ...

}
```

- scheduler()的CPU现场压入内核栈，构成context，让cpu->scheduler存储context的地址（恰好是esp寄存器的值）
- 用proc->context恢复现场，切入进程的内核栈（和内核上下文）
- 从进程的内核线程恢复CPU现场，退出到用户态执行

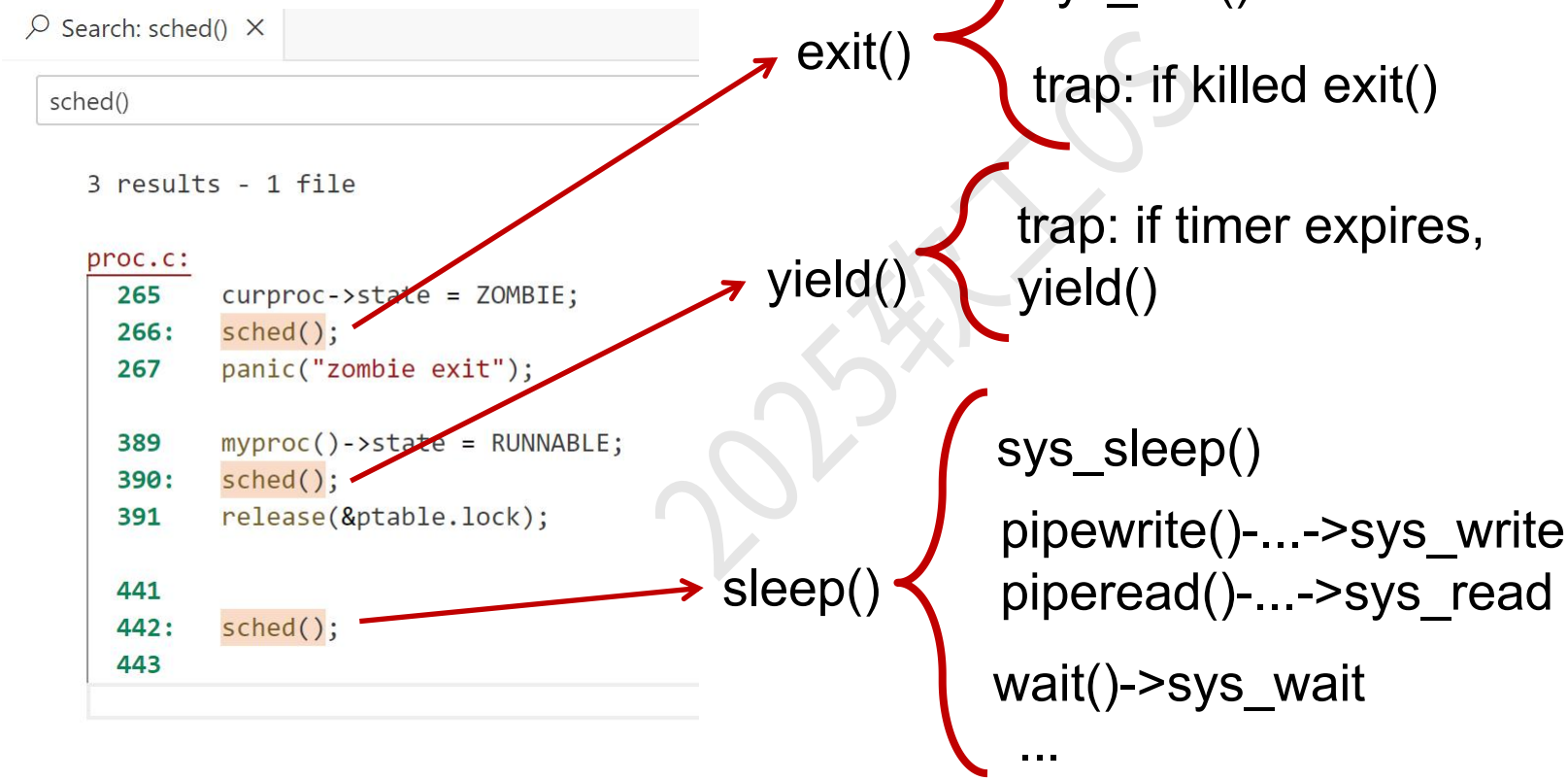
调度被触发

- 触发调度（时钟中断？exit？sleep？），进程陷入内核态，用户态上下文信息保存在进程的内核栈
- 进程在内核态调用sched()，sched调用swtch(&proc->context, cpu->scheduler)

- 进程CPU现场压入进程内核栈，构成context，地址（esp的值）存储在proc->context
- 用cpu->scheduler恢复现场，切入内核栈（和内核上下文）
- 从内核栈恢复CPU现场

xv6的调度时机

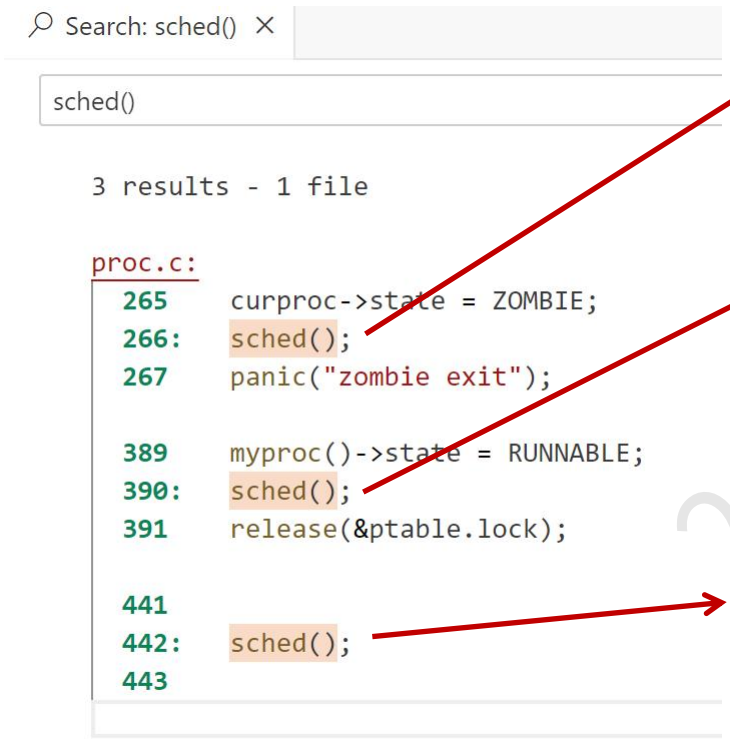
- 检查调度时机，在何处切换回scheduler？即，在何处调用sched（）？



- 哪些是自愿的？怎么定义“自愿”？

xv6的调度时机

- 检查调度时机，在何处切换回scheduler？即，在何处调用sched（）？



exit()
- sys_exit()　自愿
- trap: if killed exit()　不自愿

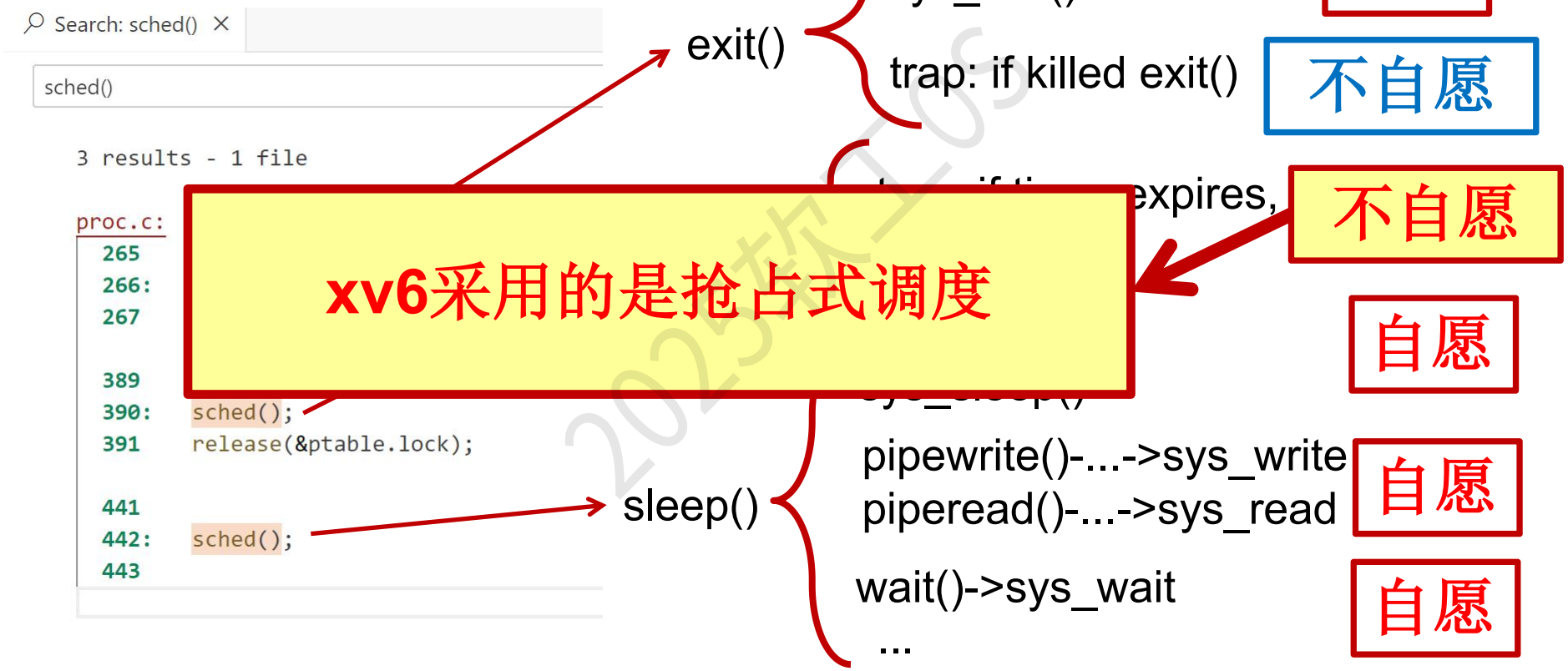
yield()
- trap: if timer expires, yield()　不自愿

sleep()
- sys_sleep()　自愿
- pipewrite()-...->sys_write
piperead()-...->sys_read　自愿
- wait()->sys_wait　自愿
- ...

- 怎么定义“自愿”？自己发出系统调用？

xv6的调度时机

- 检查调度时机，在何处切换回scheduler？即，在何处调用sched（）？



- 只要有非自愿的调度时机，即为抢占式调度
  - 信号处理除外

xv6的调度时机可以继续扩充

- 每次就绪队列发生改变时均可以触发
  - 在fork中触发？
  - 在exec中触发？
  - 在wakeup中触发？
  - 。。。
  - 要不每次可能触发时，设置一个全局变量need_resched，当从内核空间返回用户空间时，检查此变量的值？（Linux的做法）

目录

- CPU调度的对象和目标
- CPU调度的实现
- CPU调度算法

六种调度算法

先到先服务

先到先服务

进程	时间
P_1	24
P_2	3
P_3	3

- 若进程到达顺序为：P_1, P_2, P_3，则甘特图为

- 平均周转时间？

先到先服务

进程	时间
P_1	24
P_2	3
P_3	3

- 若进程到达次序为 P_2, P_3, P_1

- 平均周转时间

先到先服务调度算法的优劣

- 非抢占式调度算法
  - 没有抢占式版本
- 相对公平
  - 没有饥饿
- 缺点
  - 长短任务混合下，对短任务不友好
  - 计算密级和IO密级混合下，对IO密级任务不友好
- 问：假如要在xv6中实现先到先服务，应该怎么做？
  - 什么叫“先到”？创建时间？

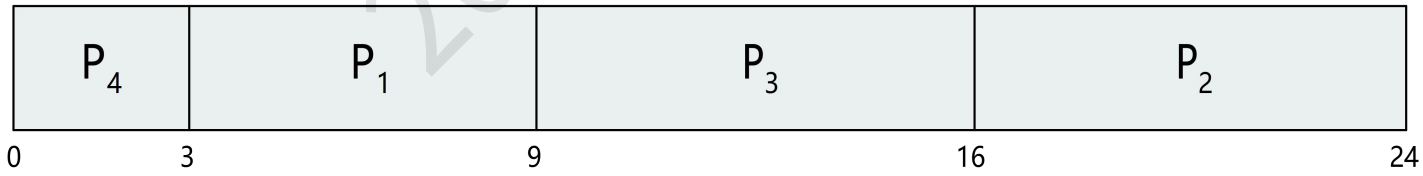
短作业优先调度算法

- 挑选CPU耗时最短的进程
- 平均周转时间最小的调度算法（抢占式版本）
  - 难点在于如何得知CPU耗时
  - 可以由用户提供，或者采用指数平均进行估计
- 指数平均
  - $est_{n+1}=a*t_n+(1-a)*est_n$
  - 距现在时间越久，权重越小，权重指数递减
  - 常见于网络通信中RTT时延估计
- 可以是抢占的或者是非抢占的
  - 抢占的又称为最短剩余时间优先调度

短作业优先练习

进程	时间
P_1	6
P_2	8
P_3	7
P_4	3

- 调度图

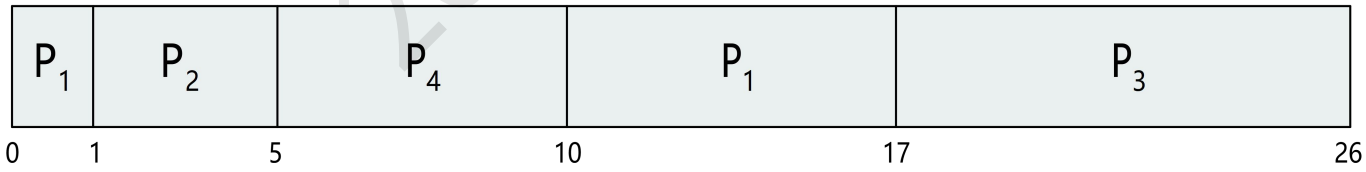


- 平均周转时间为？

最短剩余时间优先

进程	到达时刻	时间
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- 调度图



- 平均周转时间？

课堂测验

进程	到达时刻	时间
P_1	0	6
P_2	2	8
P_3	4	7
P_4	5	3

1. 画出抢占式短作业优先算法的调度图
2. 计算平均带权周转时间

短作业优先调度算法的优劣

- 优势
  - 最短周转时间
- 劣势
  - 需要估计运行时间，此外，
  - 非抢占式：性能依赖到达时间（晚到的短任务不利）
  - 抢占式：长任务饥饿
- 思考：如何在xv6中实现短作业优先调度算法？
  - 如何估算运行时间？
  - 如何选择最短任务？

优先级调度算法

- 校车排队
- 军人优先
- 妇女儿童优先
- VIP优先

…

优先级调度算法

- 每个进程有一个优先级：静态、动态
- 选择优先级最高的进程执行
  - 抢占式的
  - 非抢占式的
  - 短作业优先调度算法是一种优先级调度算法
- 特例：高响应比优先
  - FCFS，谁先到谁优先　　优先级∝ 等待时间
  - SJF，谁短谁优先　　优先级∝ 1/执行时间
  - 高响应比：（等待+执行）/执行【这不是带权周转时间吗？】

优先级调度算法例子

进程	时间	优先级
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- 能否画出调度图？
- 非抢占式版和抢占式版本有何不同？

优先级调度面临的问题

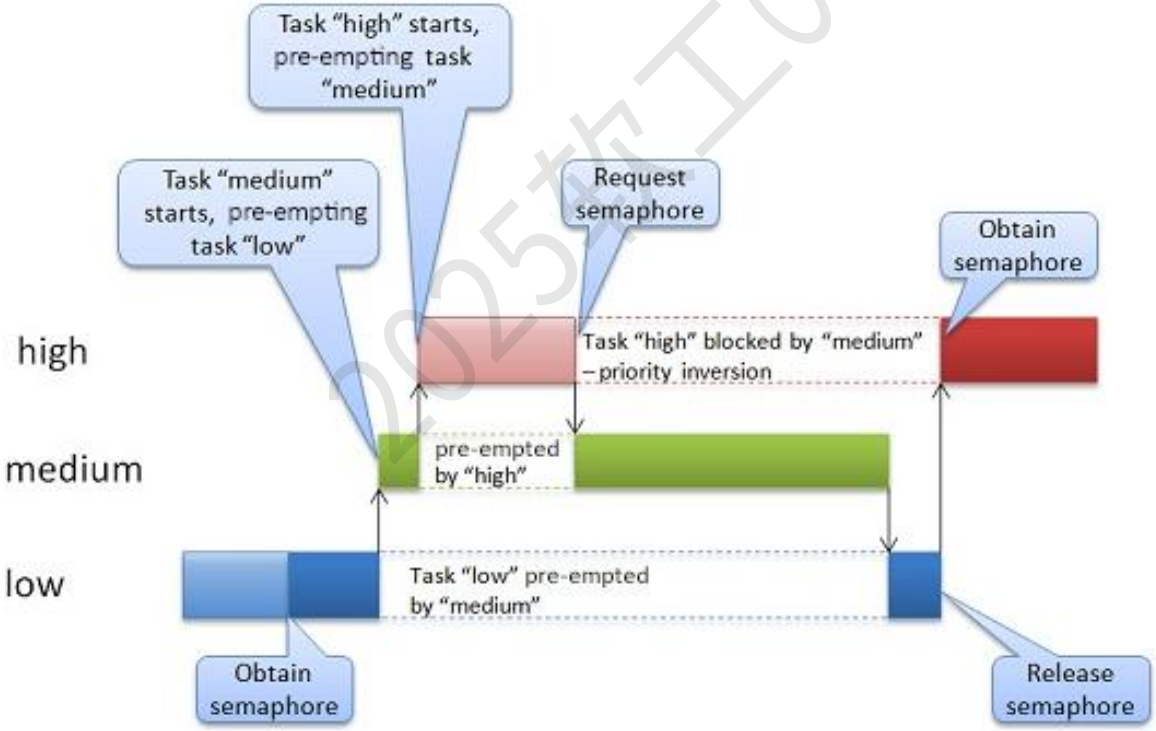
- 如何设置优先级？
- 饥饿问题？
- 如何在xv6中实现优先级调度？
  - 采用何种数据结构？
  - 如何实现这种数据结构？
- 优先级倒置问题

优先级倒置问题

Mars Pathfinder的事故

https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft

- 高优先级被低优先级间接抢占了



轮转调度算法

适用的借阅规则

规则名称	适用馆藏地	图书流通类型	最大借阅册数	借期	预约	续借	
复印外借	明故宫外文阅览室(书),明故宫中文阅览室(书),明故宫外文阅览室(刊),明故宫外文期刊库,明故宫民用航空文献阅览室,南航特藏室(文库),南航特藏室(会议),明故宫中文阅览室(刊),明故宫中文期刊库	所有流通类型	3	10	不允许	不允许	查看借阅规则
rule4	将军路外借书库,将军路外借书库(外文),将军路储备书库,将军路外借书库(新书),将军路密集书库(一楼),将军路社会科学阅览室 I (四楼南),将军路社会科学阅览室II (四楼北),将军路社会科学阅览室III(五楼南),将军路外文阅览室(外借图书)(六楼),将军路自然科学阅览室 I (二楼南),将军路自然科学阅览室II (三楼南),待入密集书库	所有流通类型	24	60	允许	允许	查看借阅规则
rule11(老馆改造)	明故宫中文借阅室(文学),明故宫中文借阅室(科技),明故宫中文借阅室(社科),明故宫中文借阅室(新书),明故宫外文借阅室,明故宫	所有流通类型	44	200	允许	不允许	查看借阅规则

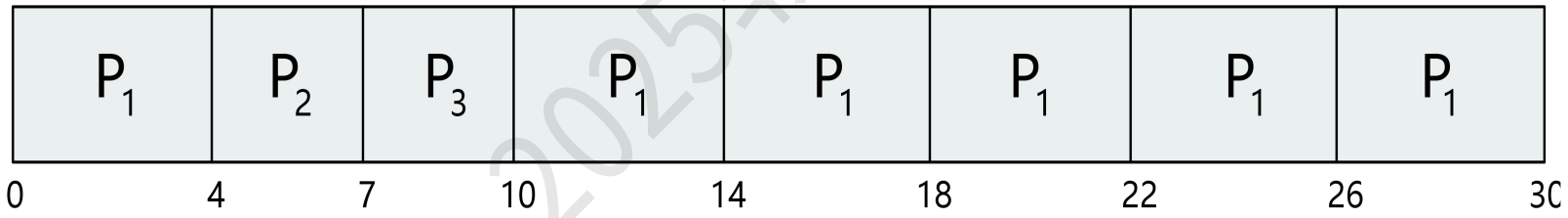
轮转调度算法 Round Robin, RR

- 背景回顾：响应时间、分时操作系统
  - A任务30秒结束，无提示；B任务40秒，但每5秒提示一次，哪个好？
- 每个进程最多允许运行一个时间片；若还没有结束，则被加入就绪队列
  - 时间片， time quantum q, 分配时间的单位，通常 10-100 毫秒
- 定时器发出时钟中断，打断当前进程的执行
  - 实际操作系统的实现方式举例：Linux
- 性能
  - q 非常大 $\Rightarrow$ 先到先服务
  - q 非常小 $\Rightarrow$ 额外开销大（上下文切换需要时间）

轮转调度算法，时间片为 4

进程	运行时间
P_1	24
P_2	3
P_3	3

- 调度图为



- 平均周转时间？平均带权周转时间？

轮转调度算法评价

- 只有抢占式的
- 普遍被采用的一种调度算法
- 可以与优先级调度算法结合使用

（查看windows任务管理器）

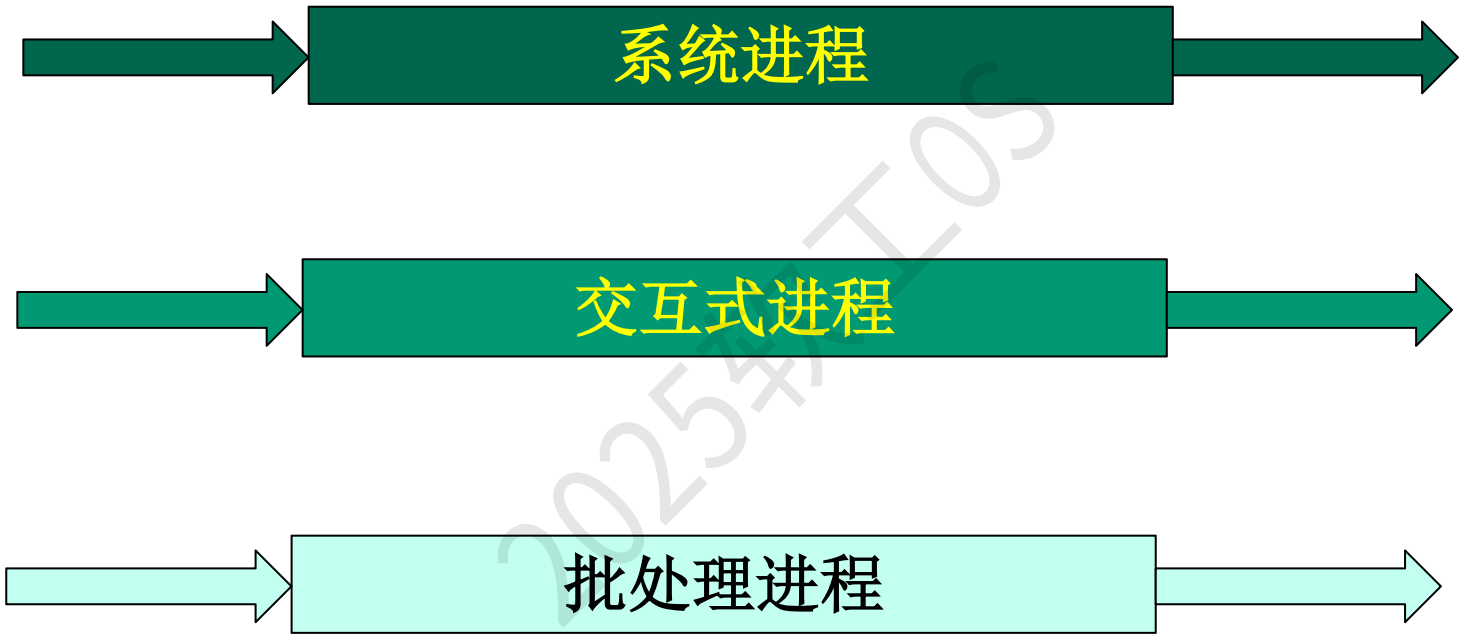
- 时间片的大小是一个问题
- 响应时间短，但平均周转时间可能长（尤其是任务时间相似时，一起“最后”结束）
- 在xv6中如何实现？--proj3
  - 进入就绪队列时初始化“可运行时间”
  - 每次tick时减少“可运行时间”，到0时切回调度器

多级队列调度

- 就绪队列进一步被划分为多个队列，如
  - 前台 (交互式的)
  - 后台 (批处理)
- 每个队列有自己独立的调度算法，如
  - 前台 – RR
  - 后台 – FCFS
- 多个队列之间如何调度？
  - 优先级调度. 可能发生饥饿
  - 按比例分配时间

多级队列调度

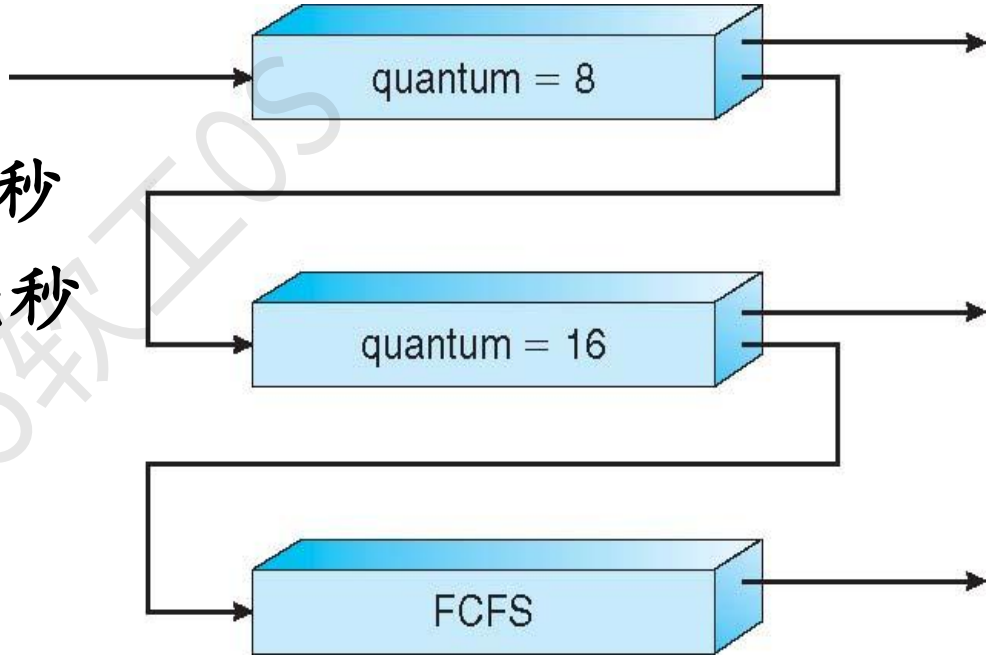
高优先级



低优先级

多级反馈队列调度

- 一个进程可以在多个队列之间移动
  - 使用CPU时间越多，优先级越低
- 例如，三个队列
  - Q0 – RR 时间片 8 毫秒
  - Q1 – RR 时间片16 毫秒
  - Q2 – FCFS

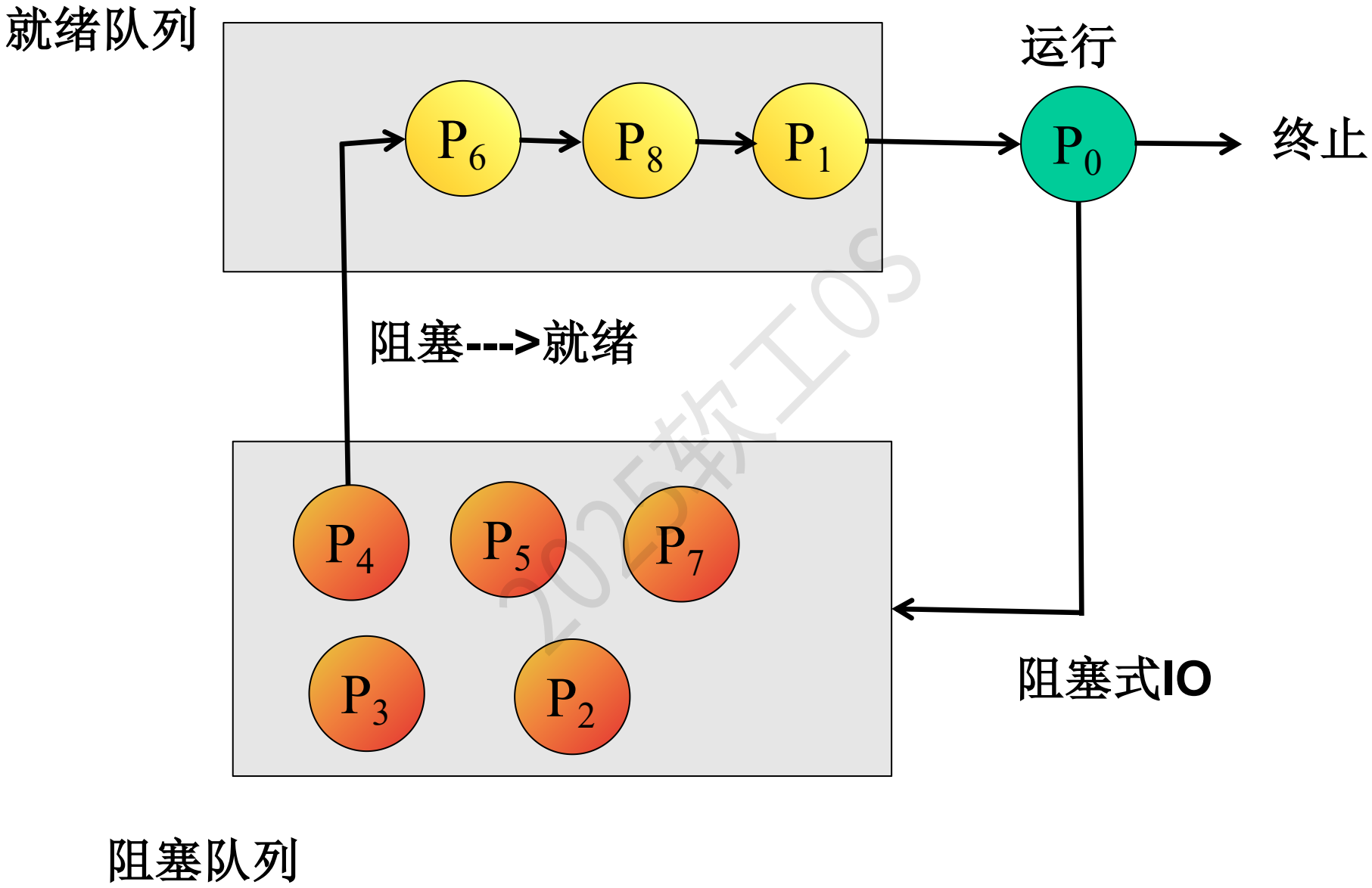


- 故事：这个调度算法有bug
  - 如果你是程序员，知道系统使用了此调度算法，你能否让你的任务一直处于高优先级？卡bug?

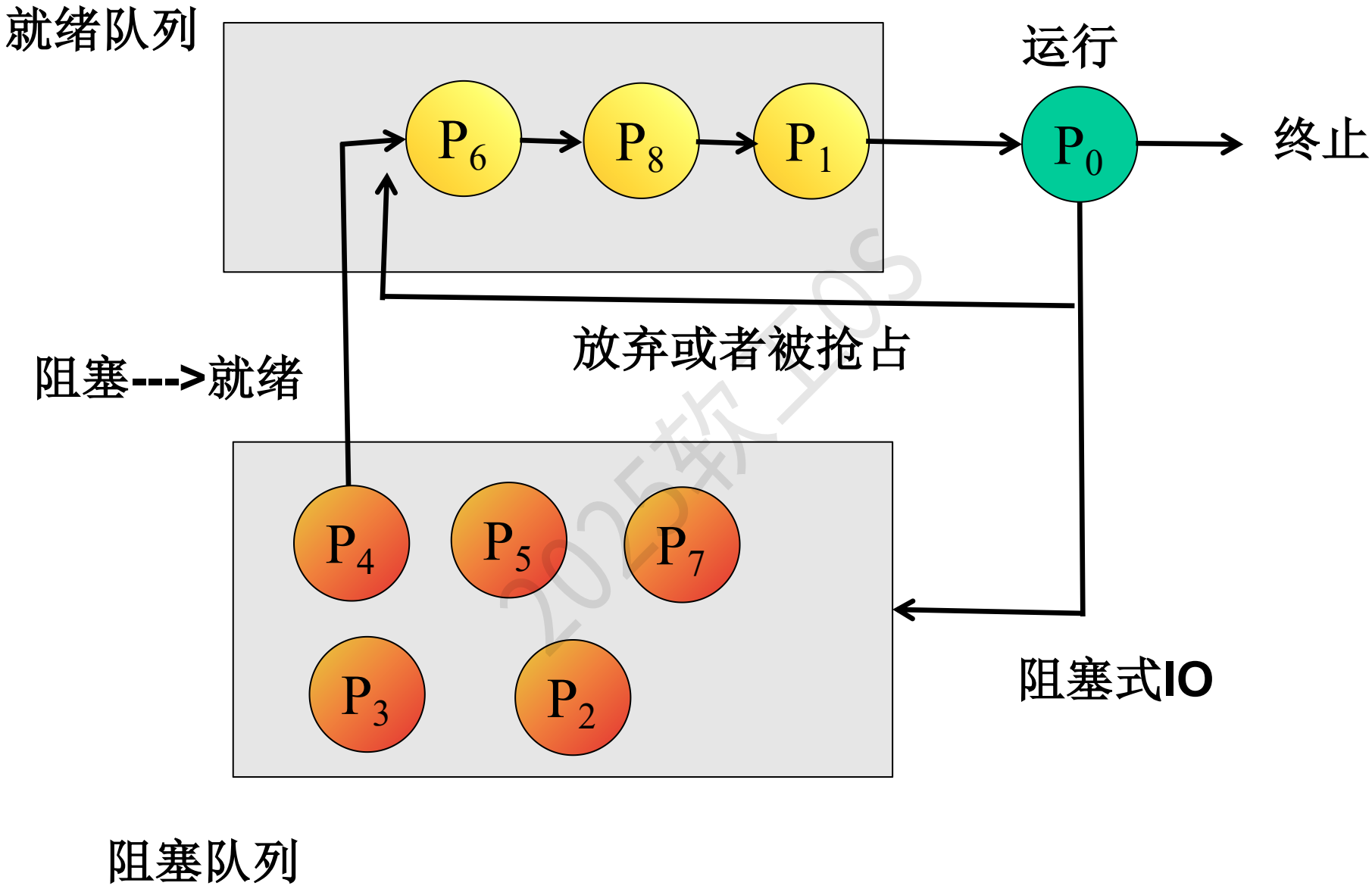
调度算法小结

- 先到先服务
- 最短作业优先
- 优先级调度
- 轮转调度
- 多级队列调度
- 多级反馈队列调度

这可能是哪一种调度算法？



这可能是哪一种调度算法？



多处理器调度

- 调度方法
- 非对称多处理器：一个CPU负责调度，其他负责执行用户代码
- 对称多处理器：
  - 每个处理器从共享的就绪队列中选择进程执行
    - 如果同时访问就绪队列，可能会出什么问题？
    - 在CPU之间迁移，高速缓存无法有效利用
  - 每个处理器一个本地就绪队列【更常见】
    - 负载均衡问题

本部分内容的要求（2025考研大纲）

- 调度的基本概念
- 调度的目标
- 调度的实现
  - 调度器/调度程序(scheduler)，调度的时机与调度方式(抢占式/非抢占式)，闲逛进程，内核级线程与用户级线程调度
- CPU调度算法
- 多处理机调度
- 上下文及切换机制

3.3 实时调度（自学）

实时调度：基本概念

- 软实时
  - 无法保证完成时刻，只保证比非实时任务优先调度
- 硬实时
  - 必须在指定的截止时间前完成
- 周期性任务
  - 每隔一段时间需要执行一次的任务，如：每200ms执行一次，每次需CPU时间为50ms
- 一般采用抢占式调度算法
- 需要某种接入控制的机制---提前拒绝
  - 如，周期性任务，单CPU，则 sum_i $C_i/P_i<=1$

最早截止时间优先算法

- 算法描述：
  - 将截止时间排序，即为调度次序
  - 如果有新的进程到达，比较截止时间，可以抢占
- 影响
  - 是采用动态优先级调度算法中的最优
- 示例：
  - 任务A、B周期分别为20ms 和 50ms，所需CPU时间分别为 10ms 和 25ms，截止时间分别为下一个周期开始之前
  - 如何调度？

最低松弛度优先算法

- 松弛度：
  - 还可以放松的时间长短 ☺
  - 截止时刻 - 当前时刻 - 需要CPU的时间
- 按松弛度从小到大的次序调度进程运行，可以抢占