



处理器调度： 单核调度策略

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰



版权信息



- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
 - <https://ipads.se.sjtu.edu.cn/courses/os/>
- 其它参考资料：
 - 清华大学操作系统公开课
 - <https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351>
 - 介绍标准内容，适合考研
 - 南京大学计算机软件研究所
 - <http://jyywiki.cn/OS/2025/>
 - <https://space.bilibili.com/202224425/channel/detail?sid=192498>
 - 比较有趣

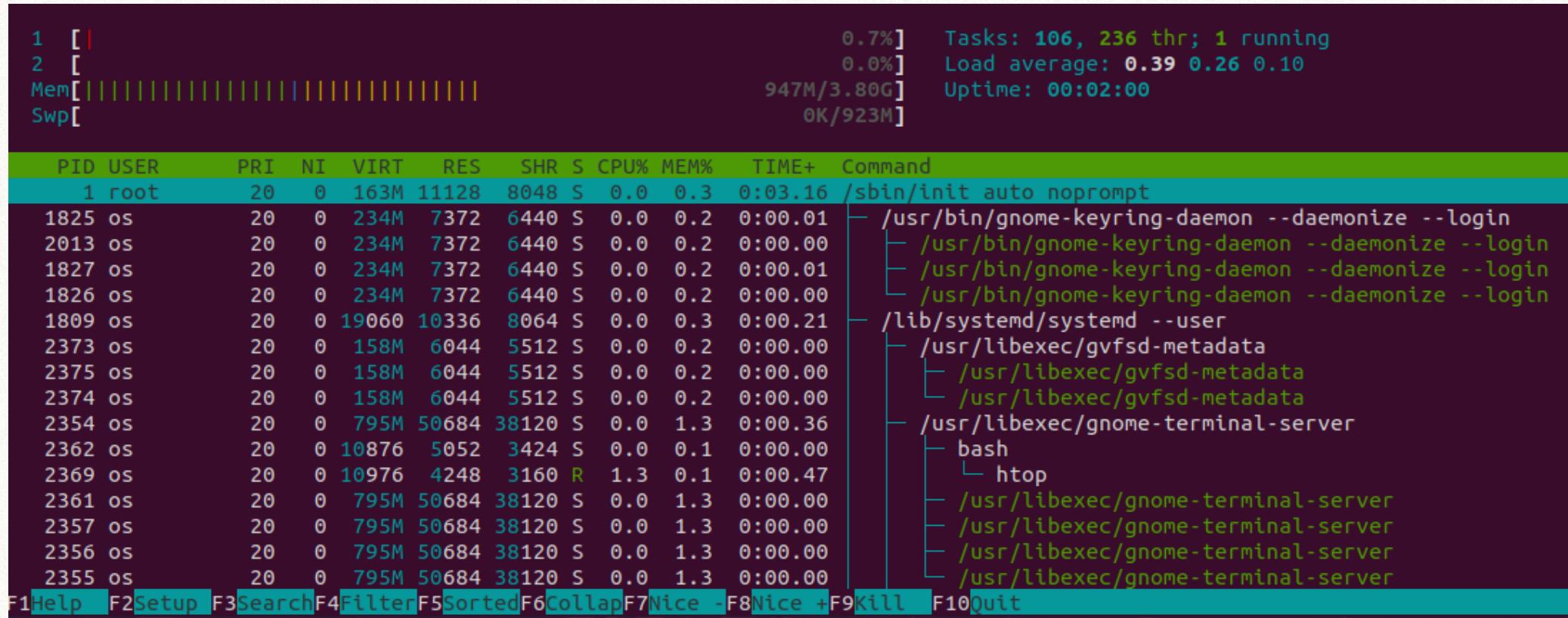


大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



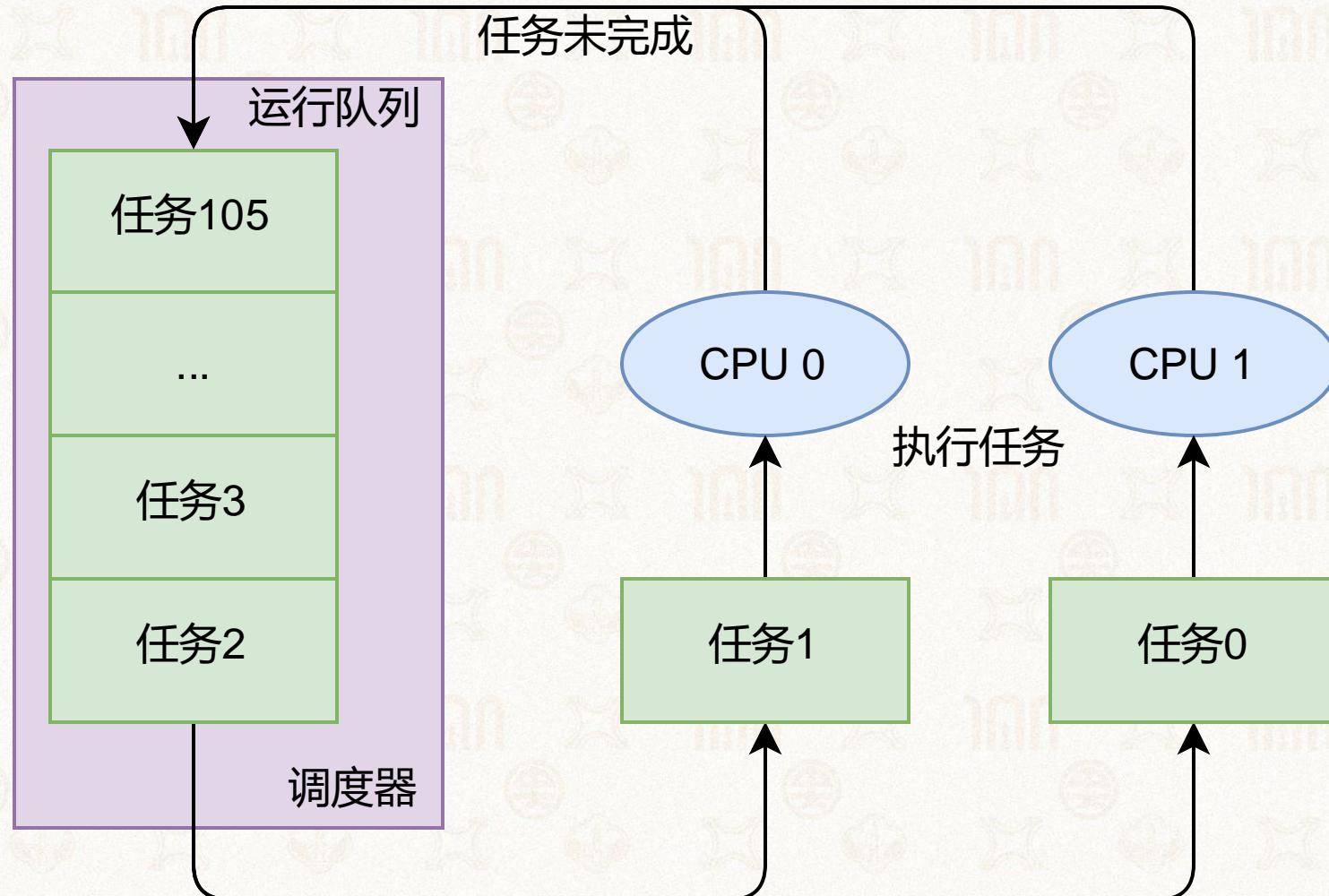
系统中的任务数远多于处理器数



仅有2个处理器，如何运行106个任务？



进程/线程调度



➤ 执行结束但未完成

- 执行时间用尽
- 等待I/O请求
- 睡眠
- 中断
- ...

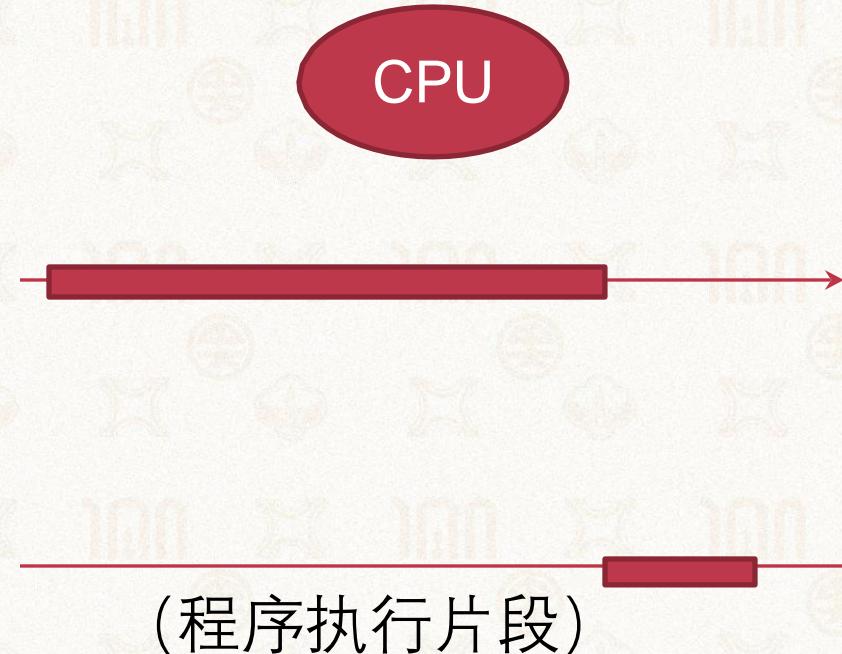
➤ 调度决策

- 下一个执行的任务
- 执行该任务的CPU
- 执行的时长



如果没有调度器

一个运行30分钟
的机器学习程序



播放音乐



程序员需要等30分钟才能播放他爱听的音乐





调度器让生活更美好

调度器 + CPU

一个运行30分钟
的机器学习程序

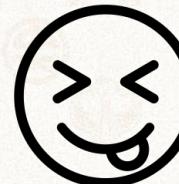


播放音乐



(程序执行片段)

调度器"人性化"地将程序切片执行
现在程序员可以边听音乐边等他的程序运行完了





什么是调度?

- 协调请求对于资源的使用





还有哪儿些调度适用的场景?

➤ I/O (磁盘)

➤ 打印机

➤ 内存

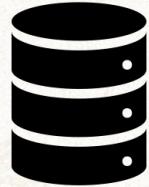
➤ 网络包

➤ ...



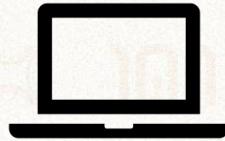
调度在不同场景下的目标

批处理系统



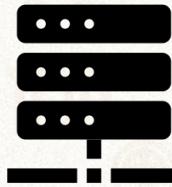
高吞吐量

交互式系统



低响应时间

网络服务器



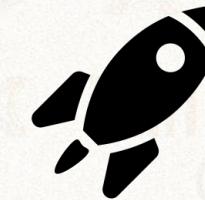
可扩展性

移动设备



低能耗

实时系统



实时性

➤ 一些共有的目标：

- 高资源利用率
- 多任务公平性
- 低调度开销



调度器的目标

- 降低周转时间
 - 任务第一次进入系统到执行结束的时间
- 降低响应时间
 - 任务第一次进入系统到第一次给用户输出的时间
- 实时性
 - 在任务的截止时间内完成任务
- 公平性
 - 每个任务都应该有机会执行，不能饿死
- 开销低
 - 调度器是为了优化系统，而非制造性能BUG
- 可扩展
 - 随着任务数量增加，仍能正常工作



调度的挑战

- 缺少信息（没有正确答案）
- 工作场景动态变化
- 线程/任务间的复杂交互
- 调度目标多样性
 - 不同的系统可能关注不一样的调度指标
- 许多方面存在取舍
 - 调度开销 V.S. 调度效果
 - 优先级 V.S. 公平
 - 能耗 V.S. 性能



大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



策略 V.S. 机制

➤ 策略

- 做什么
- 从上层去分析、解决问题

➤ 机制

- 怎么做
- 实现某一策略、功能

主题	策略	机制
上课	讲操作系统调度	课堂、网课
上课	交作业	坚果云、纸质
科研	写C++代码	VSCode、Clion
科研	写论文(Latex)	VSCode、Overleaf



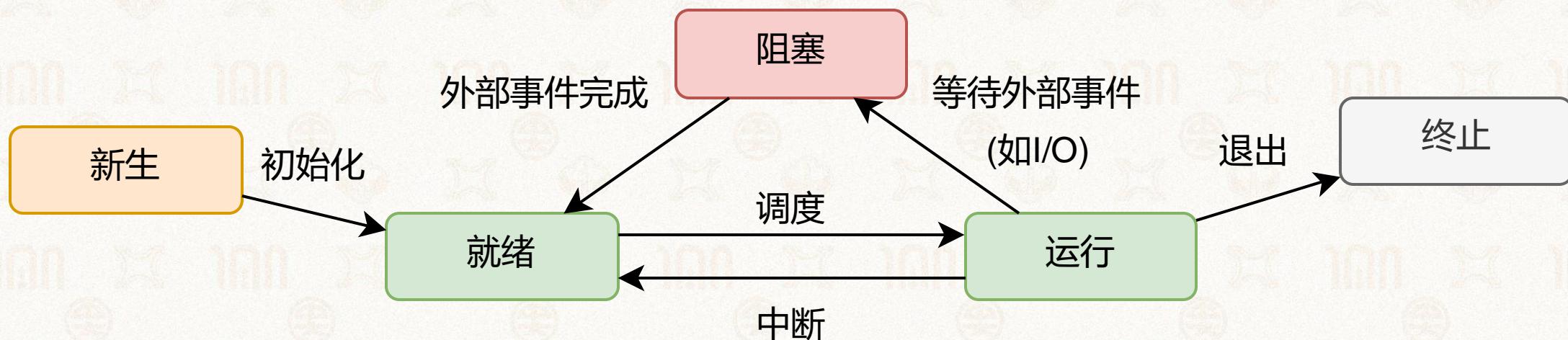
回顾：进程的状态

➤ 进程至少应当拥有以下五种状态：

- 新生状态 (new) : 进程刚被创建
- 就绪状态 (ready) : 进程可以运行, 但没有被调度
- 运行状态 (running) : 进程正在处理器上运行
- 终止状态 (terminated) : 进程完成了执行
- 阻塞状态 (blocked) : 进程进入等待状态, 短时间不再运行

➤ 进程会不断进行状态切换

- 被调度器调度, 开始执行: 准备->运行





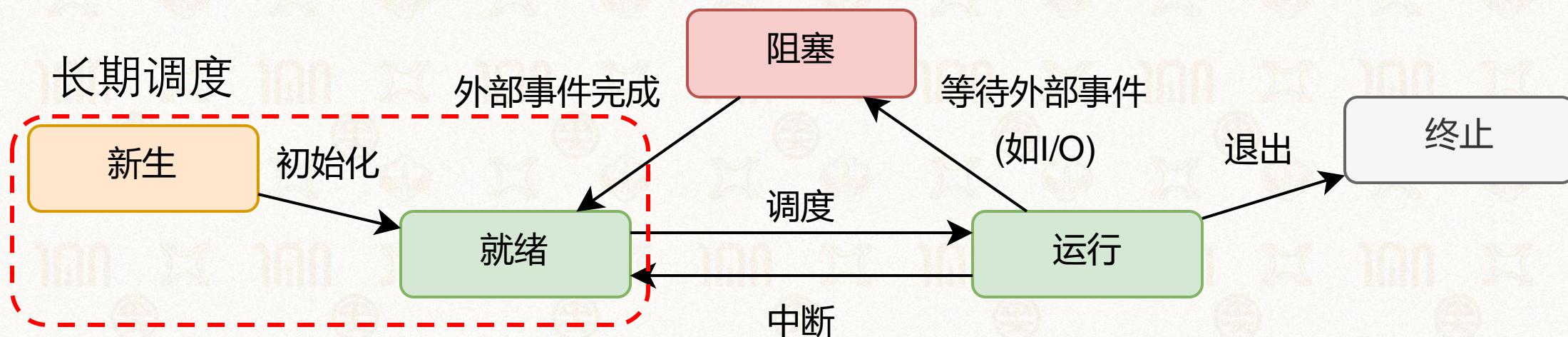
长期、短期调度

➤ 长期调度策略

- 限制真正被短期调度的进程数量
- 管理系统资源利用率
- 只管大局，不是谁都可以马上被运行

➤ 短期调度策略

- 负责和运行状态相关的调度





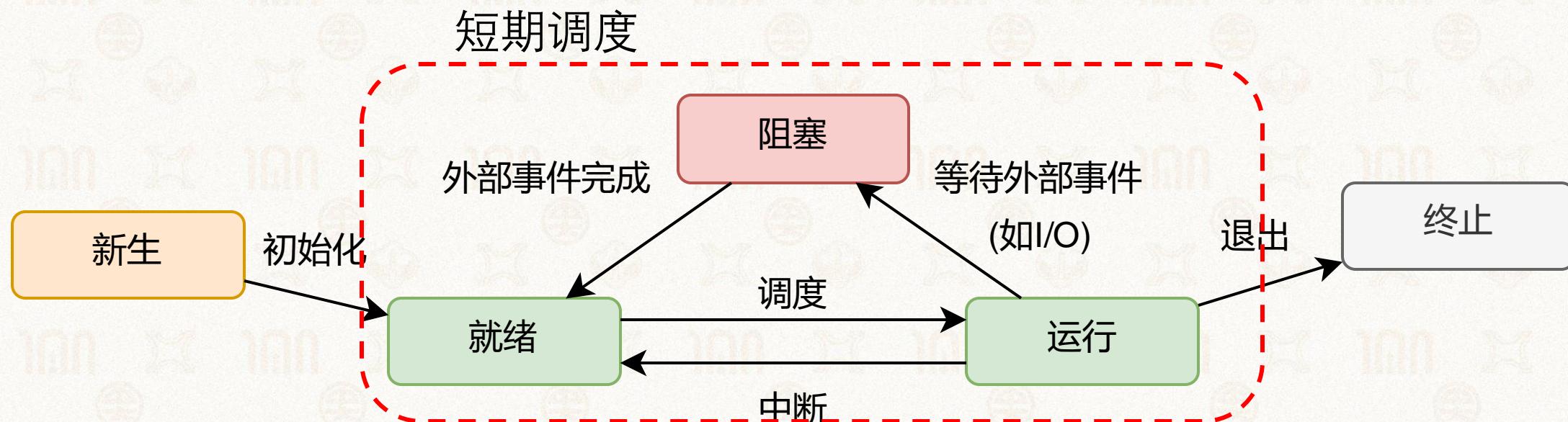
长期、短期调度

➤ 长期调度策略

- 限制真正被短期调度的进程数量
- 管
- 只管大局，不是谁都可以马上被运行

➤ 短期调度策略

- 负责和运行状态相关的调度

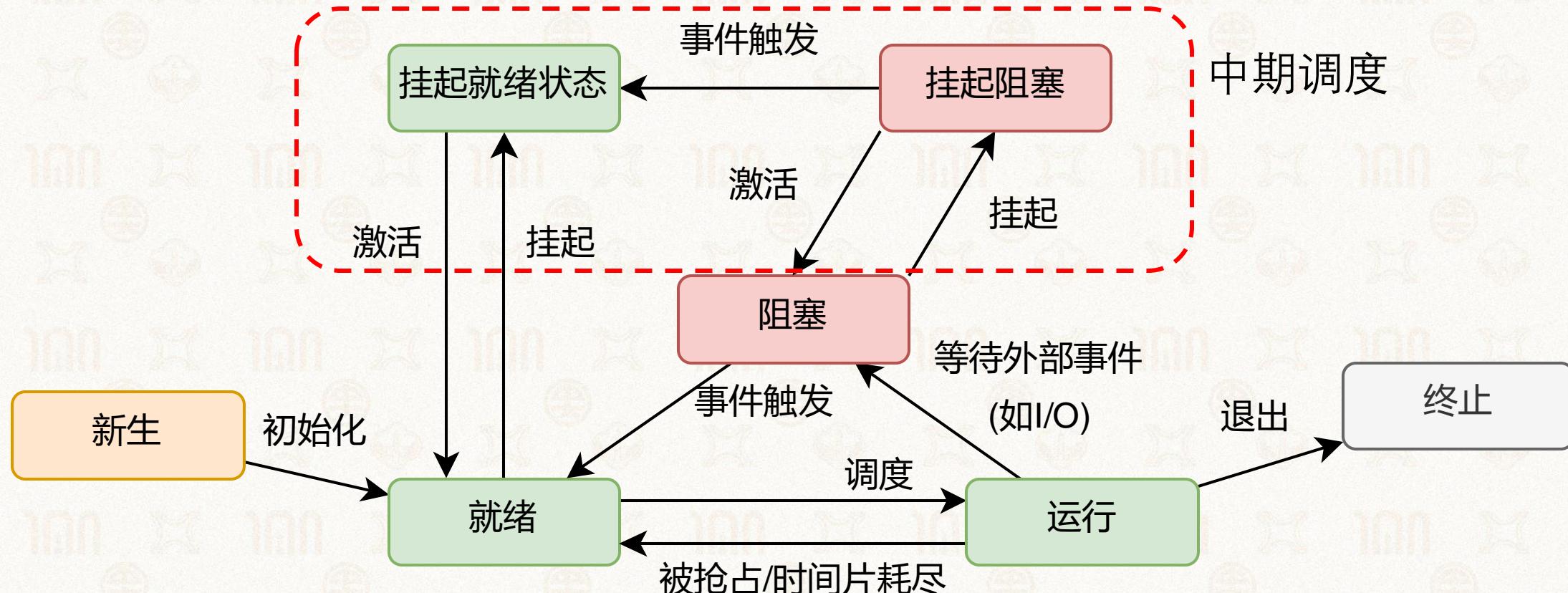




中期调度

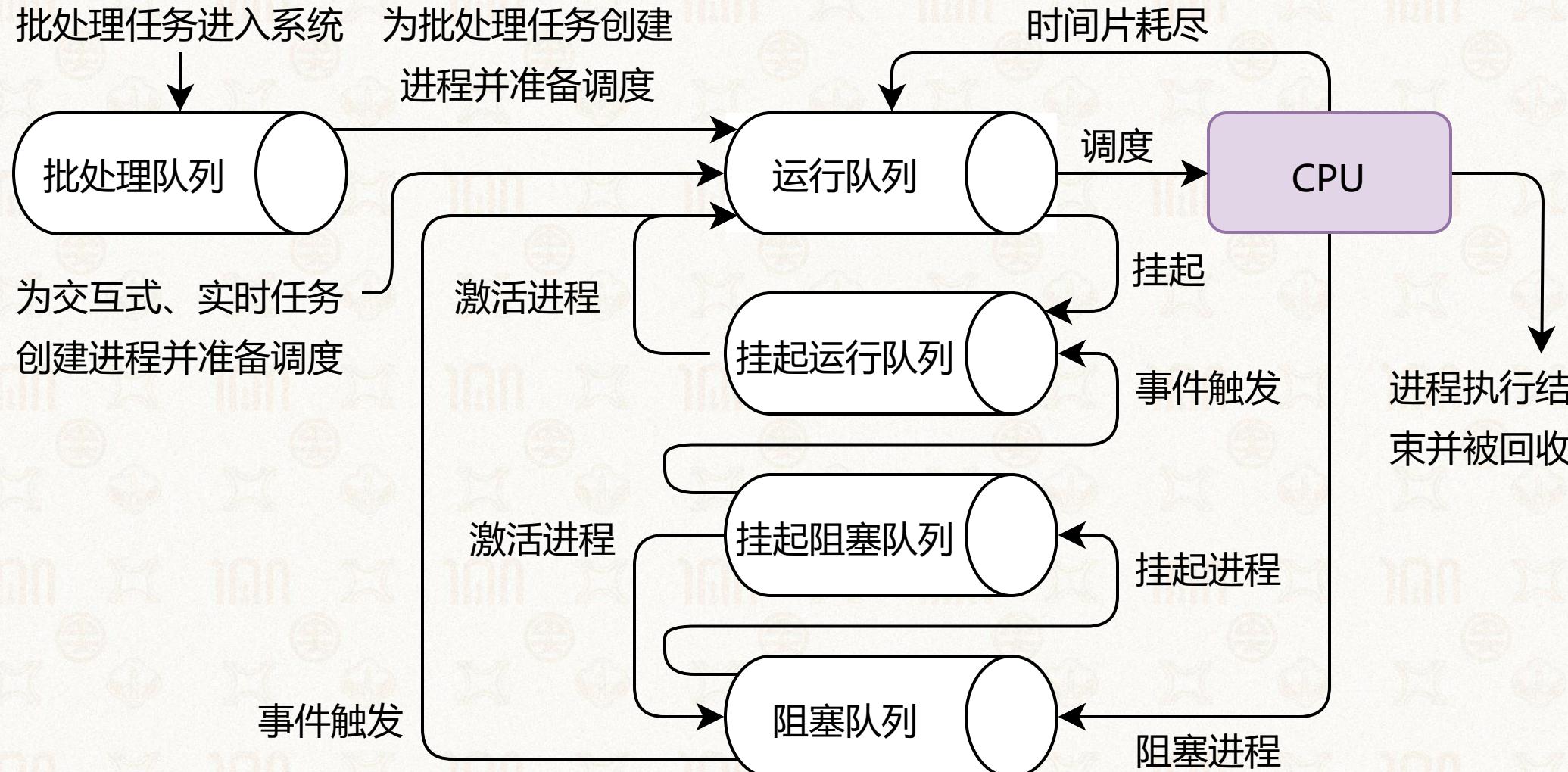
中期调度策略

- 内存空间不足时选择挂起一些进程
 - 影响性能的进程：频繁触发缺页异常、长时间未响应的进程
- 优先换出挂起的进程进入硬盘





进程调度总览





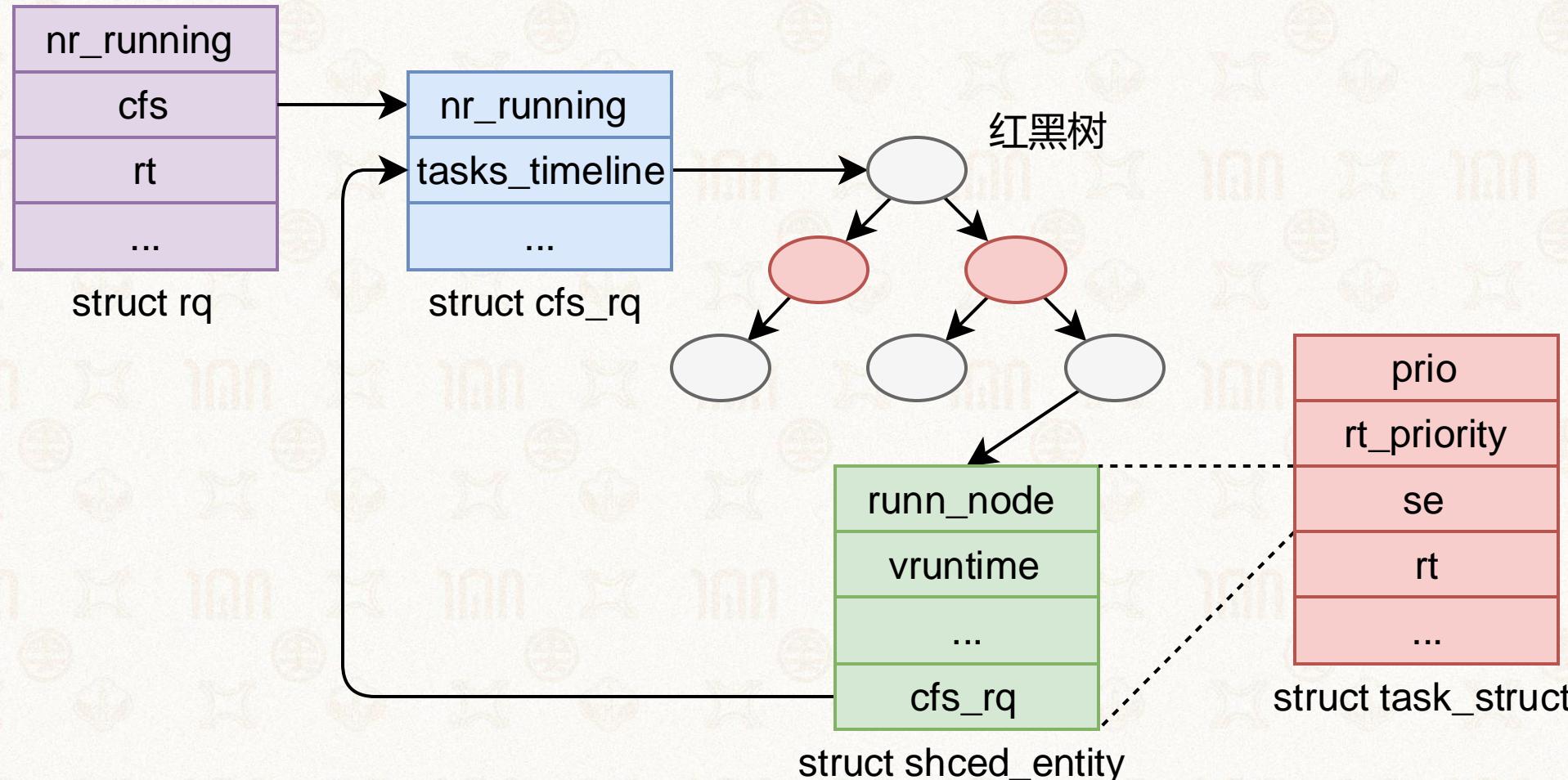
举例：Linux中的调度策略

- 为了满足不同需求提供多种调度策略
- 以Linux两种调度器为例，每种对应多个调度策略
 - 公平调度(Complete Fair Scheduler, CFS)
 - SCHED_OTHER
 - SCHED_BATCH
 - SCHED_IDLE
 - 实时调度(Real-Time Scheduler, RT)
 - SCHED_FIFO
 - SCHED_RR



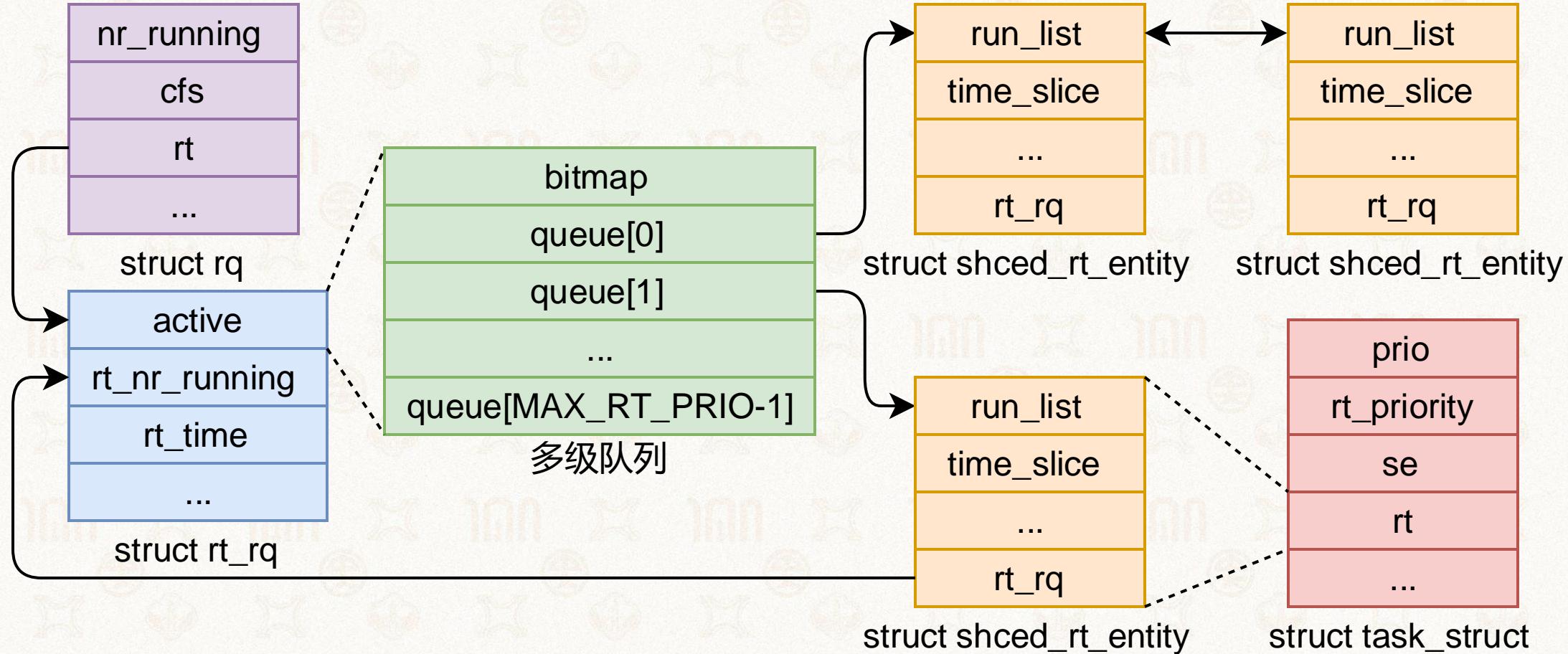
Linux调度机制：公平调度器运行队列

公平调度器(CFS)运行队列(Run Queue, RQ)





Linux调度机制：实时调度器运行队列



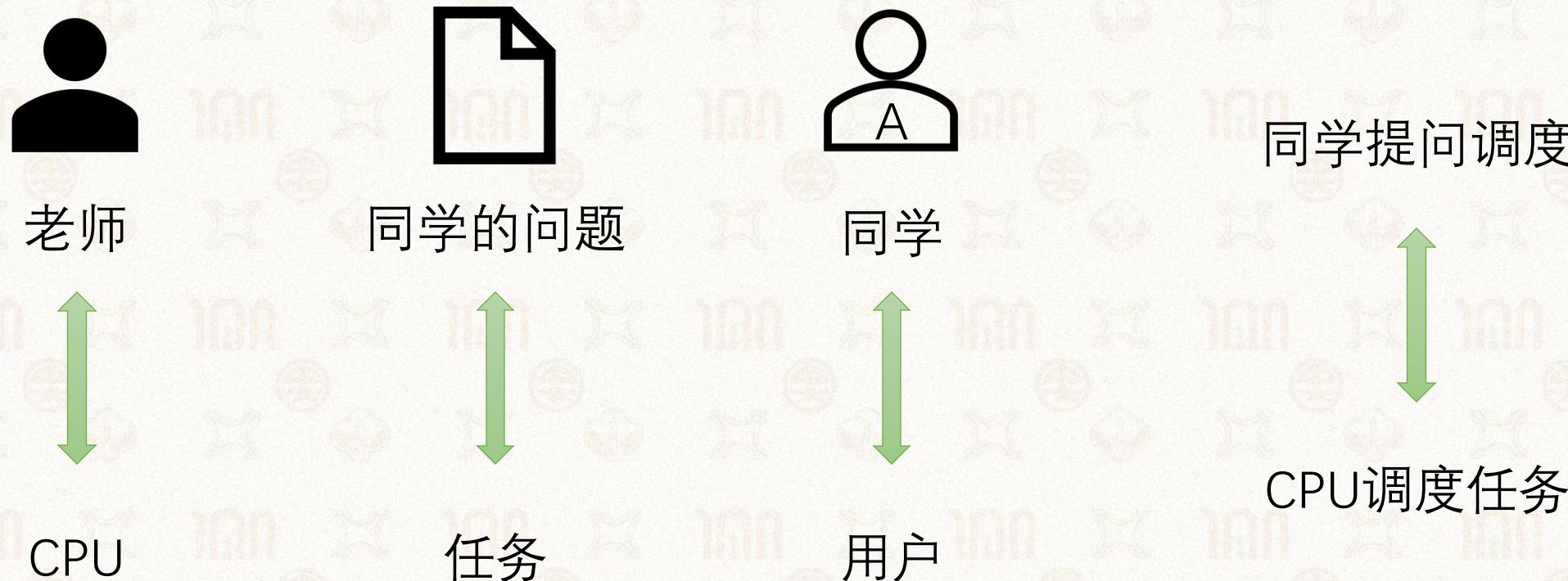


大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享受调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



CPU调度与提问调度



当前假设每位同学只提一个问题

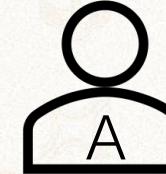
你觉得同学上台提问，老师解答问题这个场景应该怎么安排才能让你最舒服？



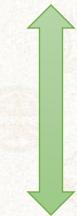
老师



同学的问题



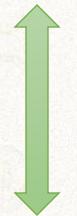
同学



CPU

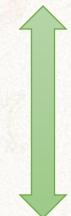


任务



用户

同学提问调度



CPU调度任务

当前假设每位同学只提一个问题

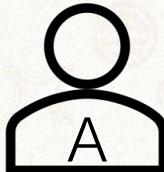
作答



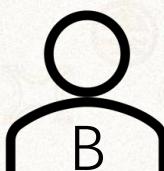
策略：先到先得(First Come First Served)



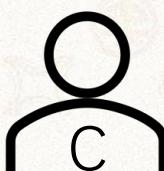
大家排队 先来后到!



得嘞，我第一

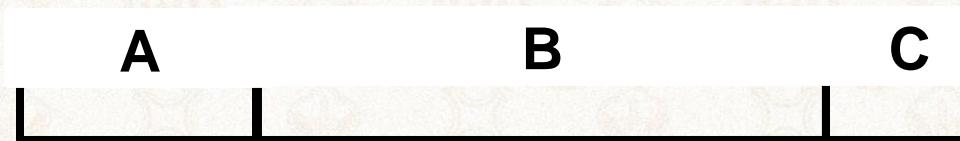


C,先来后到!



我的问题很简单却要等
那么长时间…

问题	到达时间	解答时间 (工作量)
A	0	4
B	1	7
C	2	2



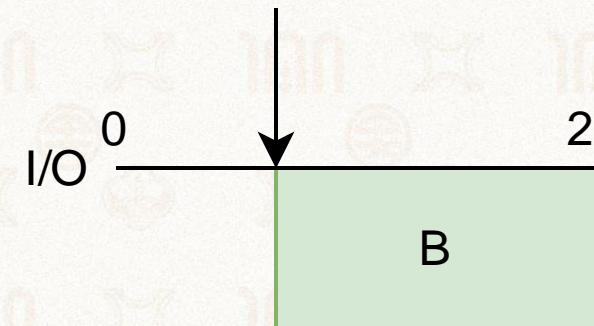
先到先得：简单、直观

问题：平均周转、响应时间过长

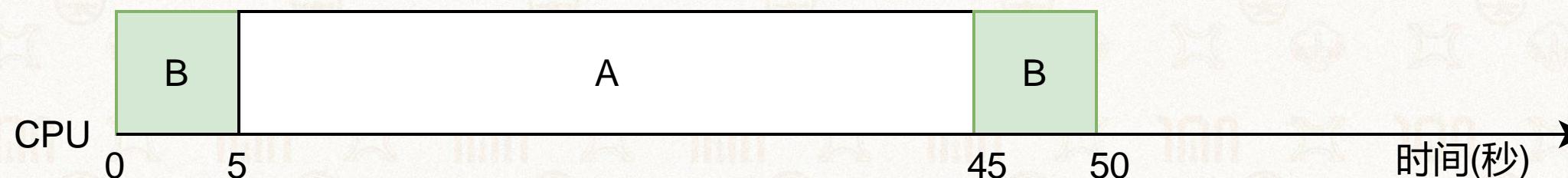
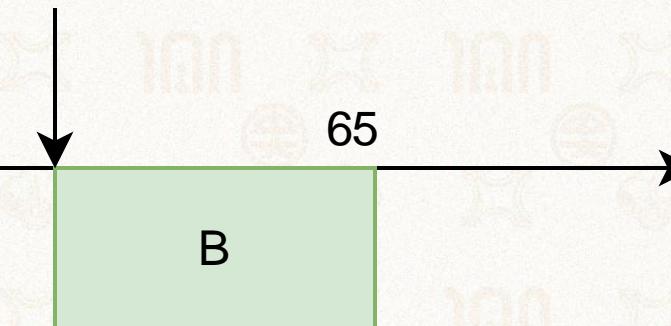


先到先得弊端：对I/O密集型任务不友好

B等待I/O事件，转入阻塞状态



B等待I/O事件，转入阻塞状态

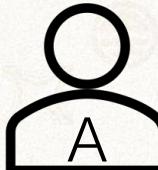




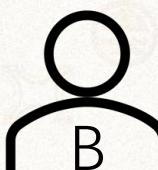
策略：最短任务优先(Shortest Job First)



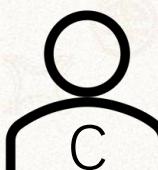
简单的问题先来



我最先到， 我还是第一！



万一再来个短时间的
D， 那我要等死了…



我可以先于B了

问题	到达时间	解答时间 (工作量)
A	0	4
B	1	7
C	2	2



短任务优先：平均周转时间短

- 问题： 1) 不公平， 任务饿死
2) 平均响应时间过长



抢占式调度 (Preemptive Scheduling)

➤ 非抢占式调度

- 一定是前一个任务执行完毕再执行下一个
- 例：先到先得、最短任务优先

➤ 抢占式调度：

- 每次任务执行一定时间后会被切换到下一任务
- 而非执行至终止
- 通过定时触发的时钟中断实现
- 例：最短完成时间任务优先(Shortest Time-to-Completion First)

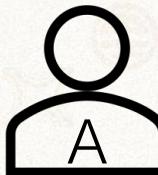


策略：最短完成时间任务优先

➤ 注意一下和“最短任务优先”的区分

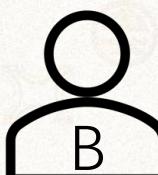


简单的问题可以**插嘴**！

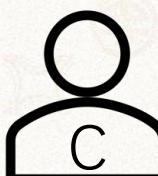


我最先到， 我先来！

哎， 我还没问完呢！

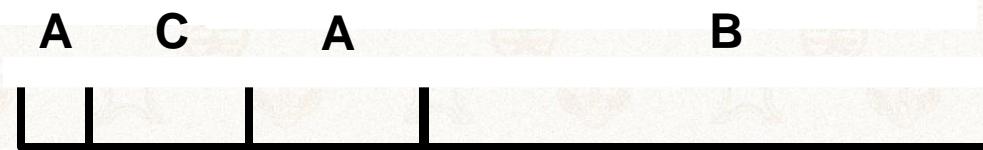


抢不过你们…



我的简单， 我来了！

问题	到达时间	解答时间 (工作量)
A	0	4
B	2	7
C	1	2



最短完成时间：平均周转时间短
问题：

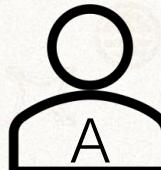
- 1) 会打断正在运行的任务



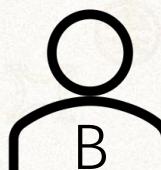
策略：时间片轮转(Round Robin)



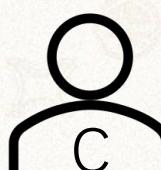
公平起见每人轮流一分钟!



感觉多等了好久…

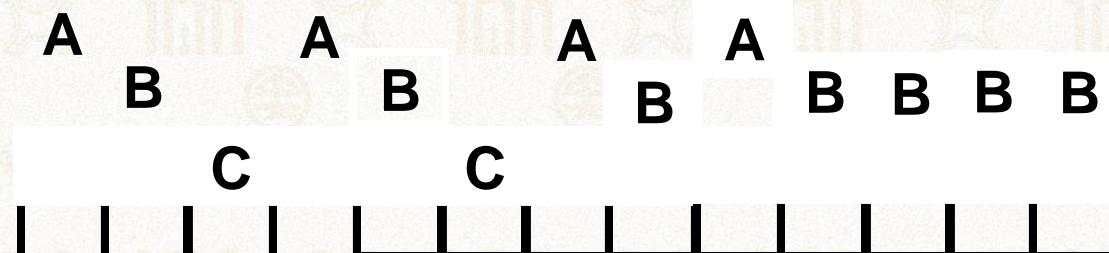


老师的响应时间短了好多



老师响应得更快了

问题	到达时间	解答时间(工作量)
A	0	4
B	1	7
C	2	2



轮询：公平、平均响应时间短

问题：牺牲周转时间



时间片轮转(Round Robin)

- 什么情况下RR的周转时间问题最为明显?
- 时间片长短应该如何确定?
 - 过长的时间片会导致什么问题?
 - 过短的时间片会导致什么问题?

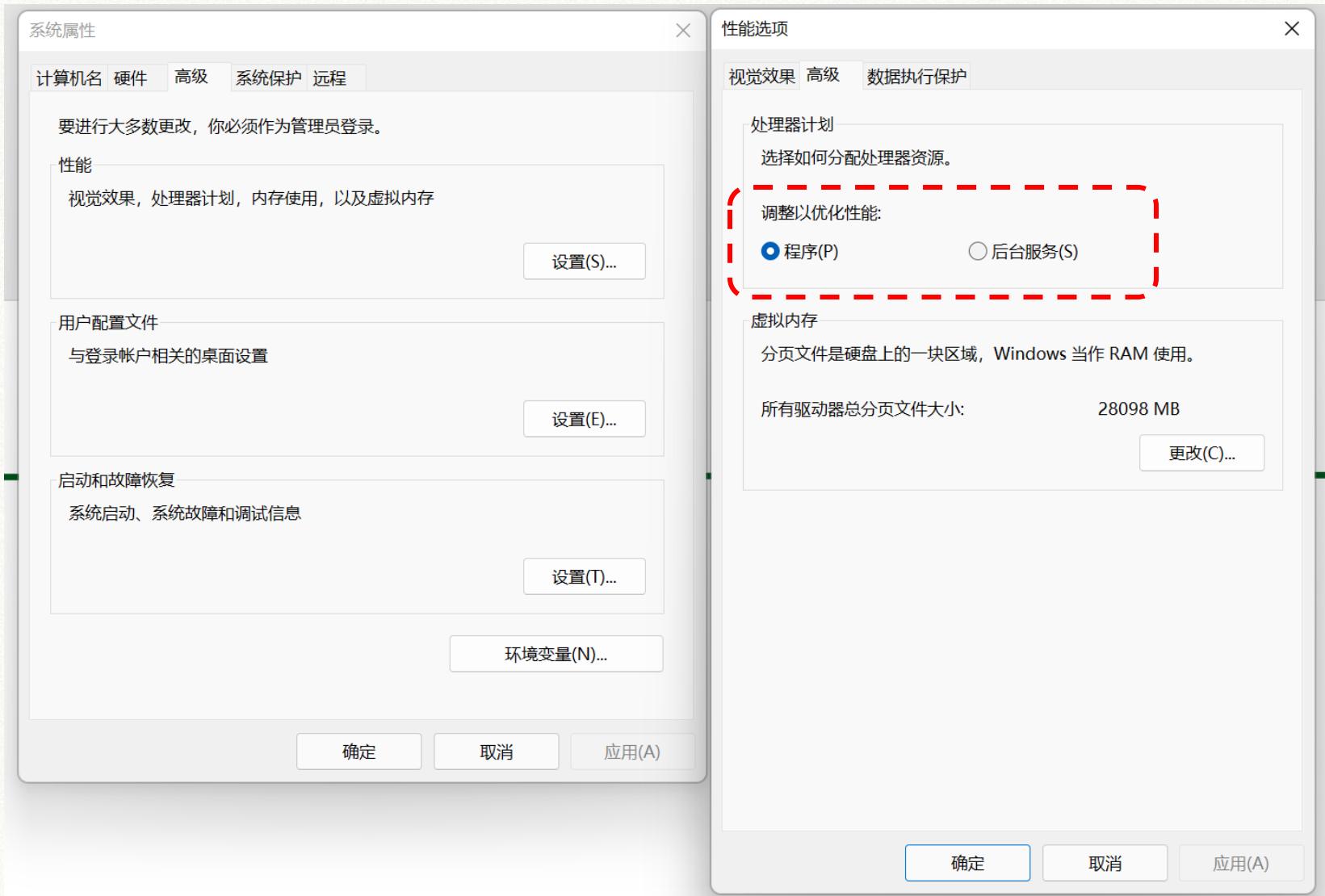


大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



操作系统里的任务是分三六九等的





调度优先级

- 操作系统中的任务是不同的，例如：
 - 系统 V.S. 用户、前台 V.S. 后台、...
- 如果不加以区分
 - 系统关键任务无法及时处理
 - "后台运算"导致"视频播放"卡顿
- 优先级用于确保重要的任务被优先调度

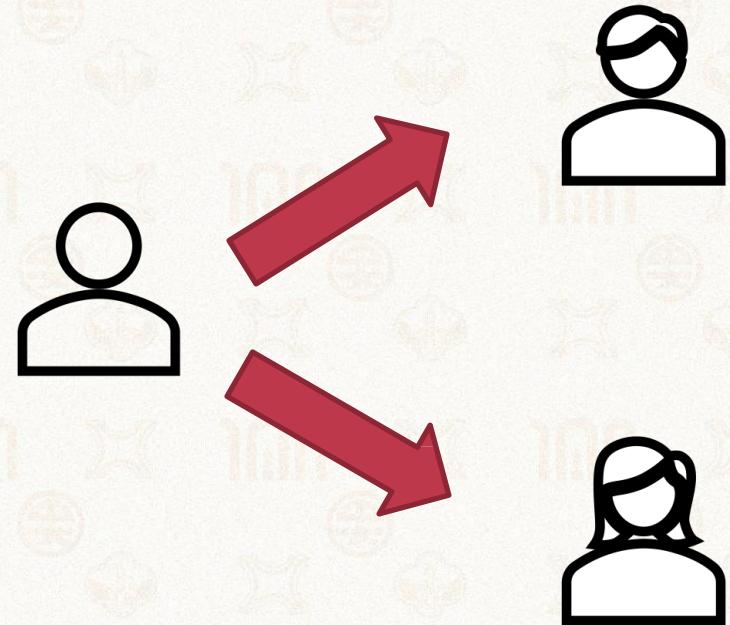


不同调度策略隐含对应的优先级确定方式

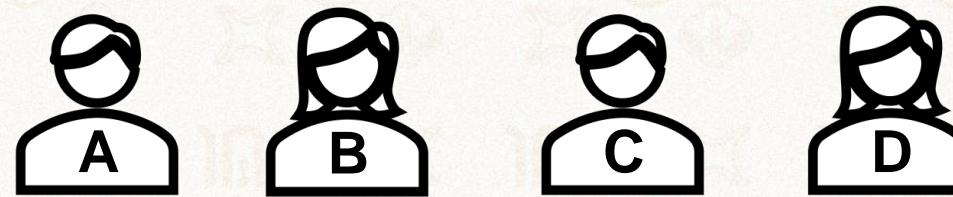
- 先到先得
 - 任务到达时间早的优先级高
- 最短任务优先
 - 任务运行时间短的优先级高
- 最短完成时间任务优先
 - 任务剩余完成时间短的优先级高
- 时间片轮转
 - 所有任务平等



添加条件：优先级



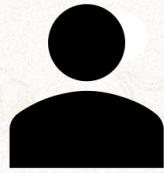
女生优先级高!



- 有明确截止时间的实时任务，优先级最高
- 交互式任务分配较高优先级
- 批处理任务优先级较低

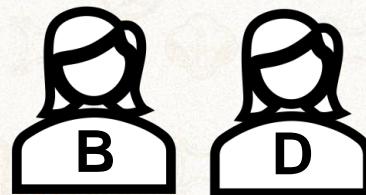


基于多级队列的调度策略

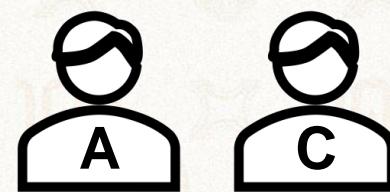


先回答 B,D
然后再回答 A,C

优先级0（高）



优先级1（低）

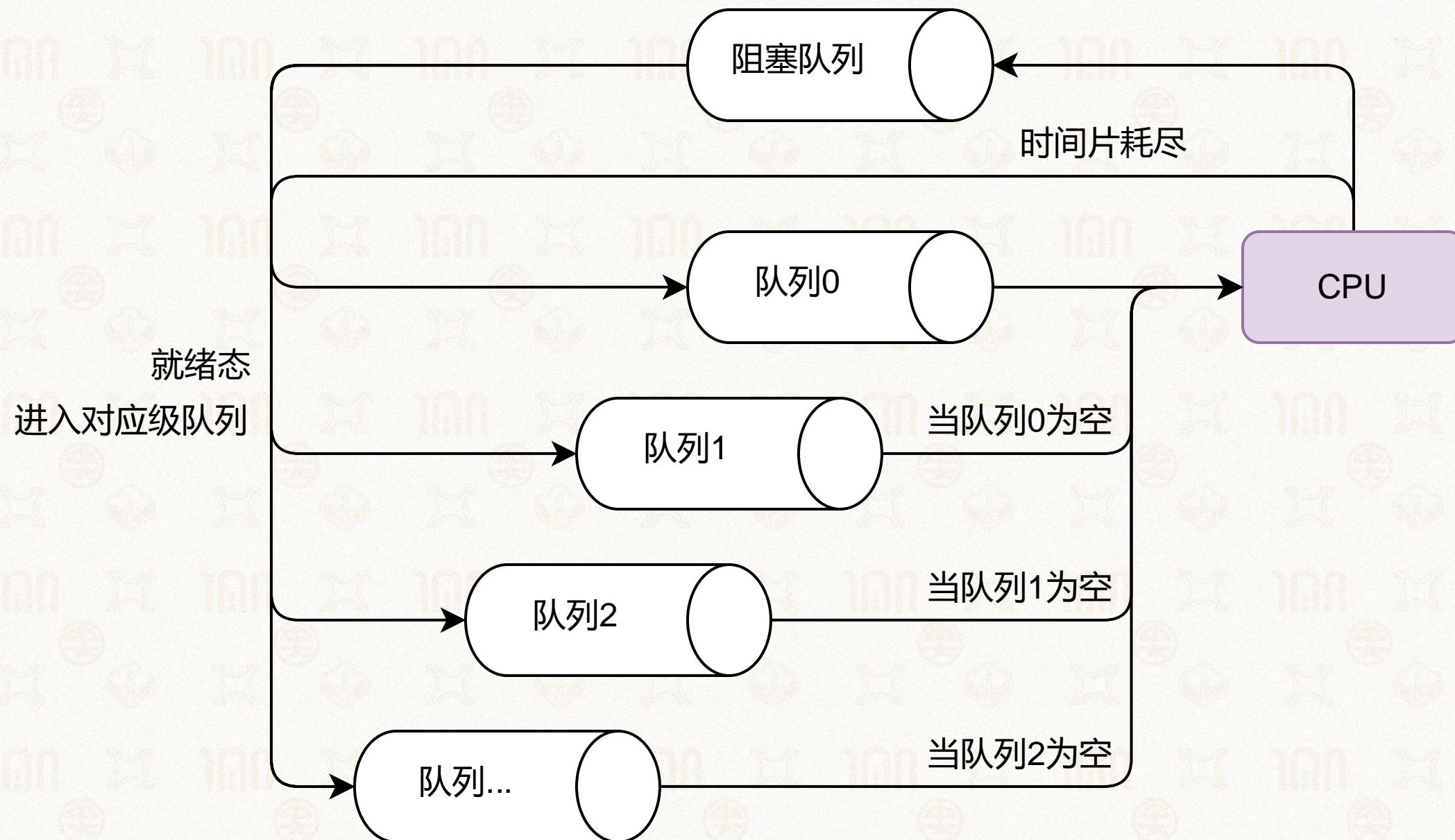


多级队列：

- 1) 维护多个优先级队列
- 2) 高优先级的任务优先执行
- 3) 同优先级内使用时间片轮转调度

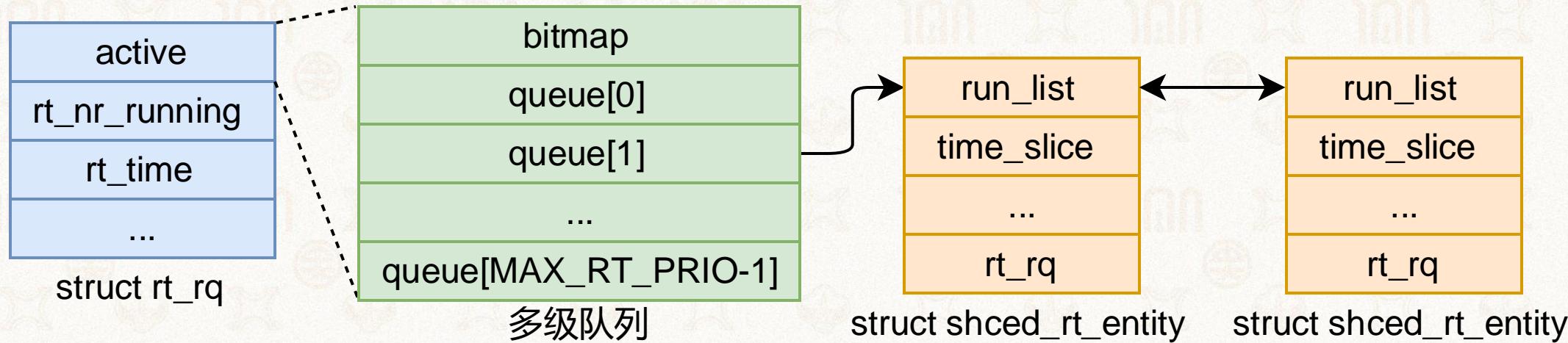


基于多级队列的调度策略





Linux调度机制：实时调度器运行队列

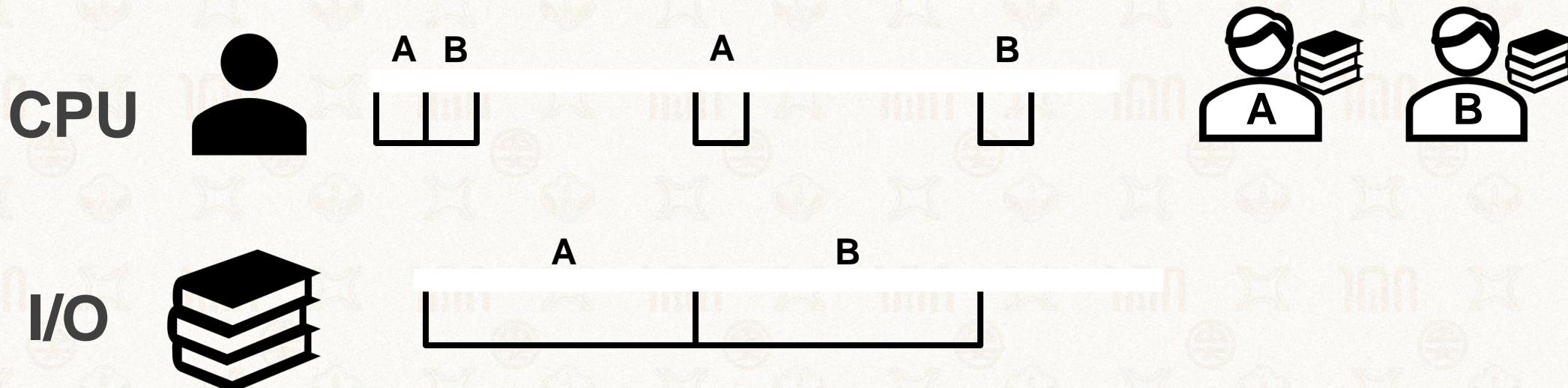


- 使用多级队列实现优先级调度
 - 每个任务有自己的优先级、具体策略
 - 具体策略可根据任务需求针对性选择
 - SCHED_RR: 任务执行一定时间片后挂起
 - SCHED_FIFO: 任务执行至结束



继续增加现实约束：阅读OS书（类比I/O操作）

- 老师告诉同学需要看OS书
 - 老师只有一本OS书，同一时间只有一个同学能够阅读
- 阅读完OS书后，同学再和老师确认知识点



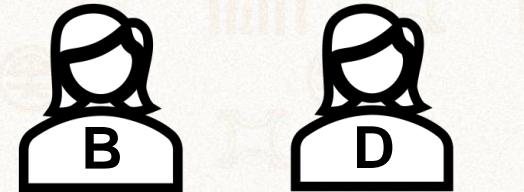


问题1：低资源利用率

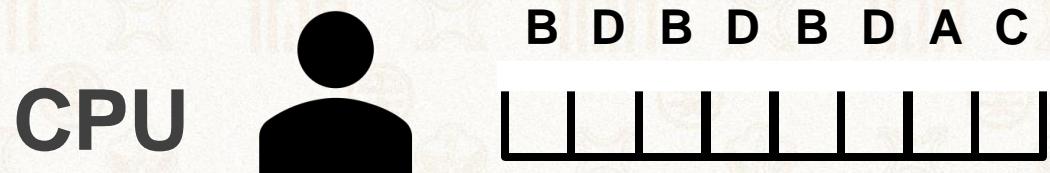
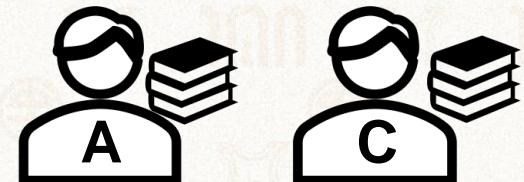
➤ 问题：

- 多种资源（老师和OS书）没有同时利用起来

优先级0（高）



优先级1（低）





什么样的任务应该有高优先级?

- I/O绑定的任务
 - 为了更高的资源利用率
- 用户主动设置的重要任务
- 时延要求极高（必须在短时间内完成）的任务
- 等待时间过长的任务
- 为了公平性



问题2：优先级反转

- 高、低优先级任务都需要独占共享资源
 - 共享资源
 - 存储
 - 硬件
 - OS书
 - ...
 - 通常使用信号量、互斥锁实现独占
- 反转：低优先任务占用资源导致高优先级任务被阻塞

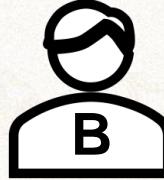
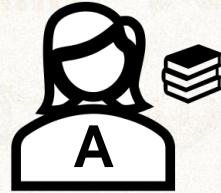


问题2：优先级反转

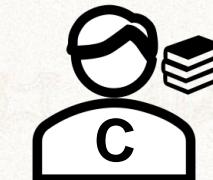
问题：

A被C占有的资源**阻塞**

优先级较低的B先于A学习



优先级：A>B>C



2. 抢占C
申请OS书失败
等待

3.B优先级高于C
可以向老师提问



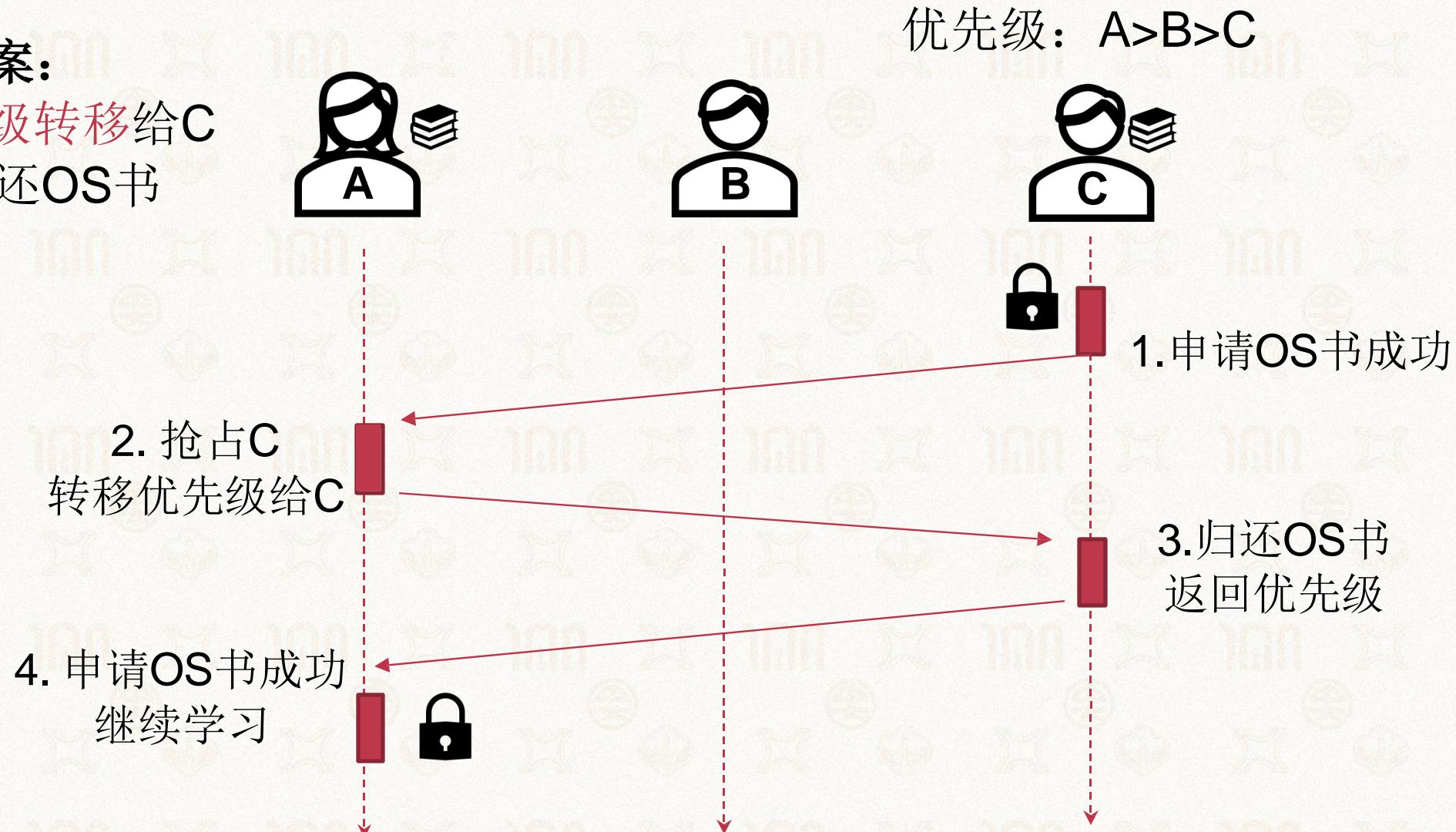
1. 申请OS书成功



解决方法：优先级继承

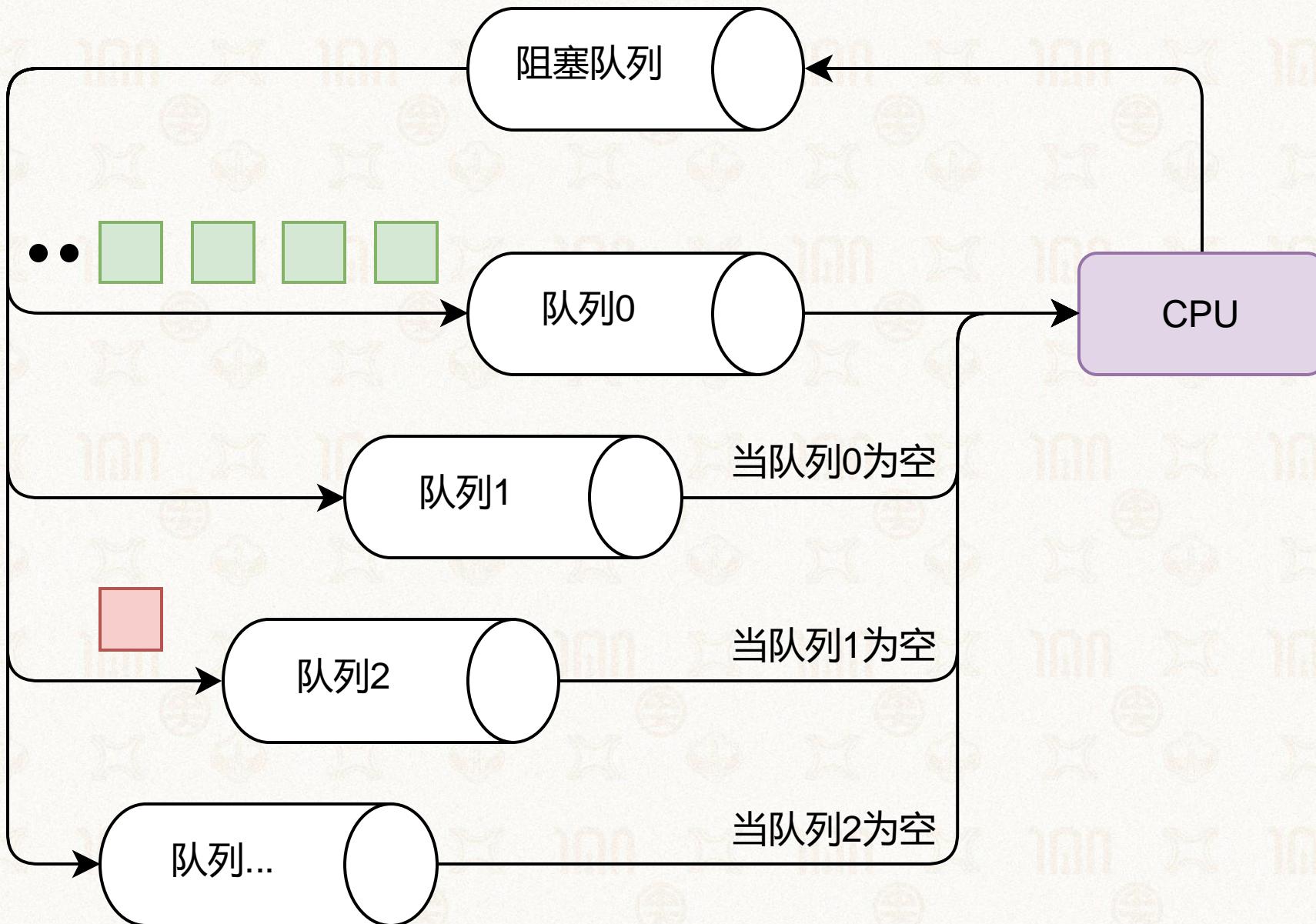
解决方案：

A暂时将优先级转移给C
让C尽快归还OS书





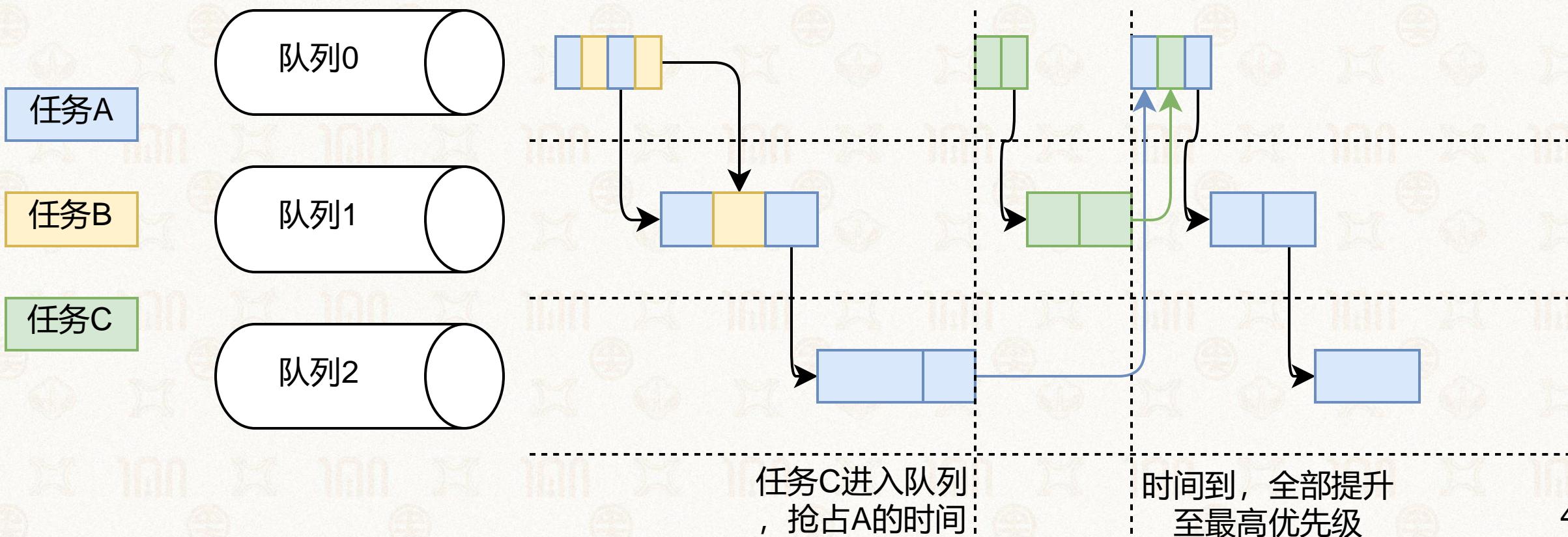
问题3：低优先级任务饥饿





多级反馈队列(Multi-Level Feedback Queue, MLFQ)

- 初始默认都是短任务，短任务拥有更高优先级
- 低优先级的任务采用更长的时间片
- 定时将所有任务的优先级提升至最高





大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



场景：云计算平台多租户共享处理器

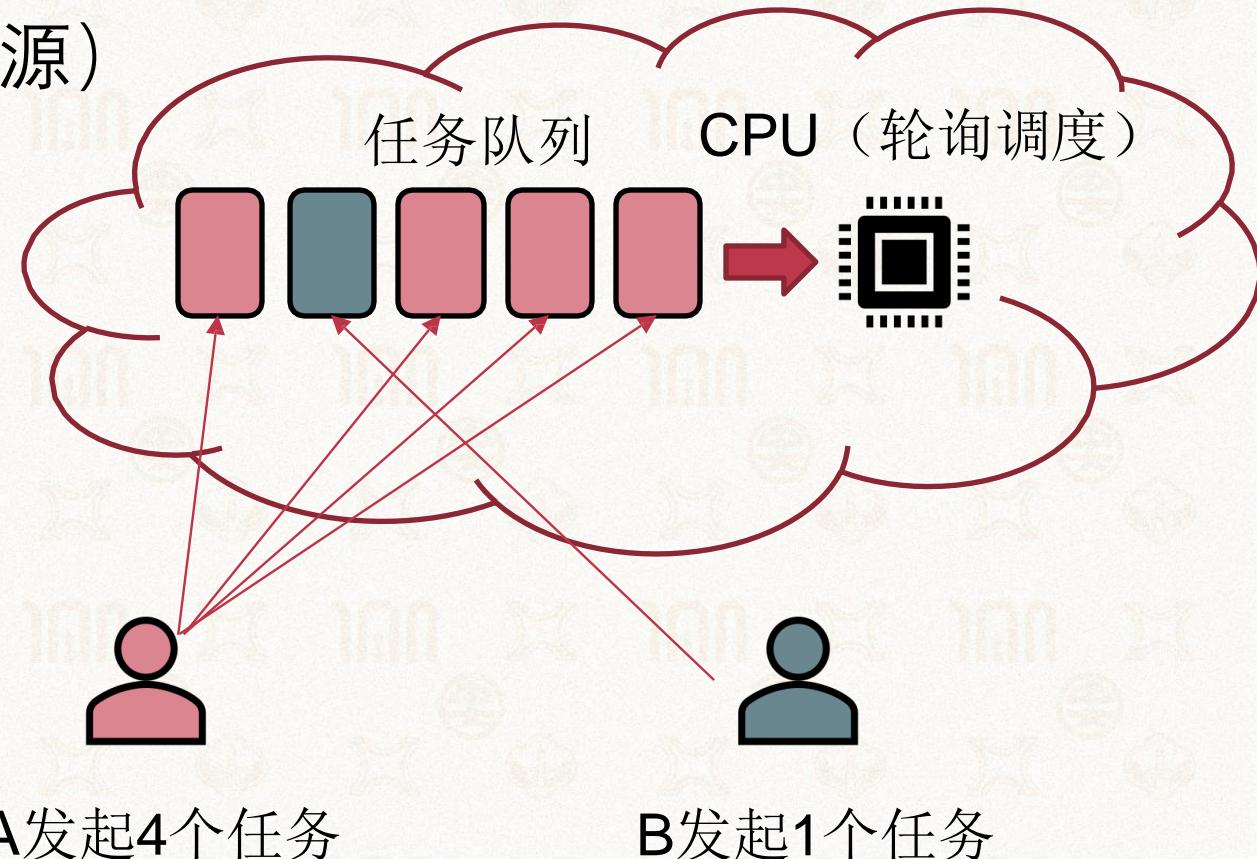
➤ 在云平台中，计算资源是有价值的

➤ 租户在意自己的CPU时间（资源）

- 两个相同的租户应均分CPU时间
- 而非被发起的任务数量决定

➤ 假设CPU使用时间片轮转调度

- A占用80%CPU时间
- B占用20%CPU时间



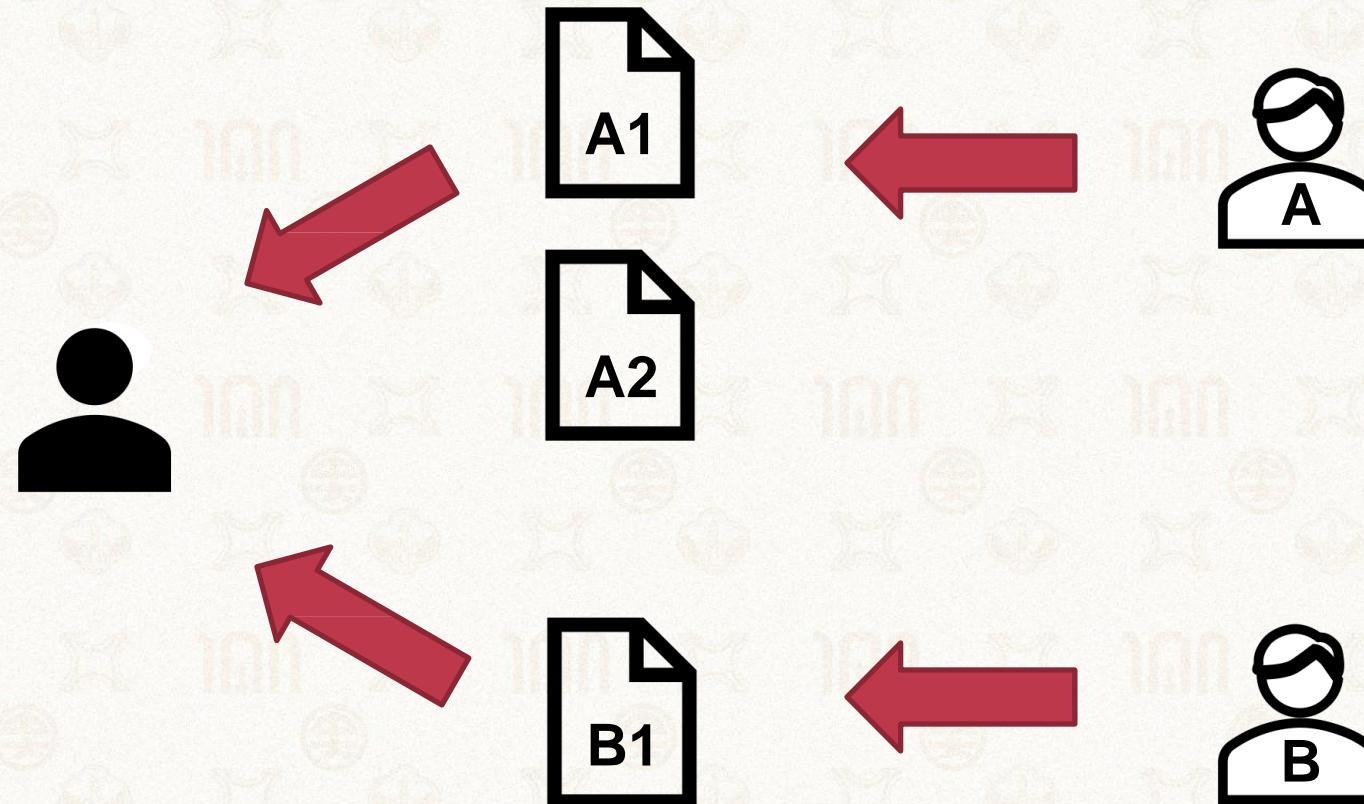


公平共享

- 每个用户占用的资源是成比例的
 - 而非被任务的数量决定
- 每个用户占用的资源是可以被计算的
 - 设定"权重值"以确定相对比例 (绝对值不重要)
 - 例：权重为4的用户使用资源，是权重为2的用户的2倍



添加条件：一个同学会问多个问题



老师66.6%的时间都
会回答我的问题

A要赖！我们应该平
分老师的时间！

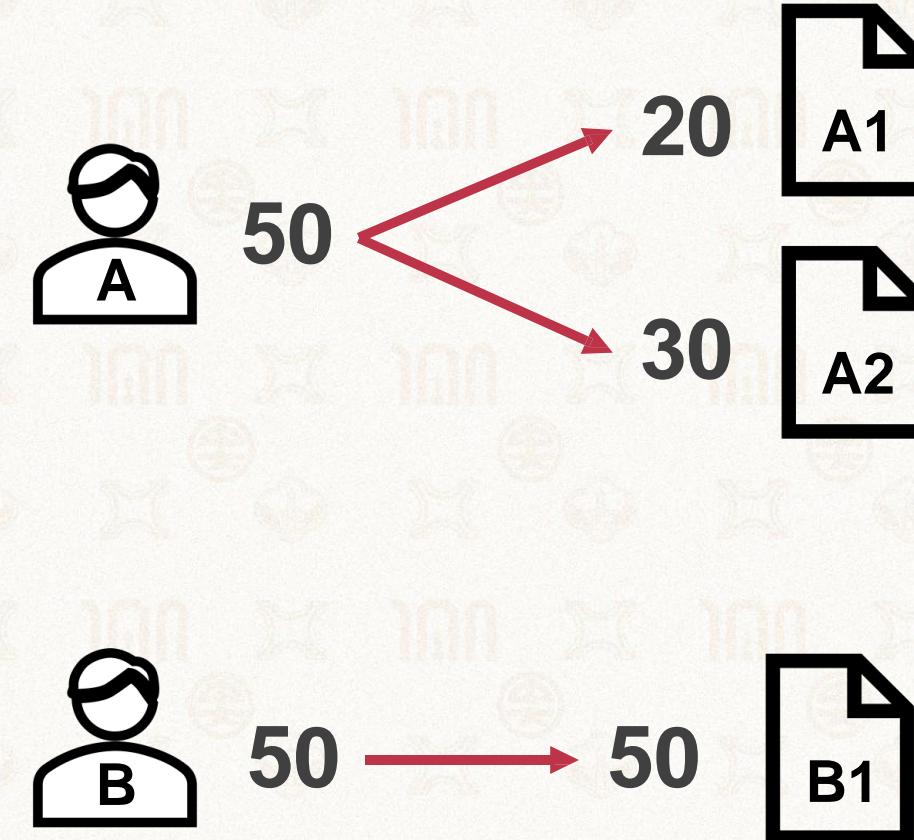


方法：使用"ticket"表示任务的权重

- ticket: 每个问题对应的权重
- T: ticket的总量
- 问题A1可占用老师时间的比例：

- $\frac{\text{ticket}_{A1}}{T} = \frac{20}{100} = \frac{1}{5}$

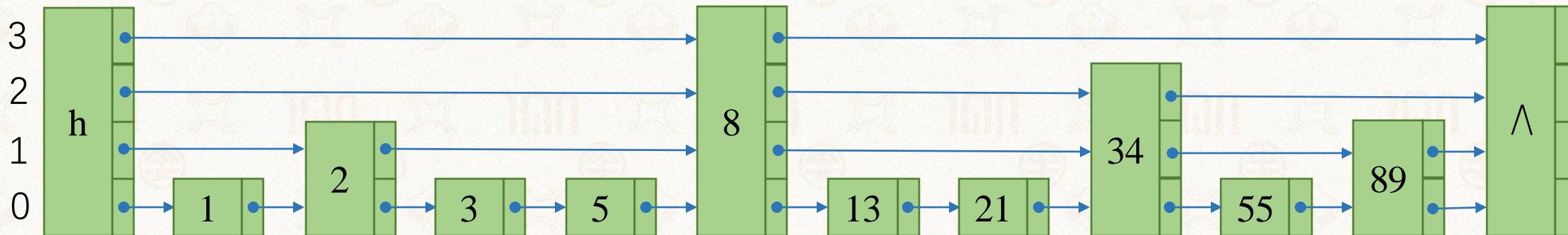
- 同学A可占用老师时间的比例
 - $\frac{\text{ticket}_A}{T} = \frac{\text{ticket}_{A1} + \text{ticket}_{A2}}{T} = \frac{50}{100} = \frac{1}{2}$





一种公平共享的实现：彩票调度

➤ 跳表级数的确定方式

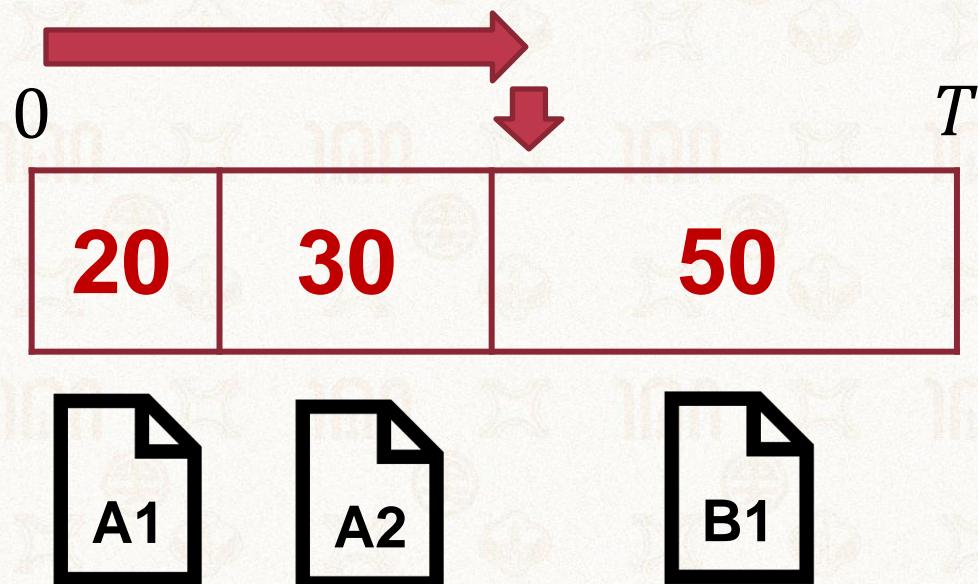


```
int SkipList::randomLevel() {
    float r = (float)rand()/RAND_MAX; // 随机生成一个[0,1]的浮点数
    int lvl = 0;
    while(r < P && lvl < MAXLVL) { // 随机数小于预定义的阈值P，被选中了
        lvl++;
        r = (float)rand()/RAND_MAX; // 继续随机生成一个[0,1]的浮点数
    }
    return lvl; // 什么时候返回：运气不好，冲击更高级失败，或已到满级
};
```



一种公平共享的实现：彩票调度

- 每次调度时，生成随机数 $R \in [0, T)$
- 根据R，找到对应的任务
 - 例如： $R = 51$ ，就去调度B1
- 彩票组织成链表的形式



```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task->ticket
    if (R < sum) {
        schedule()
        break
    }
}
```



彩票调度：彩票转让

- 场景：
 - 在通信过程中，客户端需要等到服务端返回才能继续执行
- 客户端将自己所有的ticket移交给服务端
 - 确保服务端可以尽可能使用更多资源，迅速处理
- 同样适用于其他同步场景





权重与优先的异同?

➤ 权重影响任务对CPU的占用比例

- 永远不会有任务饿死

➤ 优先级影响任务对CPU的使用顺序

- 可能产生饿死



随机的利弊

➤ 随机的好处是？

- 简单

➤ 随机带来的问题是？

- 不精确——伪随机非真随机
- 各个任务对CPU时间的占比会有误差
- 存在不确定性



步幅调度

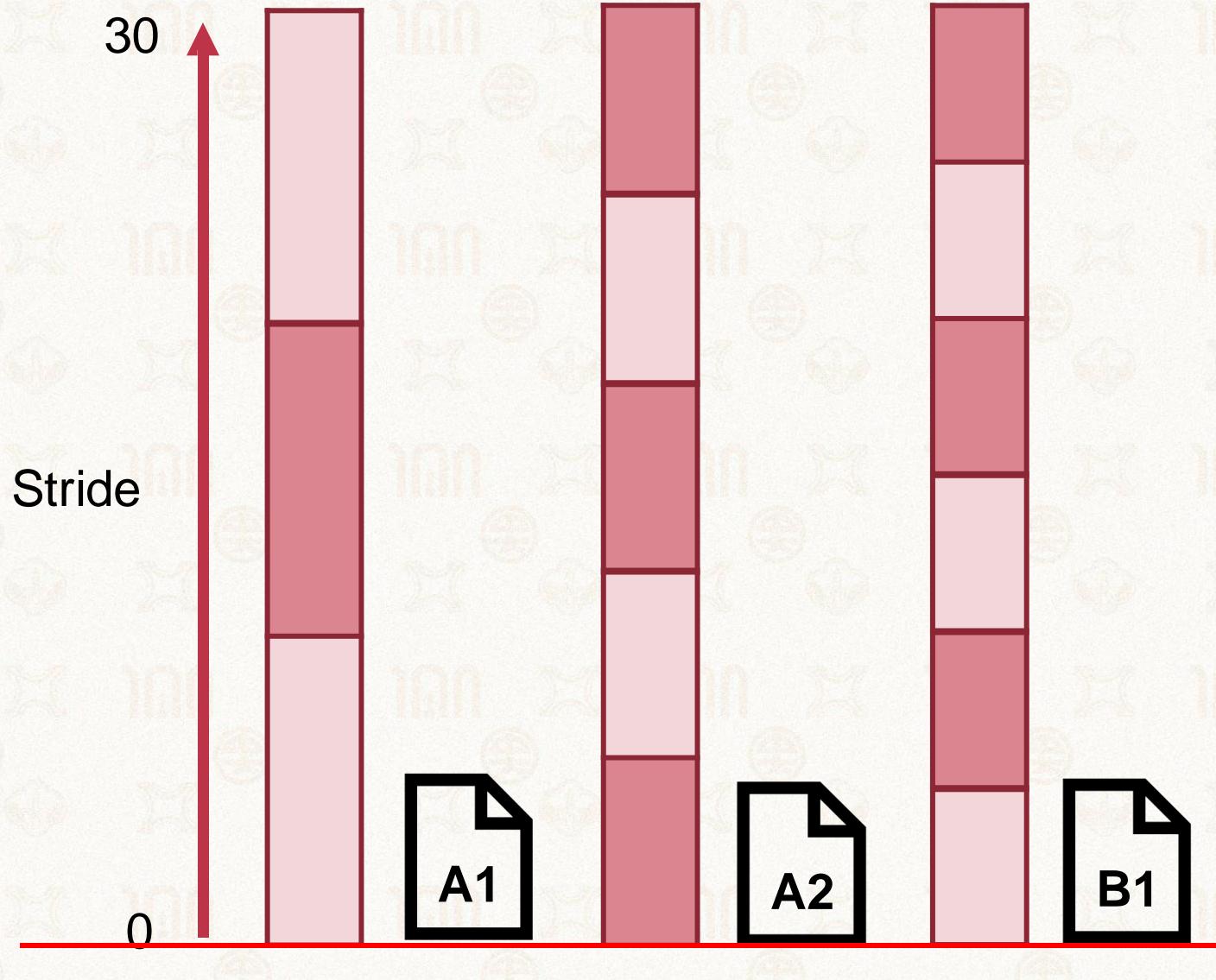
- 确定性版本的彩票调度
 - 可以沿用tickets的概念
- Stride——步幅，任务一次执行增加的虚拟时间
 - $\text{stride} = \frac{\text{MaxStride}}{\text{ticket}}$
 - MaxStride是一个足够大的整数
 - 设为所有tickets的最小公倍数
- Pass——累计执行的虚拟时间

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5

MaxStride=300



步幅调度

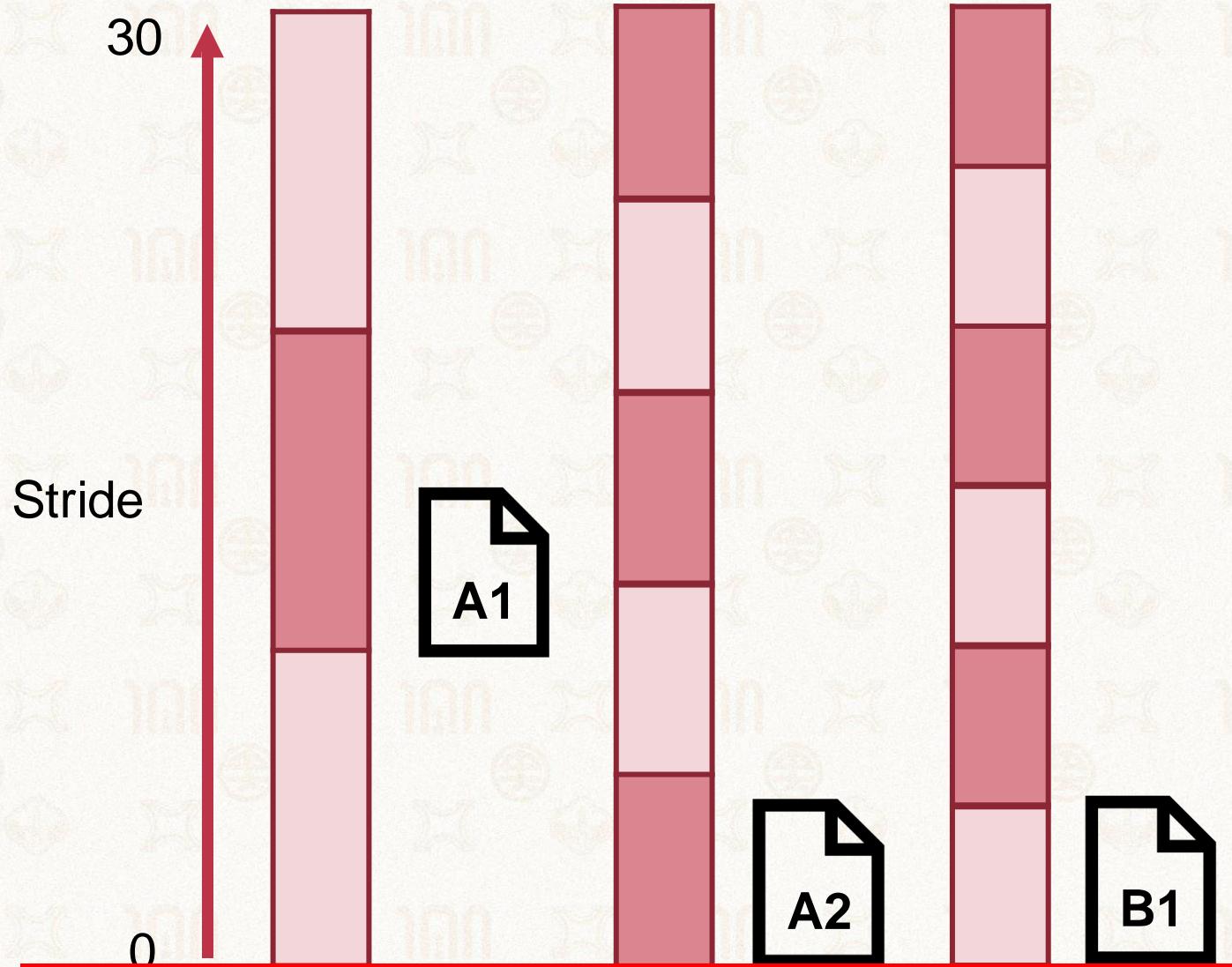


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

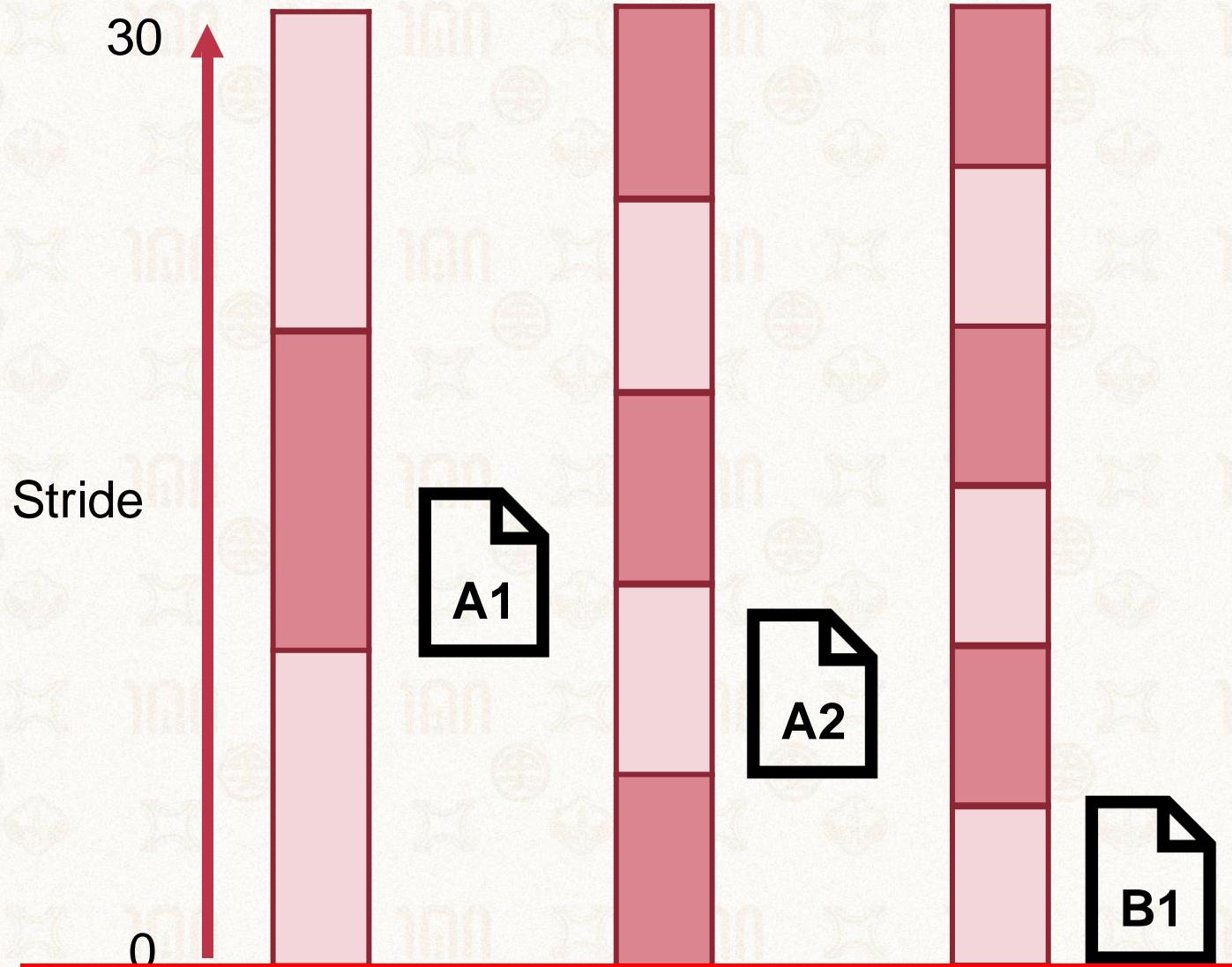


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

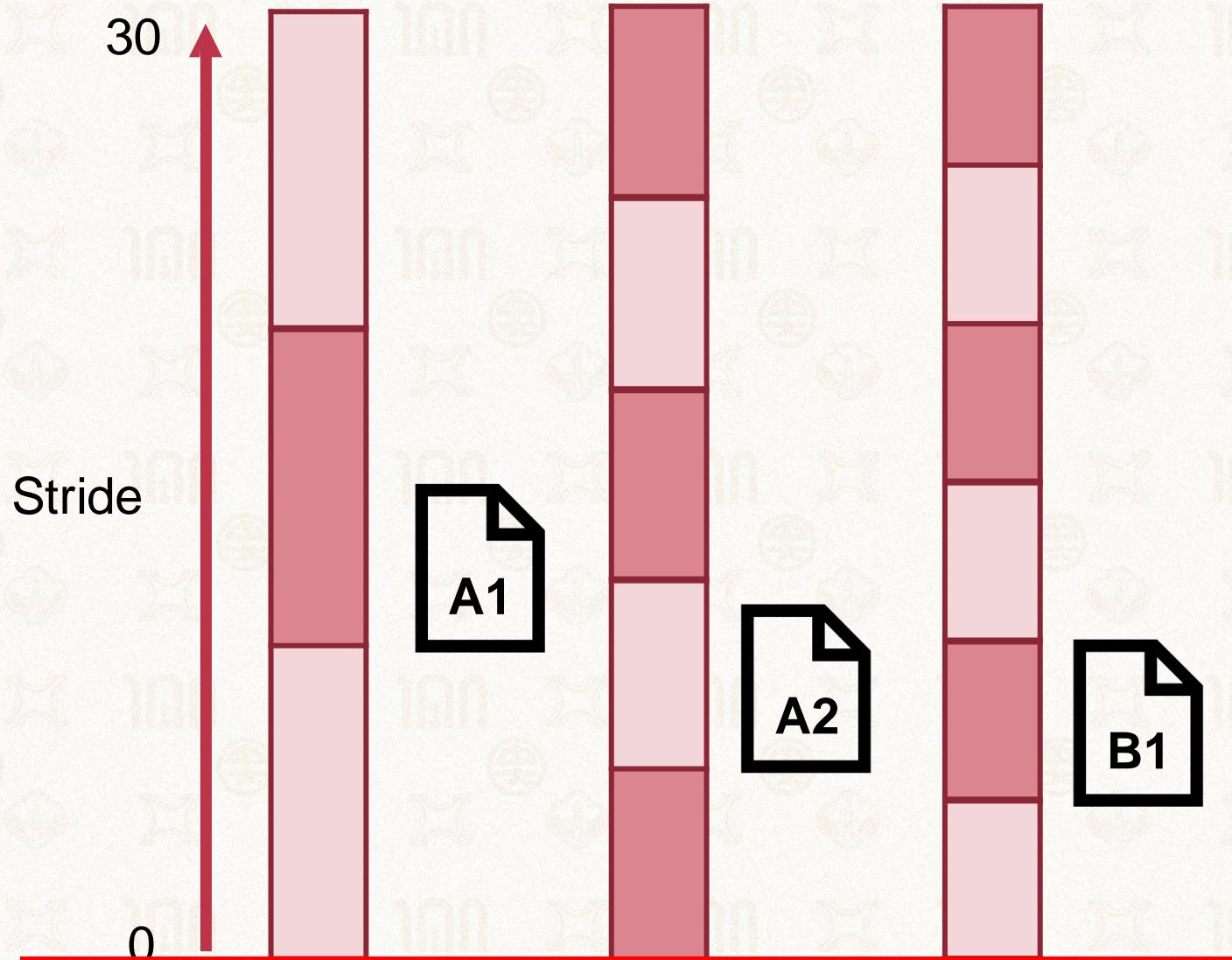


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

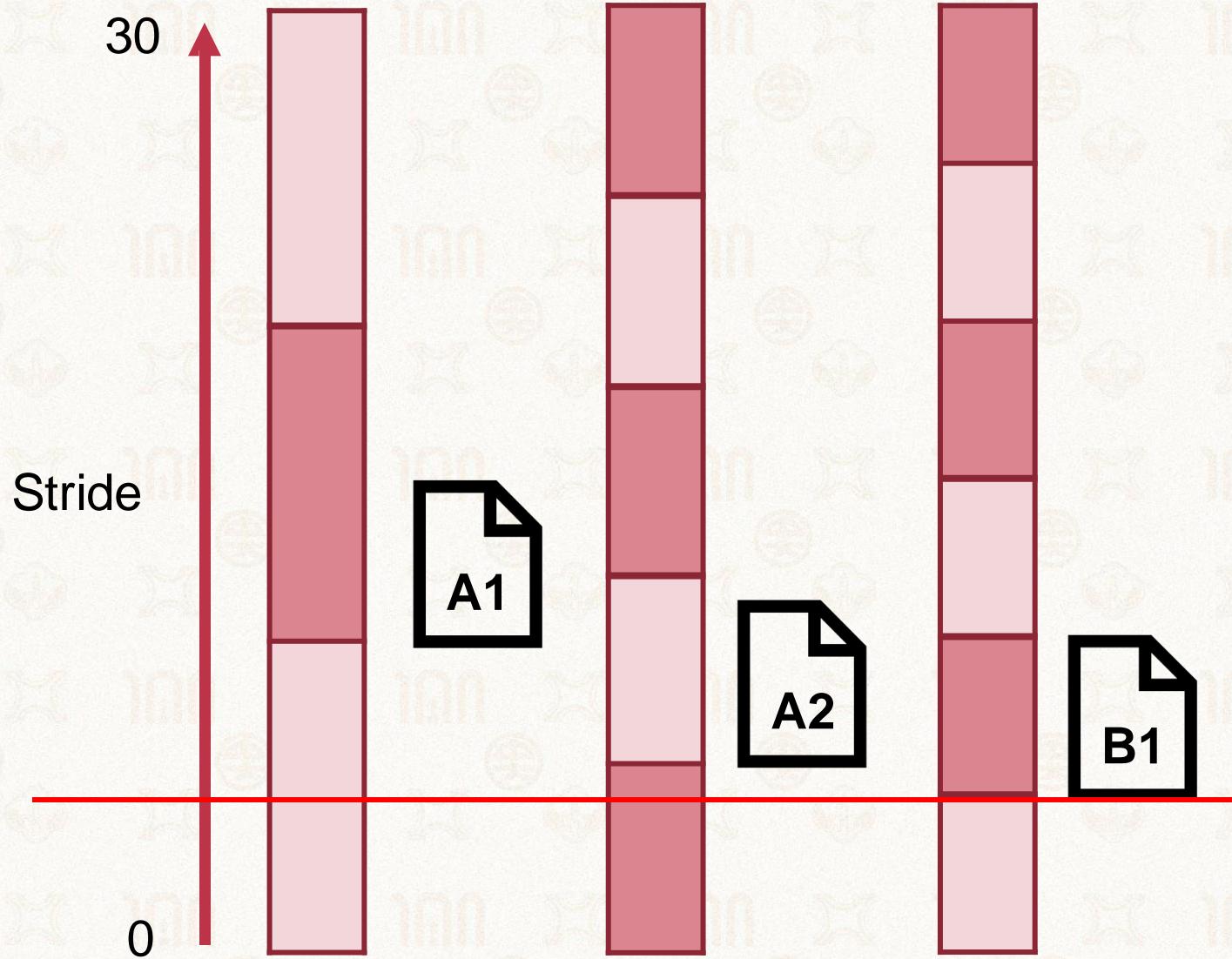


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

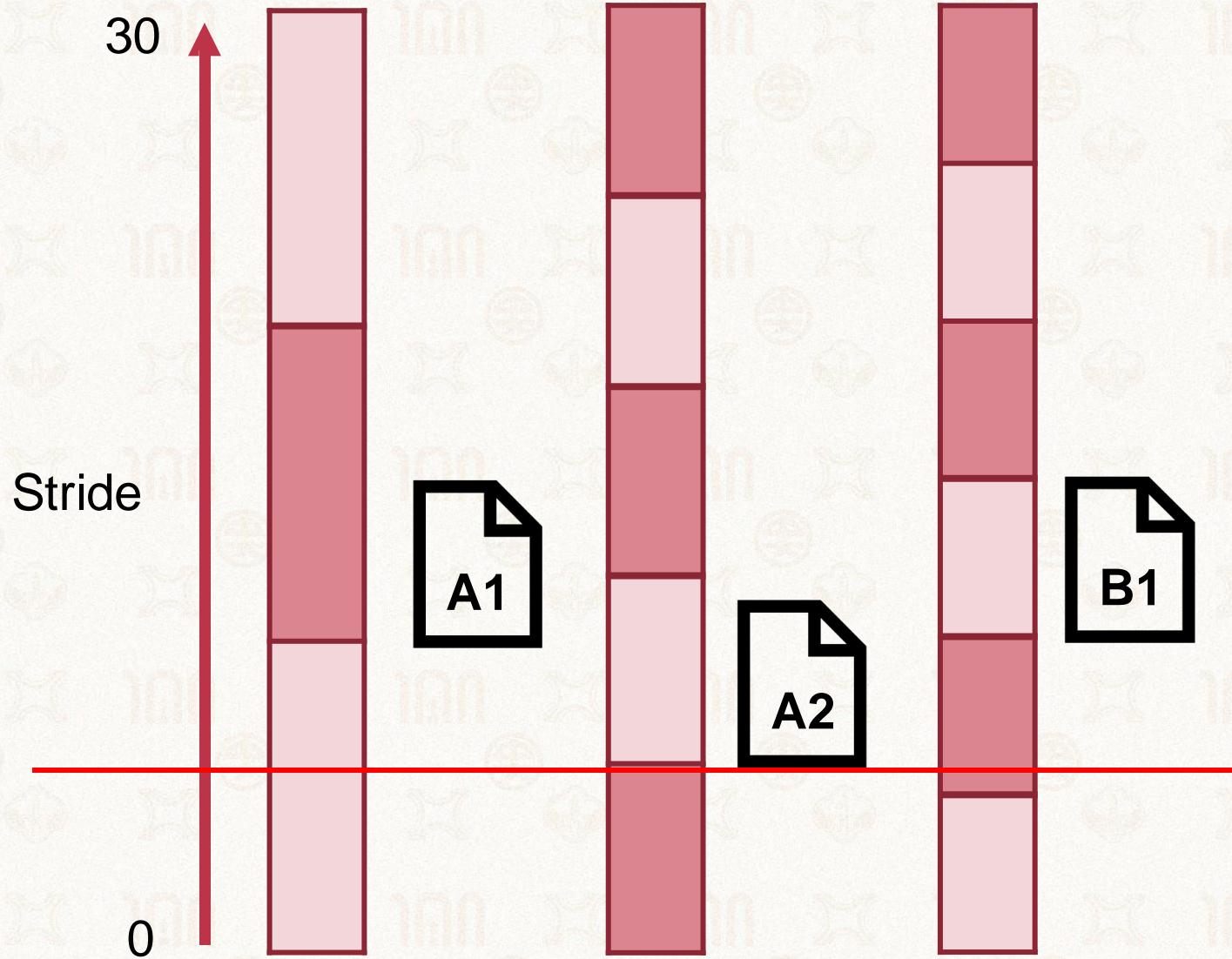


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

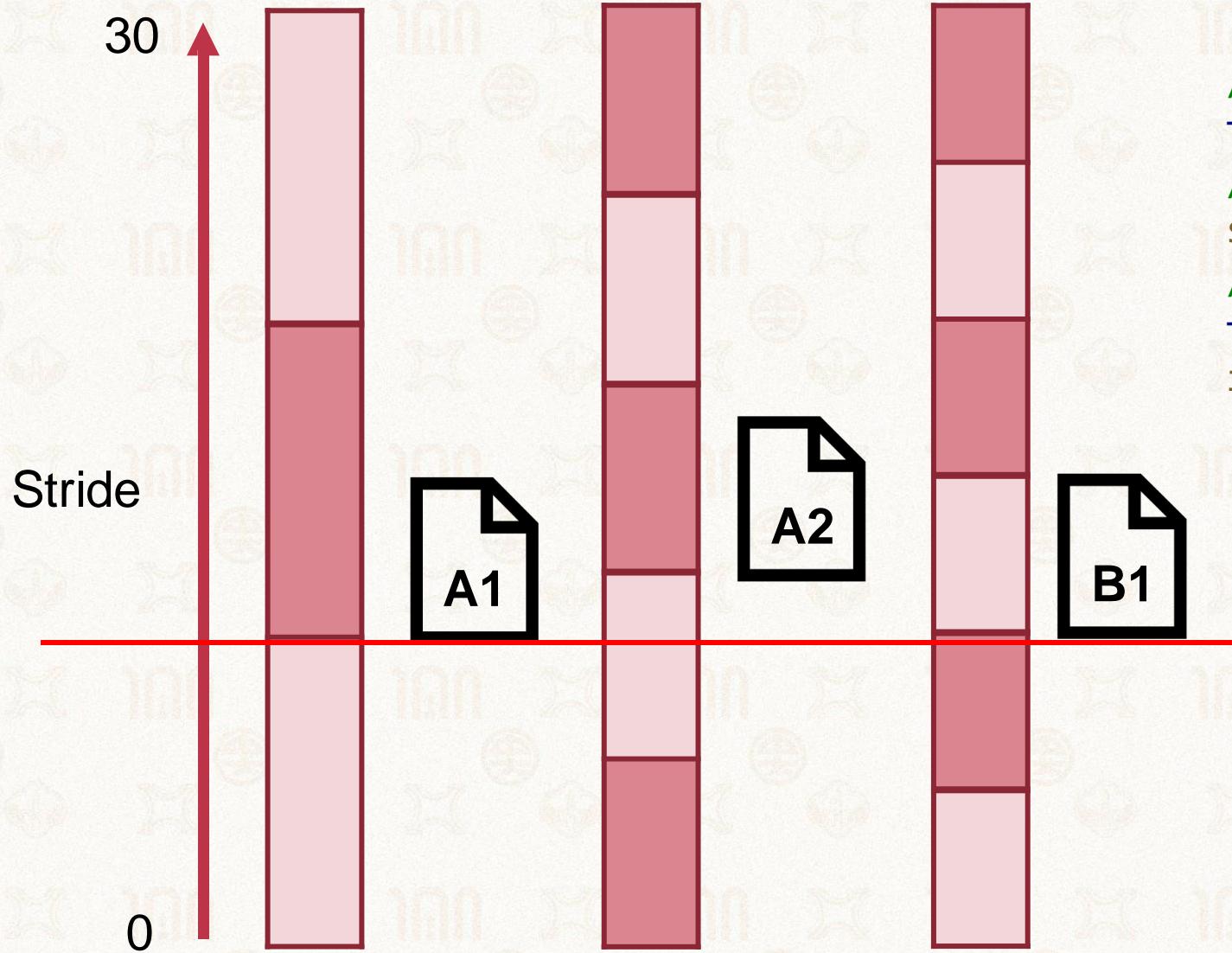


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

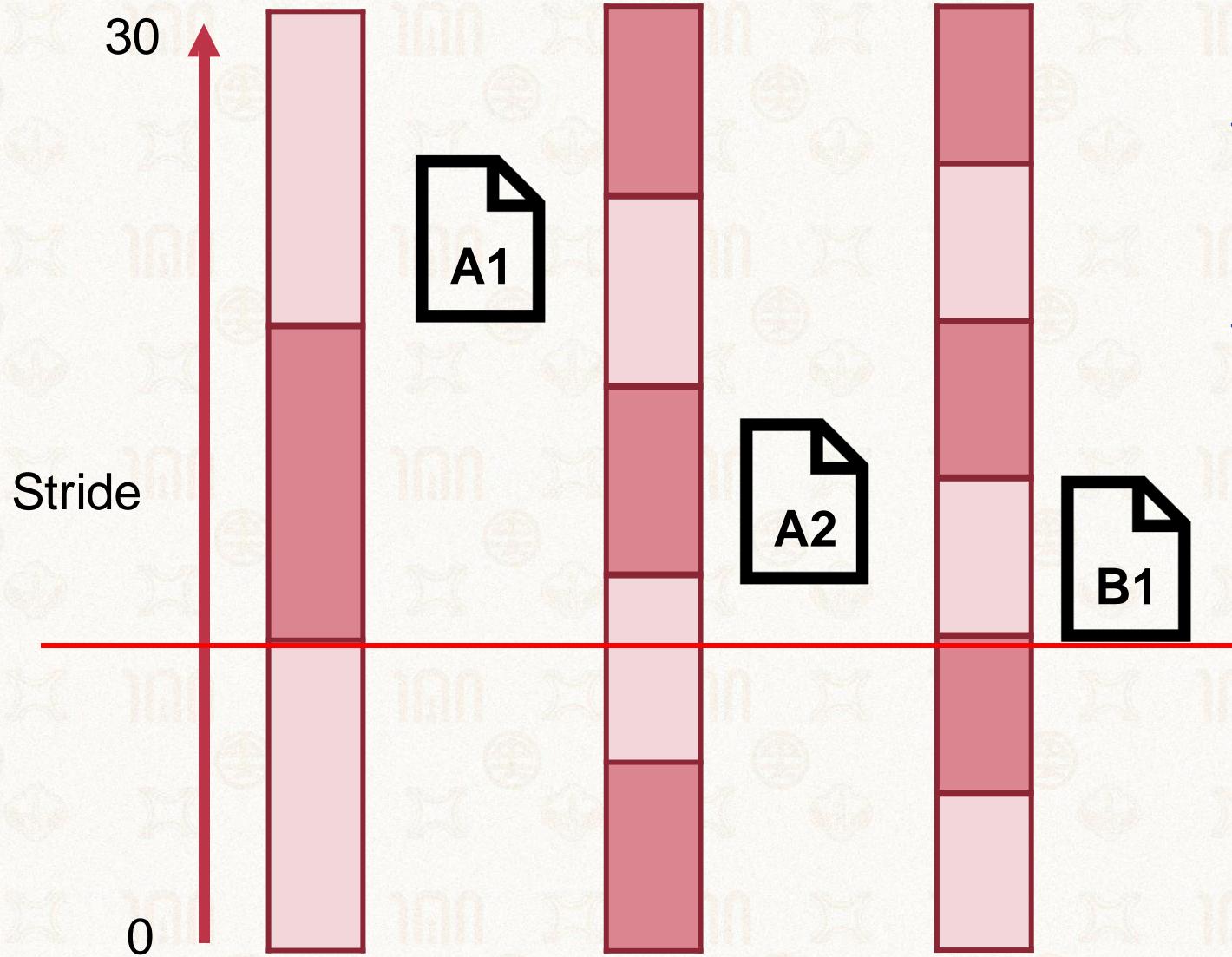


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

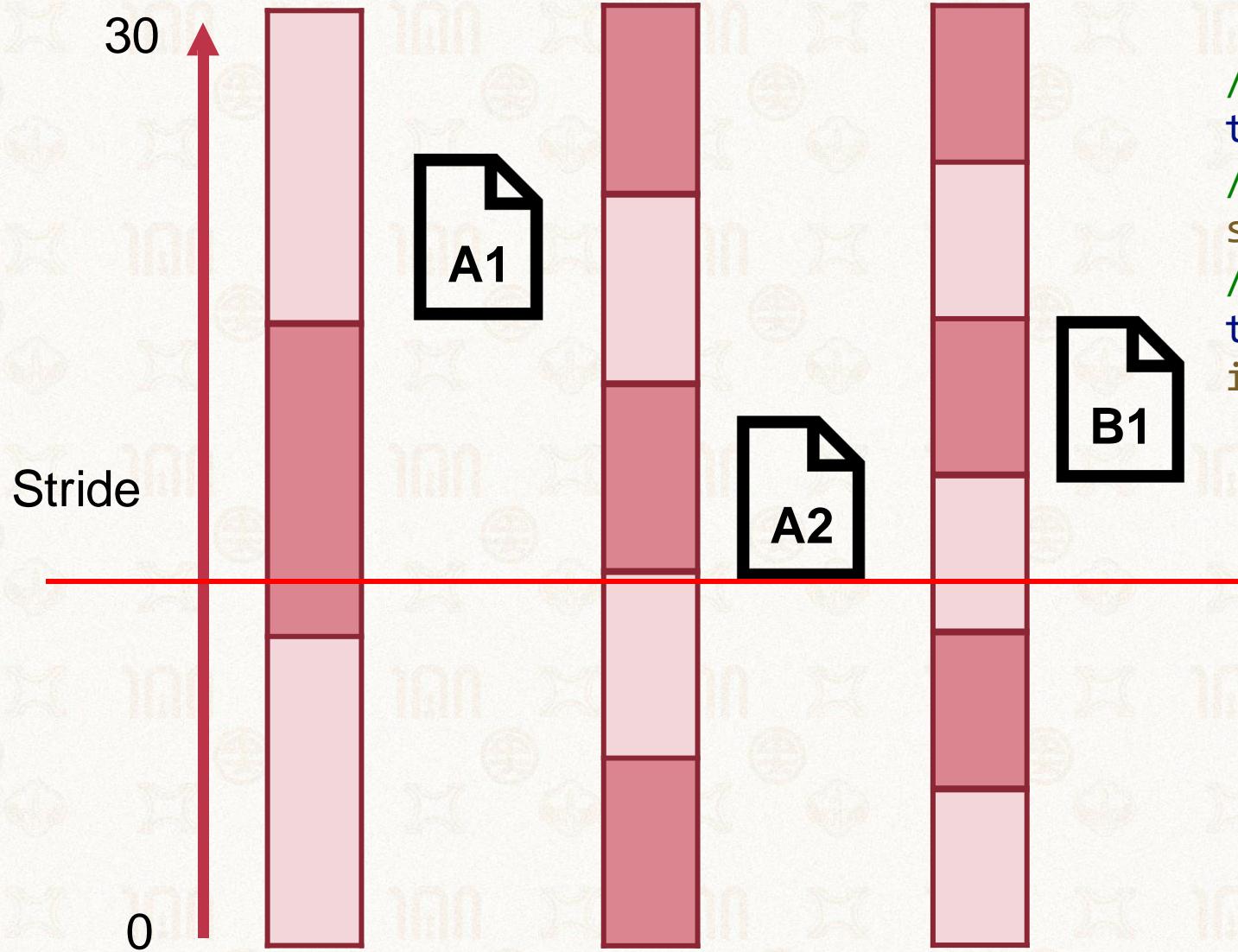


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

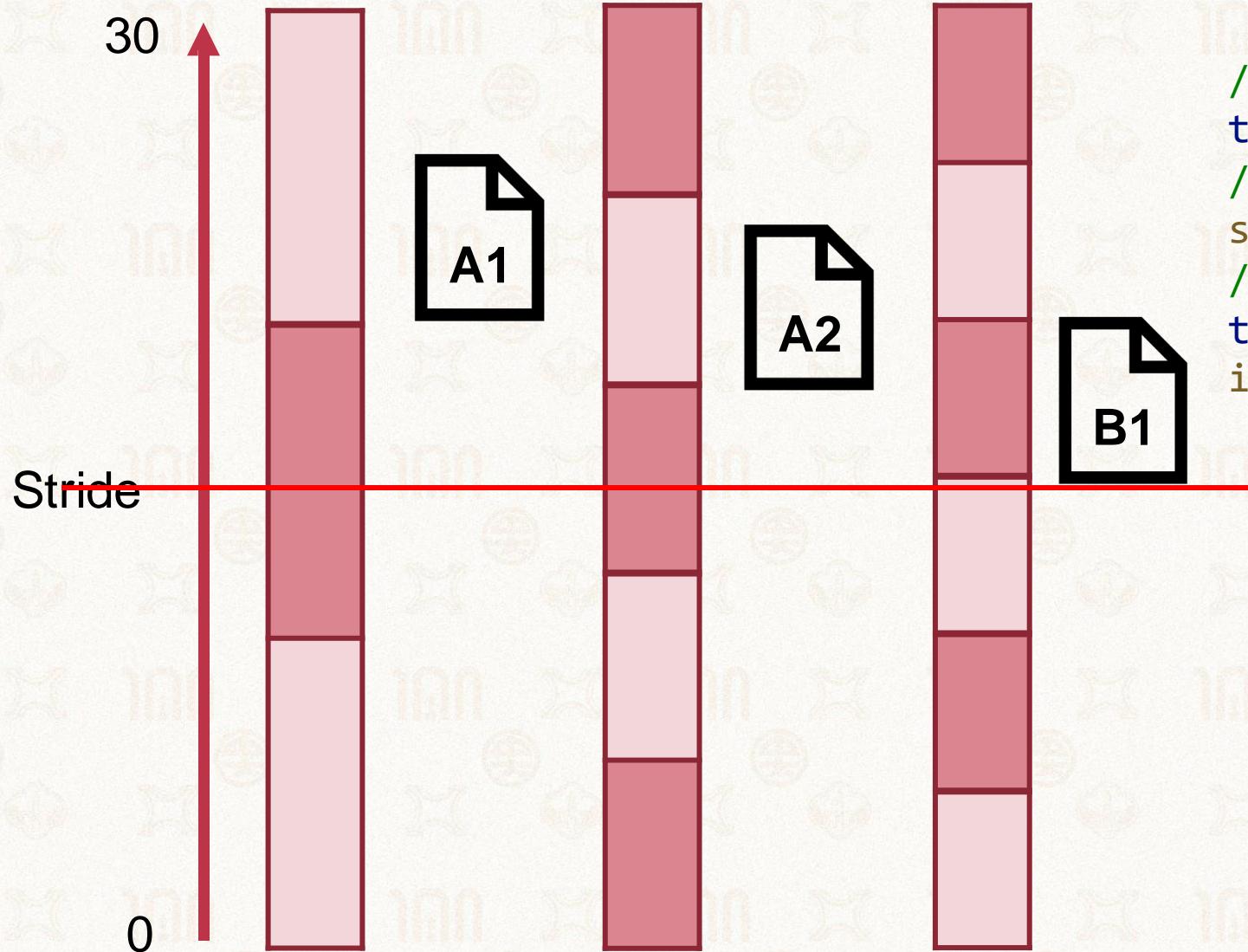


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

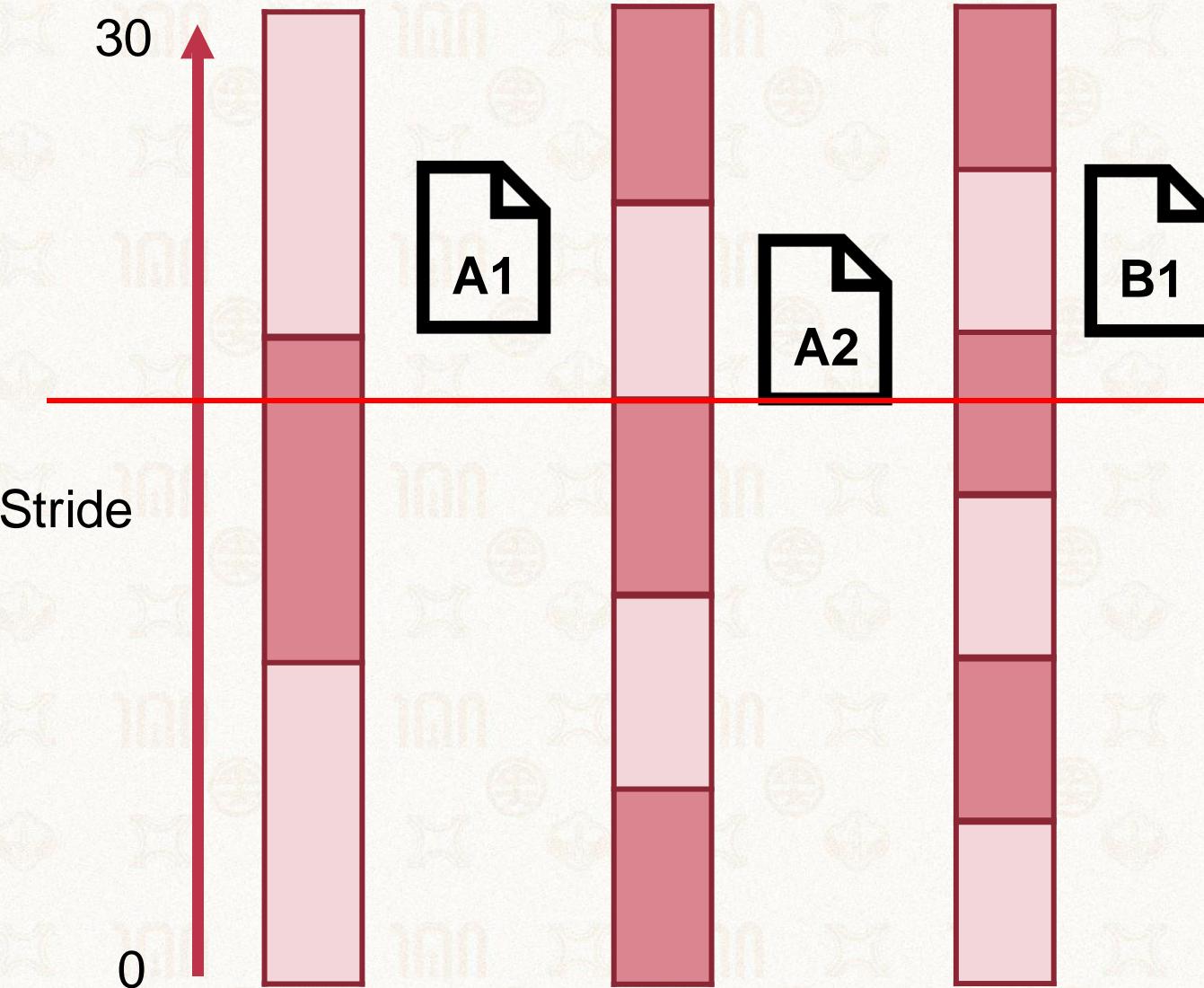


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

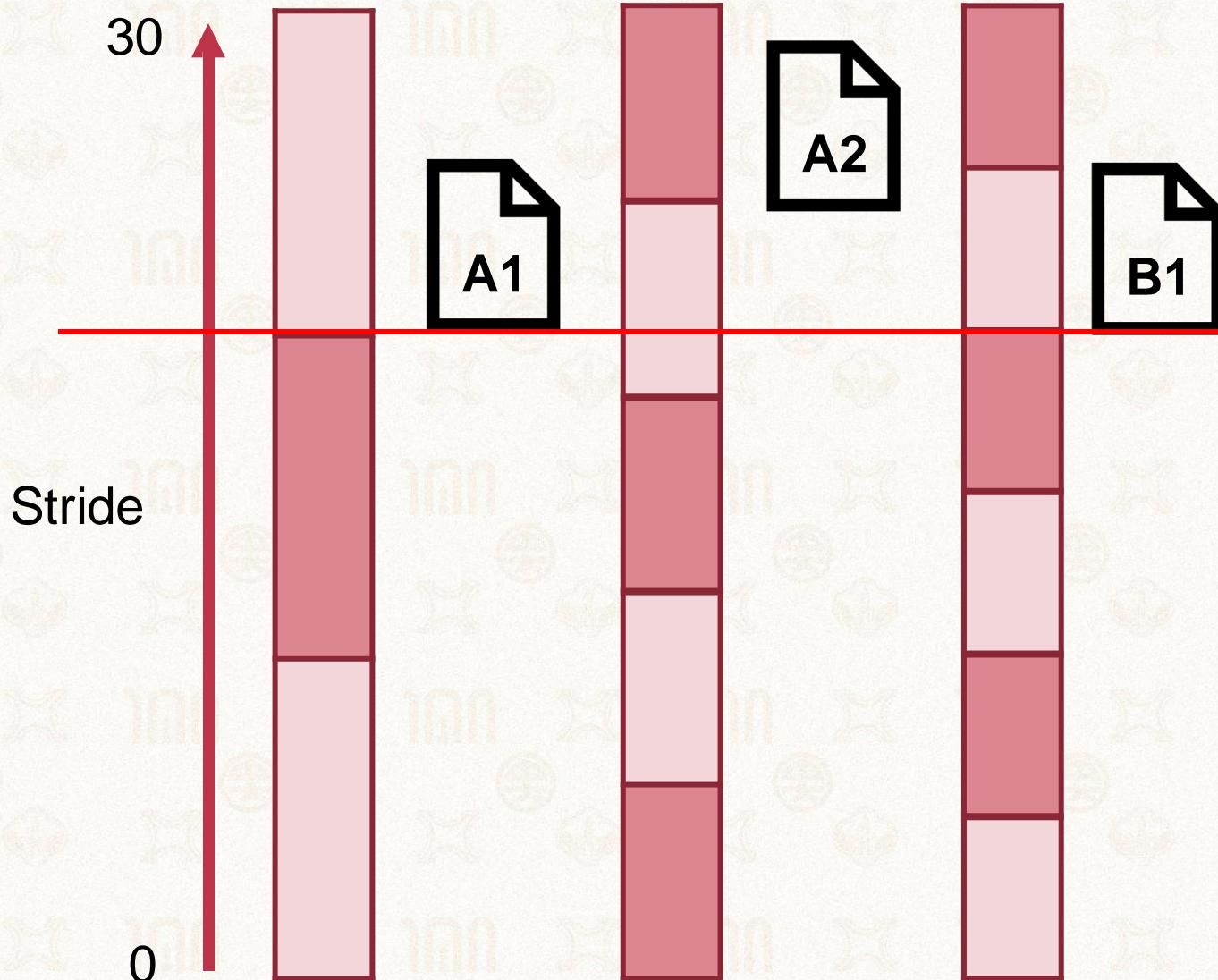


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度



```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度



Stride

0



```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

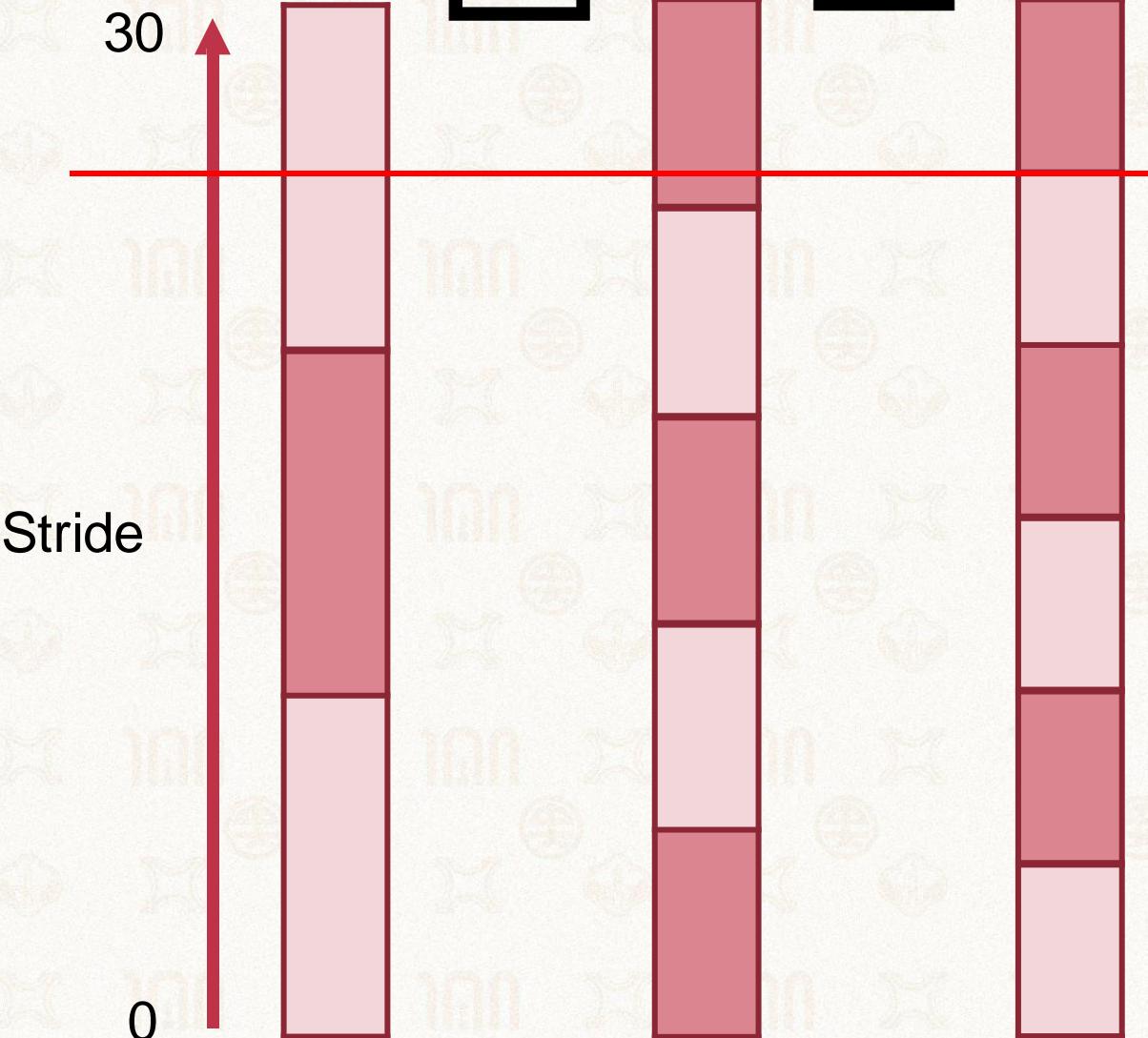
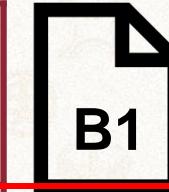


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度

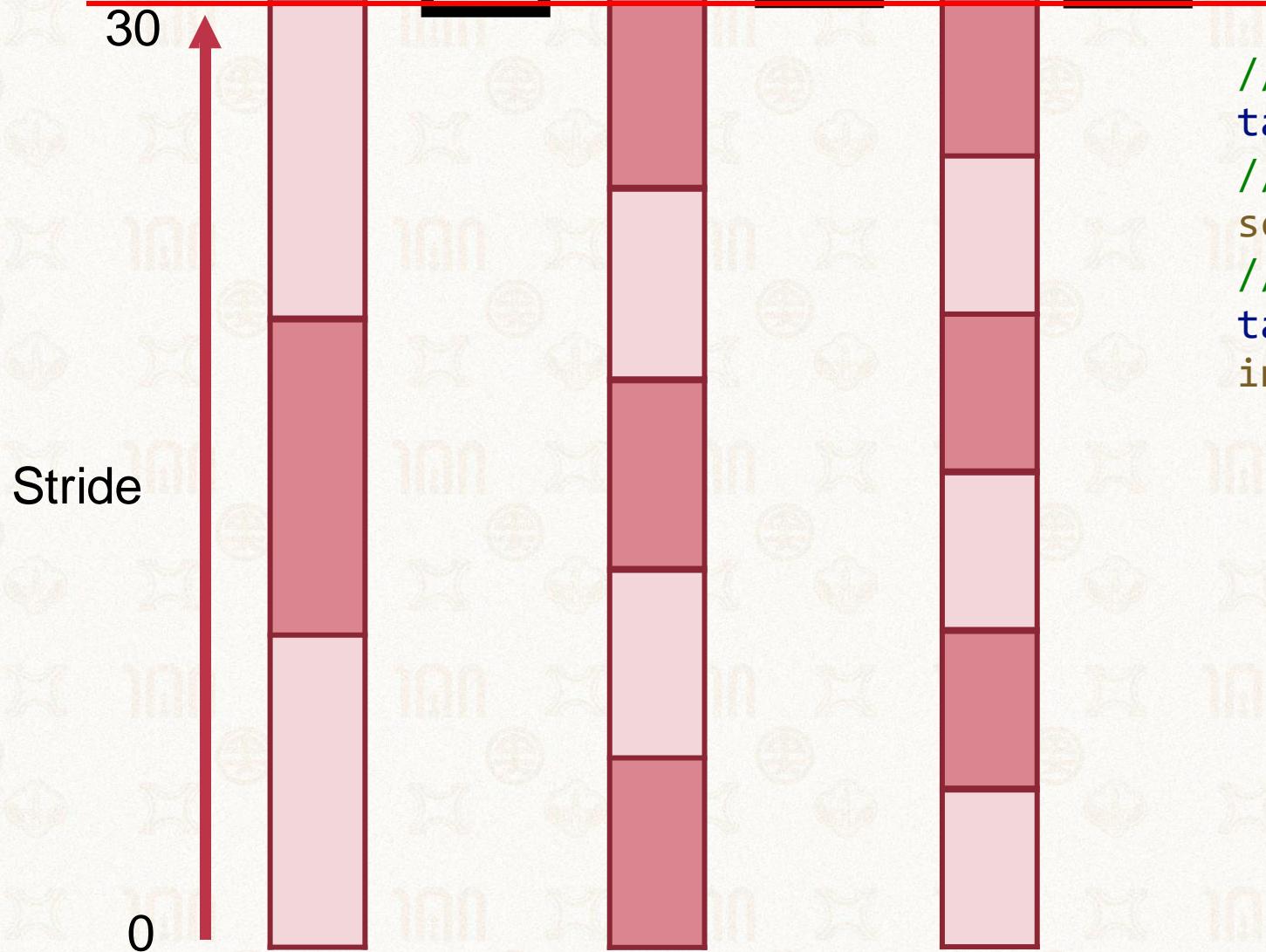
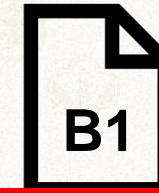


```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



步幅调度



```
// 选择并移除运行队列中虚拟时间最小的任务  
task = remove_queue_min(q);  
// 调度该任务并让其执行一个时间片  
schedule(task);  
// 使用该任务的步幅计算调度后的虚拟时间  
task->pass += task->stride;  
insert_queue(q, current);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5



公平共享调度

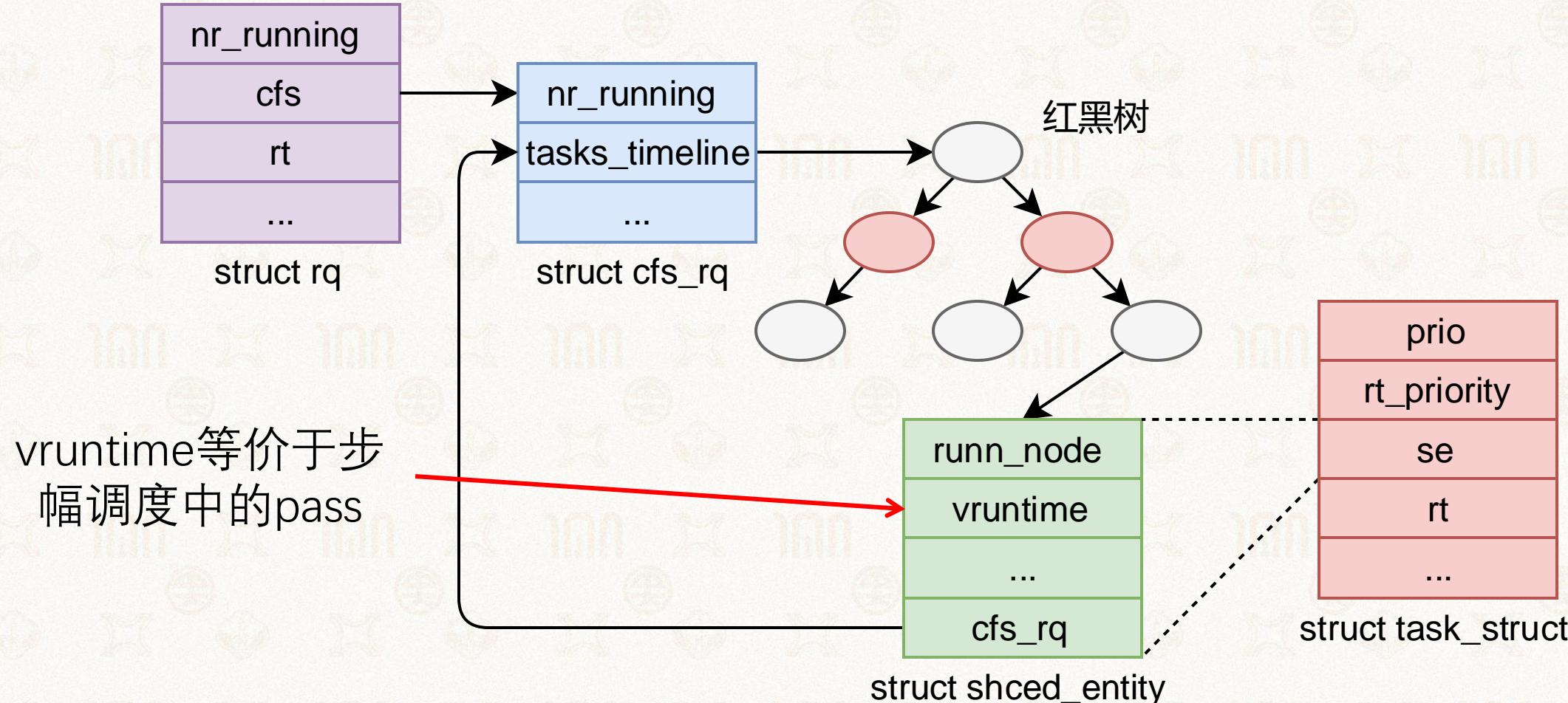
- 预期——根据任务权重计算的执行时间期望

	彩票调度	步幅调度
调度决策生成	随机	确定性计算
任务实际执行时间 与预期的差距	大	小



Linux调度机制：公平调度器运行队列

- 公平调度器(CFS): 使用类似步幅调度的公平共享调度策略





大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



实时调度

- 每个任务都有截止时间(Deadline)
- 软实时(Soft Real Time)
 - 视频播放，每一帧的渲染
 - 超过截止时间
 - 画质差
- 硬实时(Hard Real Time)
 - 自动驾驶汽车的刹车任务
 - 方向盘电子助力系统
 - 超过截止时间
 - 严重后果

速度 (千米/小时)	速度 (米 / 秒)	停车距离(米) 干地	停车距离(米) 潮地	停车距离(米) 雪地
60	16.67	17.15	25.72	51.44
90	25.00	38.58	57.87	115.74
120	33.33	68.59	102.88	205.76
150	41.67	107.17	160.75	321.50

不同条件下的刹车距离



普通系统很难做到确定性时延

```
os@ubuntu:~$ sysbench --test=cpu --threads=30 run
WARNING: the --test option is deprecated. You can pass a script name or path
on the command line without any options.
sysbench 1.0.18 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 30
Initializing random number generator from current time

Prime numbers limit: 10000

Initializing worker threads...
Threads started!

CPU speed:
events per second: 7923.47

General statistics:
total time: 10.0020s
total number of events: 79259

Latency (ms):
min: 0.23
avg: 3.76
max: 104.26
95th percentile: 51.94
sum: 298308.20

Threads fairness:
events (avg/stddev): 2641.9667/173.39
execution time (avg/stddev): 9.9436/0.04

sysbench --test=cpu --threads=30 run
```

➤ 调度时延:

- 最小时延0.23毫秒
- 最大时延104.26毫秒

➤ 相差太大，且最大时延也不稳定

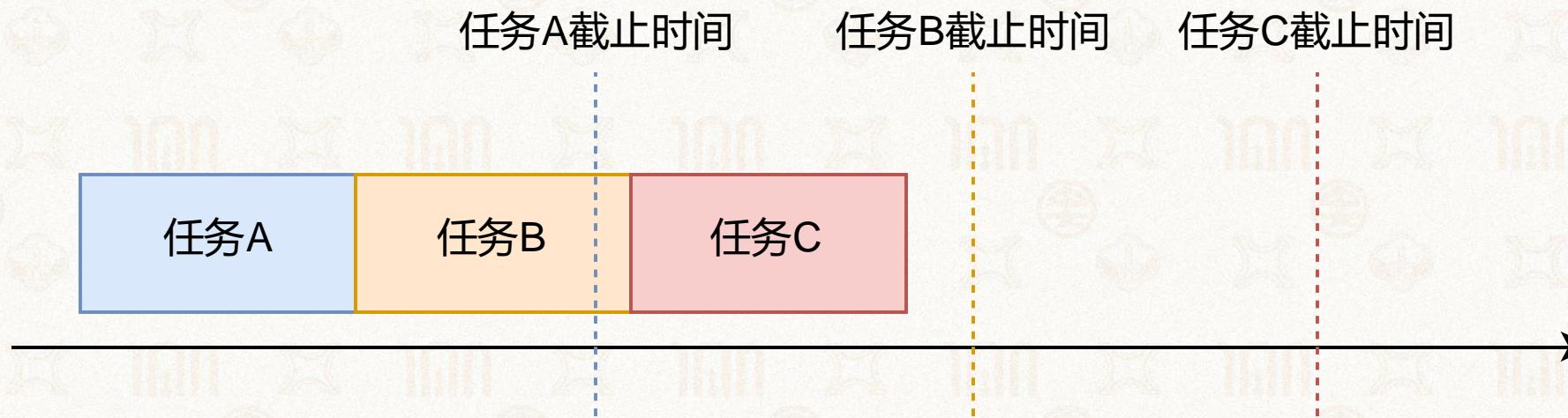
➤ 程序运行时间是稳定的，但调度太不稳定



实时操作系统的特点

➤ 确定性!

- 完成时间有明确上界
- 调度时延可以被准确预测

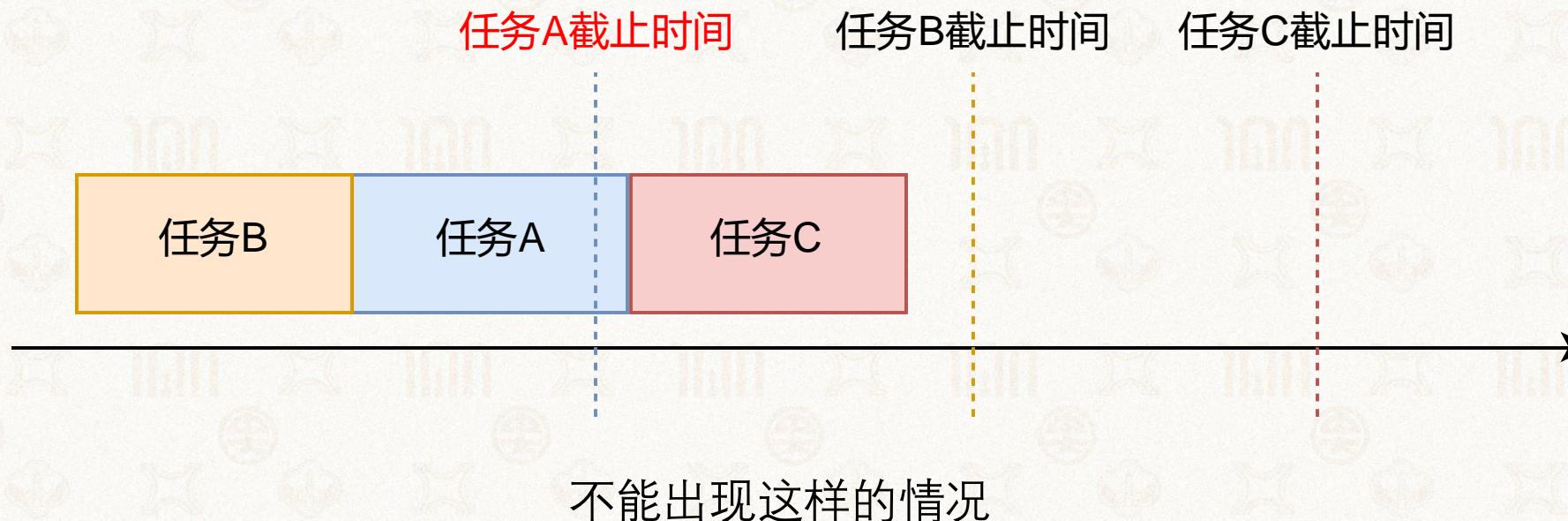




实时操作系统的特点

➤ 确定性!

- 完成时间有明确上界
- 调度时延可以被准确预测





CPU利用率

➤ 简化操作：

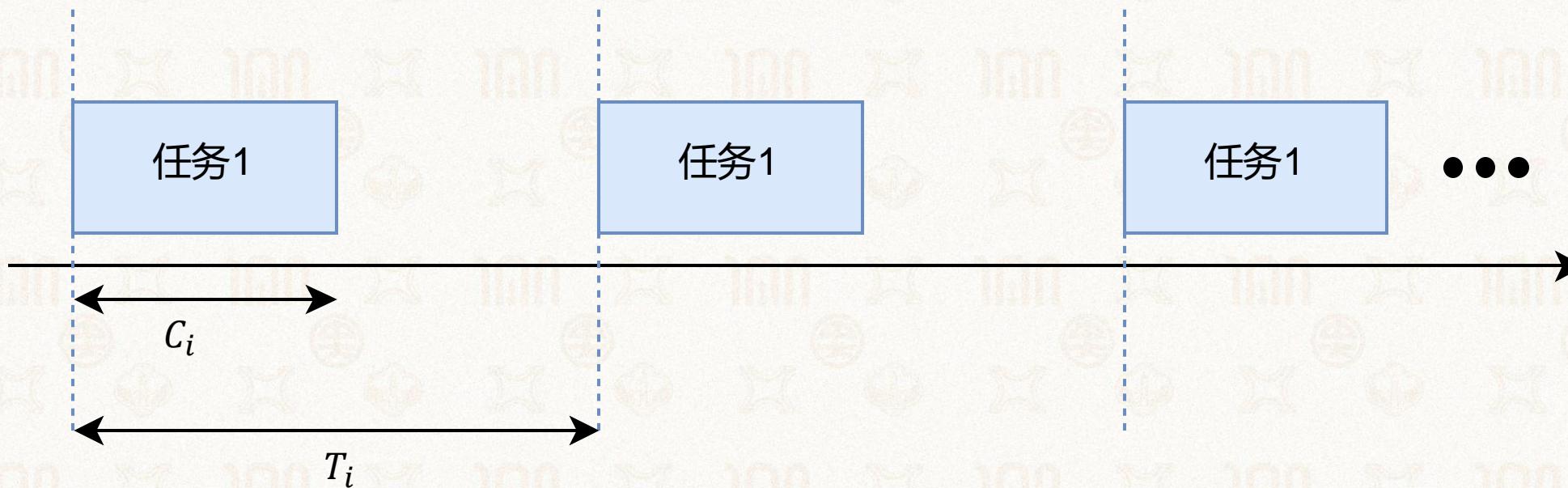
- 只考慮周期任务，且沒有依赖关系
- 周期就是截止时间

➤ CPU利用率：

- 所有任务利用率之和：

$$U = \sum_{i=1}^m C_i / T_i$$

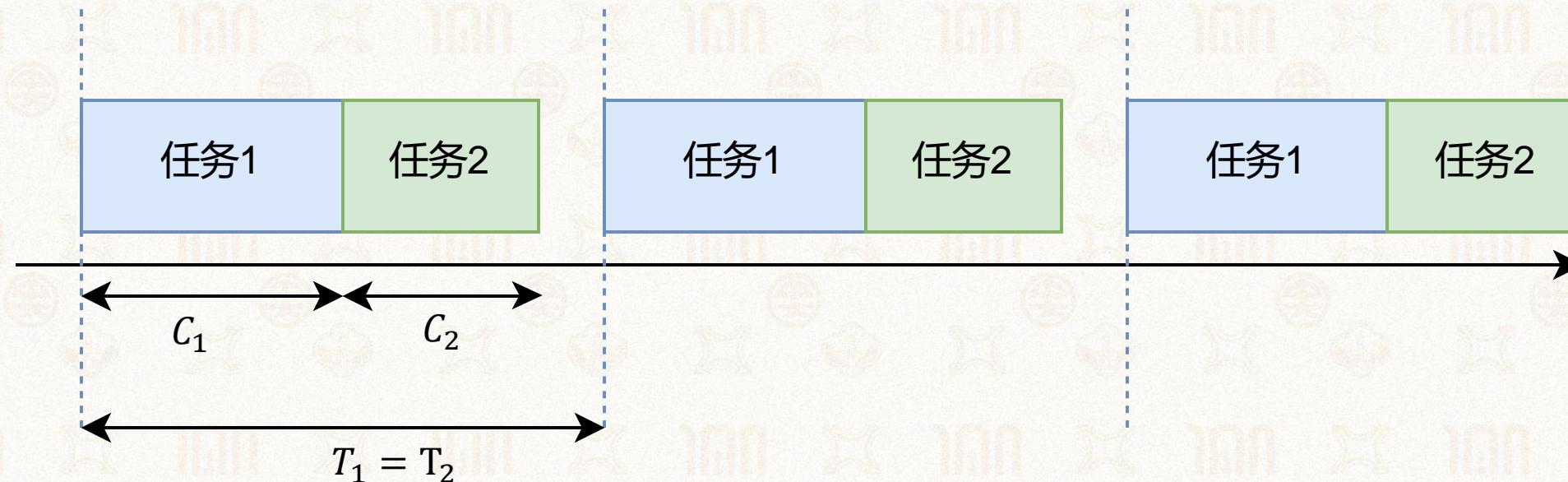
任务1截止时间





CPU利用率

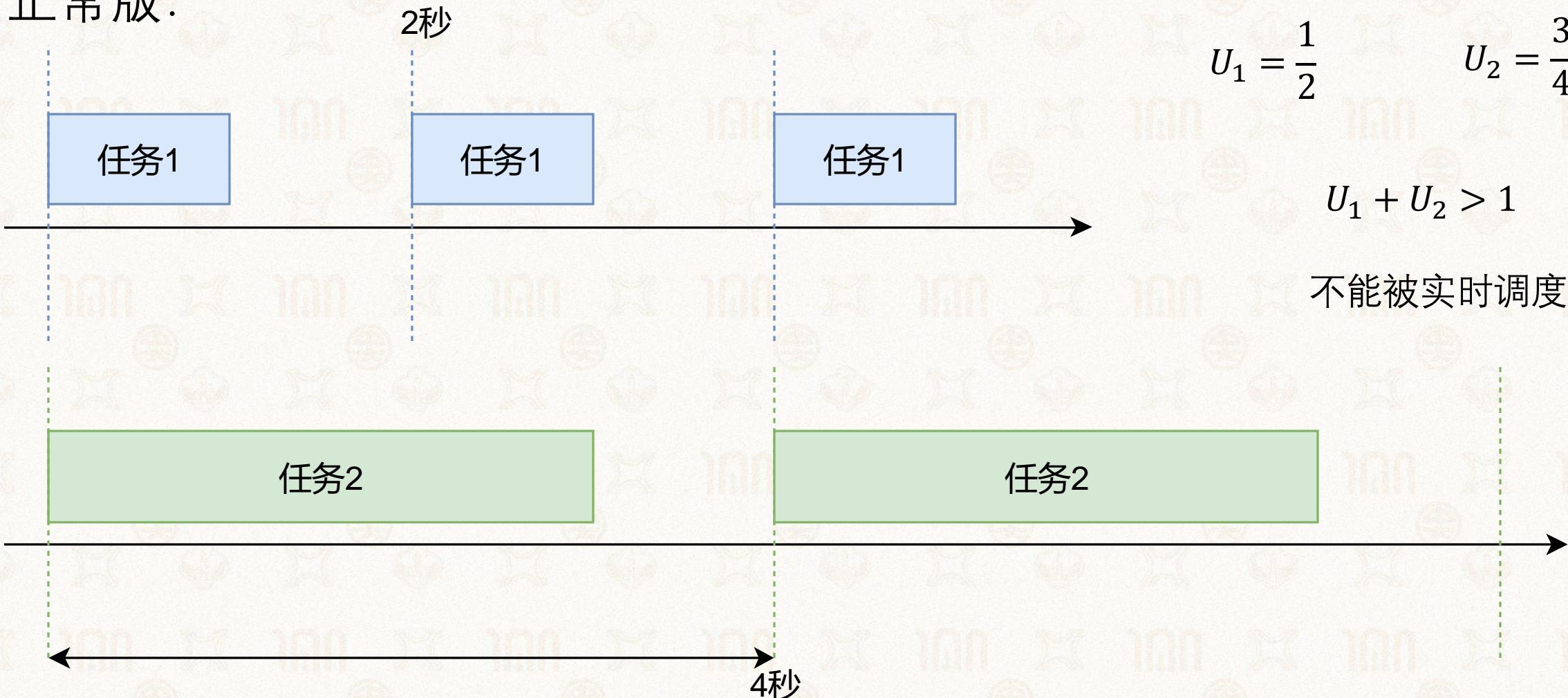
- 满足实时调度要求的必要条件是U一定小于或等于1
- 简化版：





CPU利用率

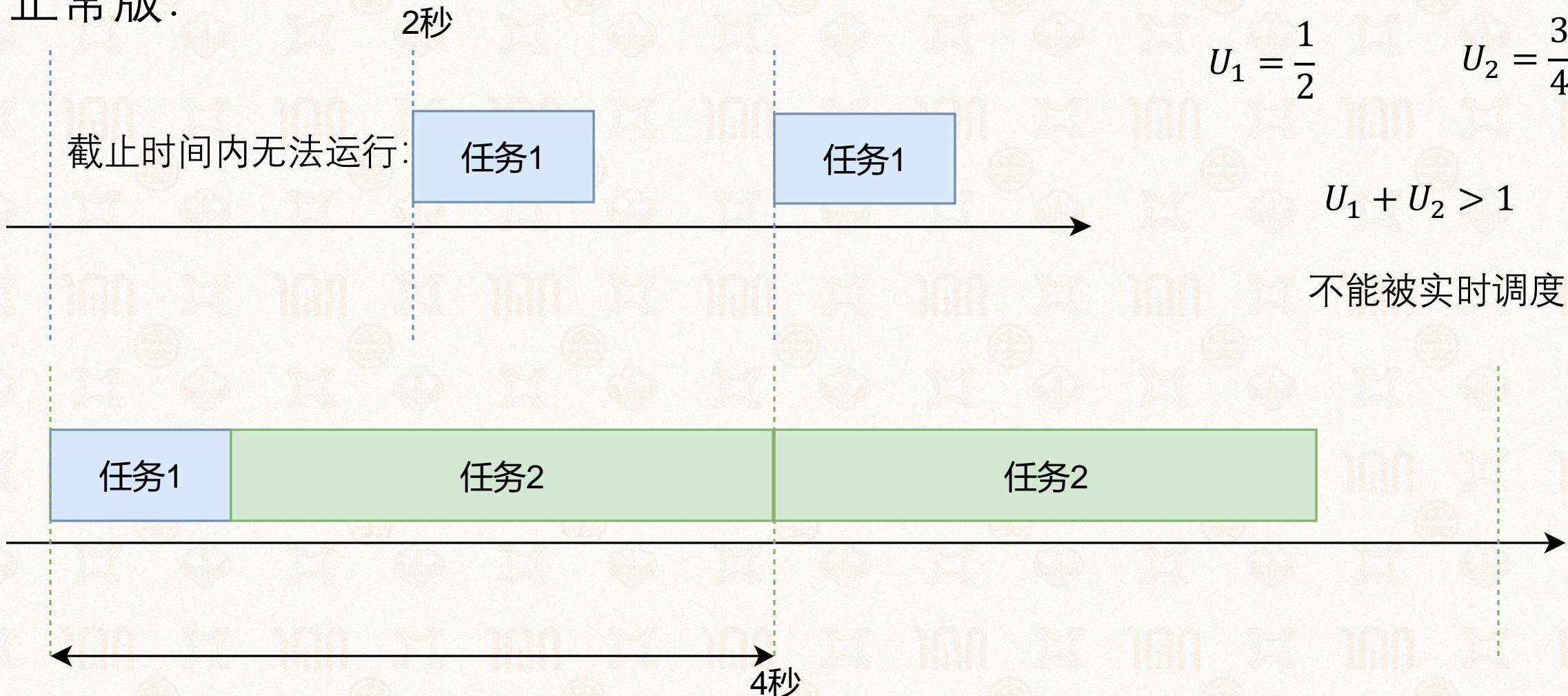
- 满足实时调度要求的必要条件是U一定小于或等于1
- 正常版：





CPU利用率

- 满足实时调度要求的必要条件是U一定小于或等于1
- 正常版：

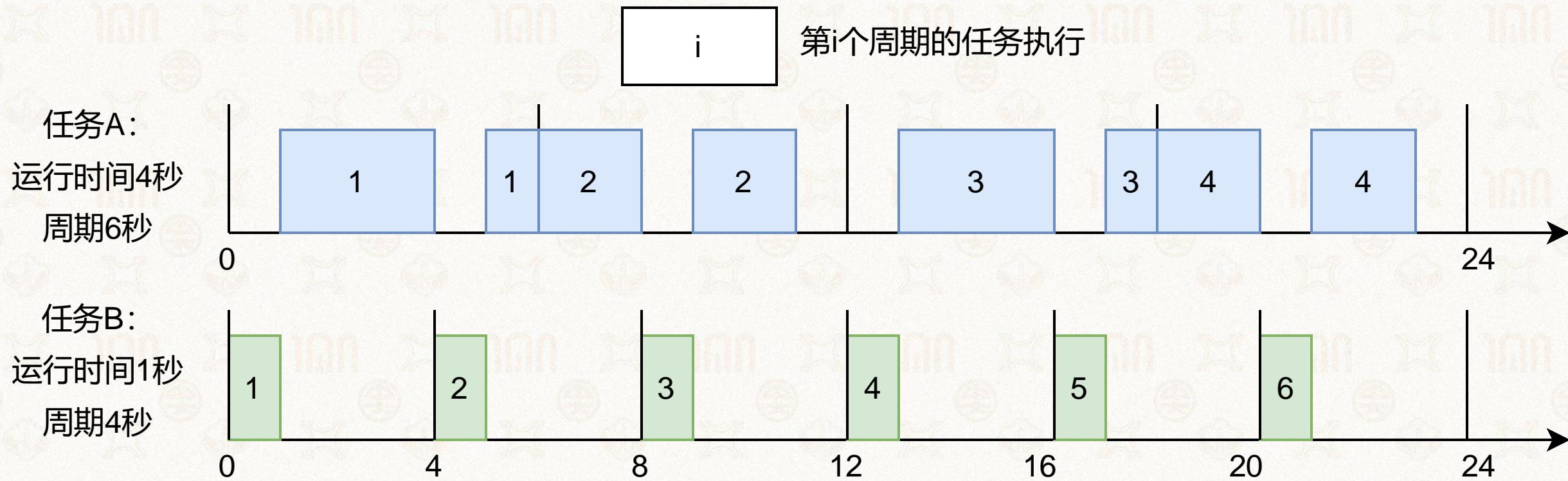




速率单调(Rate-Monotonic, RM)策略

➤ 静态优先级实时调度

- 任务周期越短，优先级越高，应先被调度
 - 任务B的优先级更高，且是抢占式调度



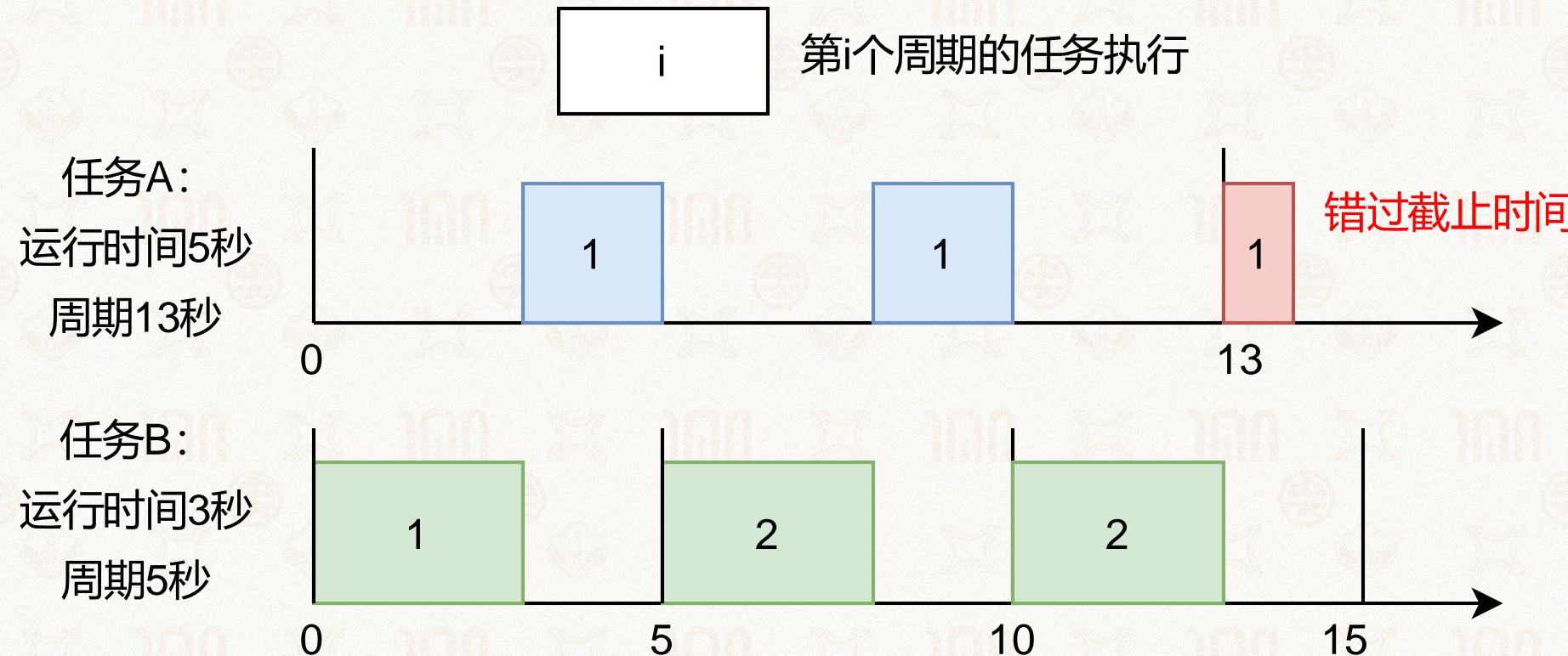


速率单调(Rate-Monotonic, RM)策略

➤ 静态优先级实时调度

- 任务周期越短，优先级越高，应先被调度
 - 任务B的优先级更高，且是抢占式调度

$$U = \frac{5}{13} + \frac{3}{5} = \frac{64}{65} < 1$$





最早截止时间优先(Earliest Deadline First, EDF)

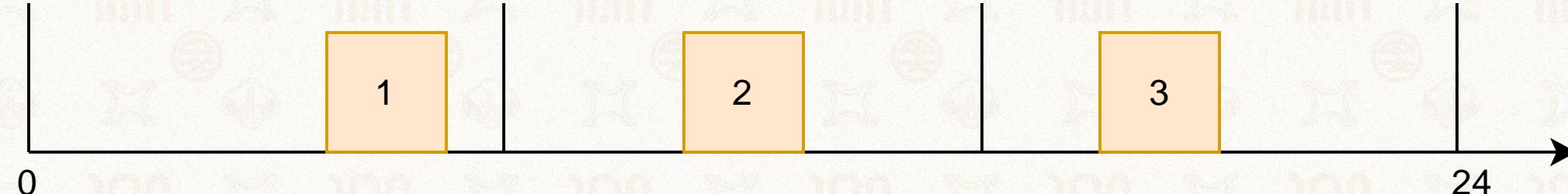
➤ 动态优先级实时调度

- 无需预知执行时间、任务周期

任务A:

运行时间2秒

周期8秒



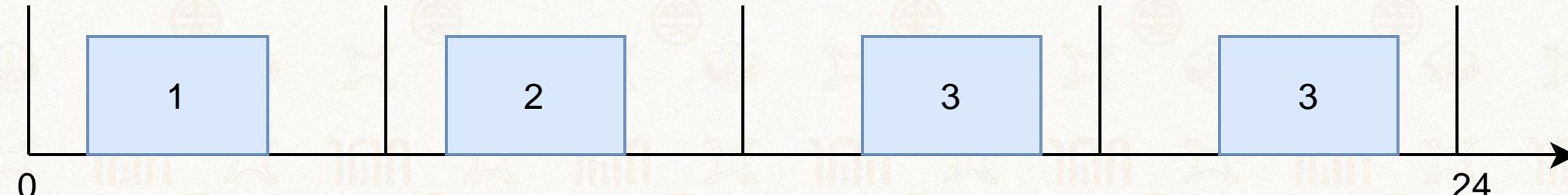
➤ 每次调度截止时间最近的任务

➤ 在任务可调度的情况下能够实现最优调度

任务B:

运行时间3秒

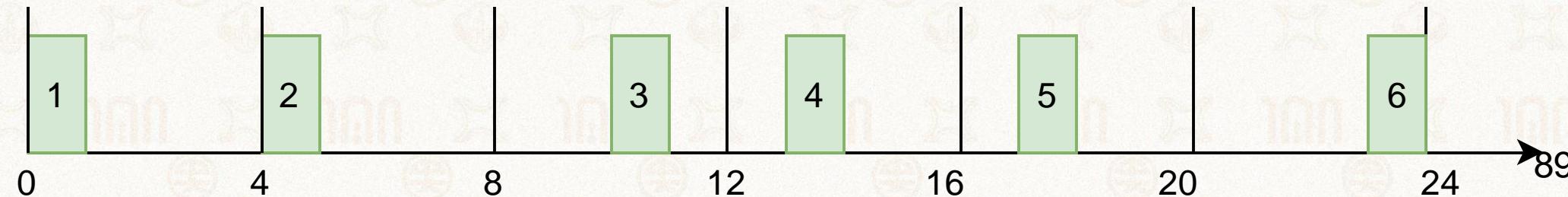
周期6秒



任务C:

运行时间1秒

周期4秒



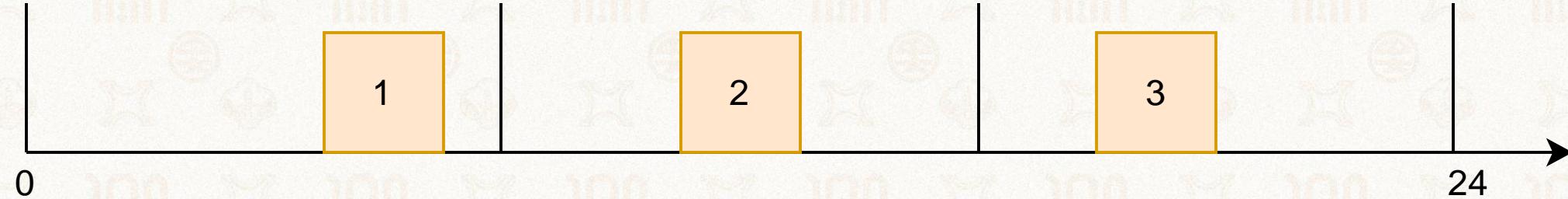


最早截止时间优先(Earliest Deadline First, EDF)

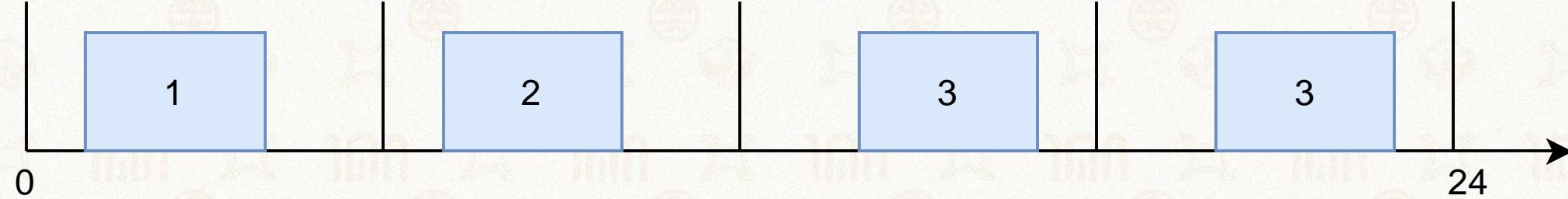
➤ EDF可调度的必要条件是U小于等于1

➤ EDF是可以被信赖的
➤ “Deadline是第一生产力”诚不我欺

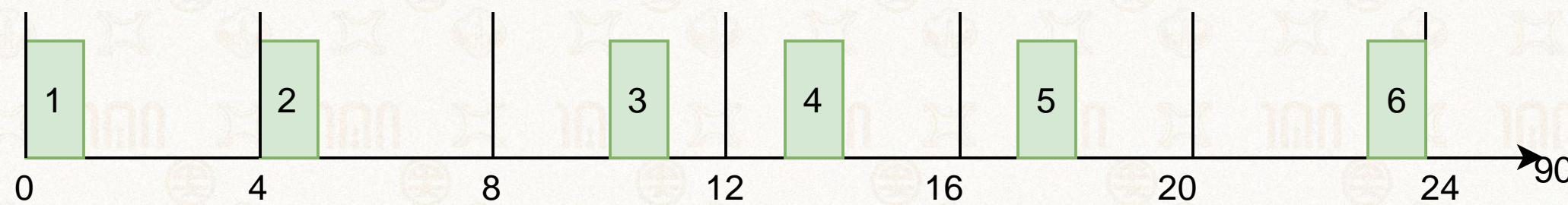
任务A:
运行时间2秒
周期8秒



任务B:
运行时间3秒
周期6秒



任务C:
运行时间1秒
周期4秒





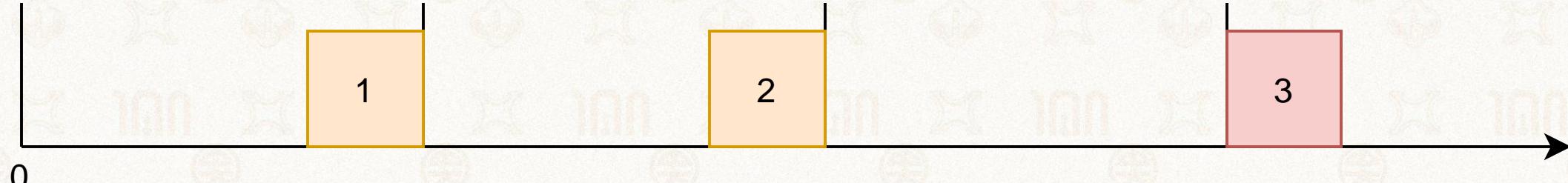
多米诺效应：需要进行可调度性分析

- 在任务不可调度时($U > 1$)，EDF会造成多数任务都错过截止时间

任务A:

运行时间2秒

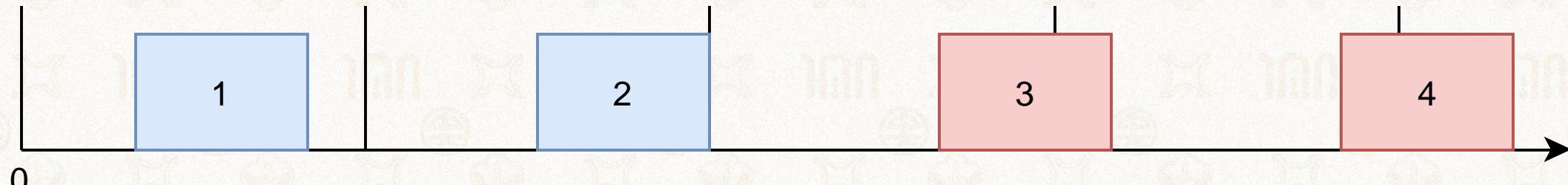
周期7秒



任务B:

运行时间3秒

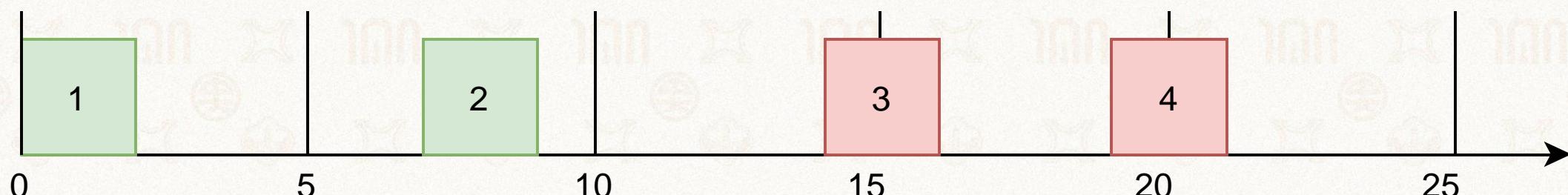
周期6秒



任务C:

运行时间2秒

周期5秒



错过截止时间的任务



大纲

- 调度的含义
- 调度的机制
- 单核调度策略
 - 经典调度
 - 优先级调度
 - 公平共享调度
 - 实时调度
- 多核调度策略
 - 调度进阶机制
 - 处理器亲和性
 - 现代Linux调度器



1924-2024
中山大學 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

1924-2024

谢谢

微信: suyuxin

钉钉: 苏玉鑫

B站: <https://space.bilibili.com/502854403>

软工集市课程专区: <https://ssemarket.cn/new/course>

匿名提问箱: <https://suask.me/ask-teacher/106/苏玉鑫>