

处理器调度

上海交通大学并行与分布式系统研究所

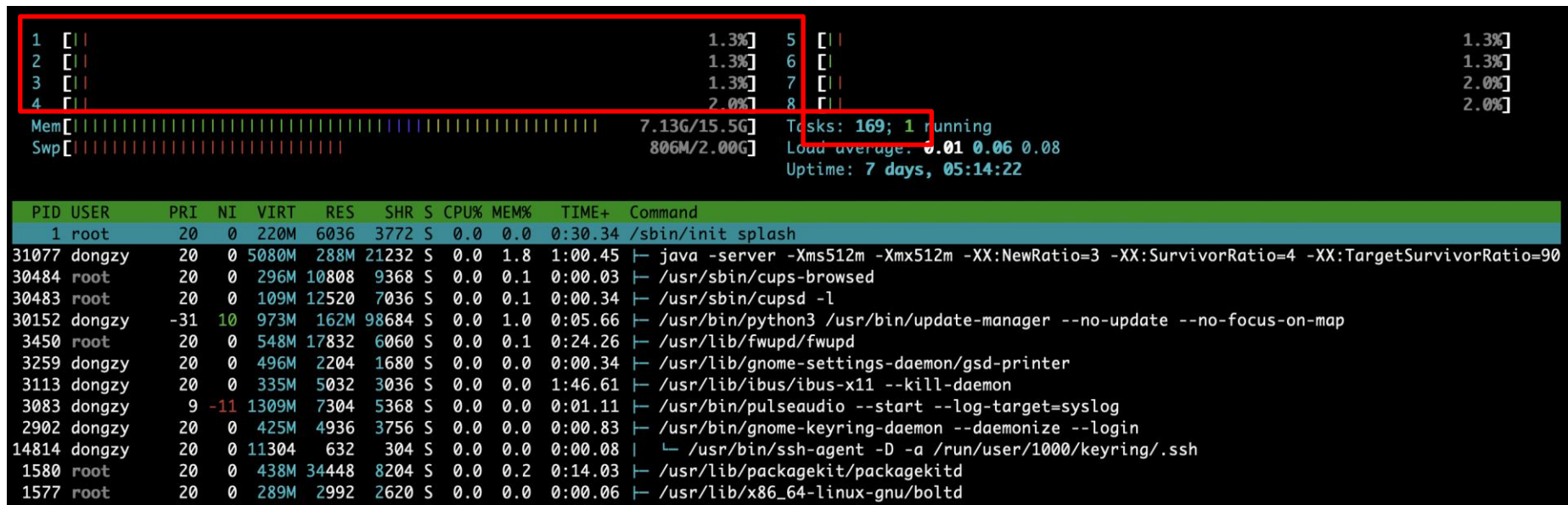
<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
 - 资料来自上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

处理器调度

系统中的任务数远多于处理器数

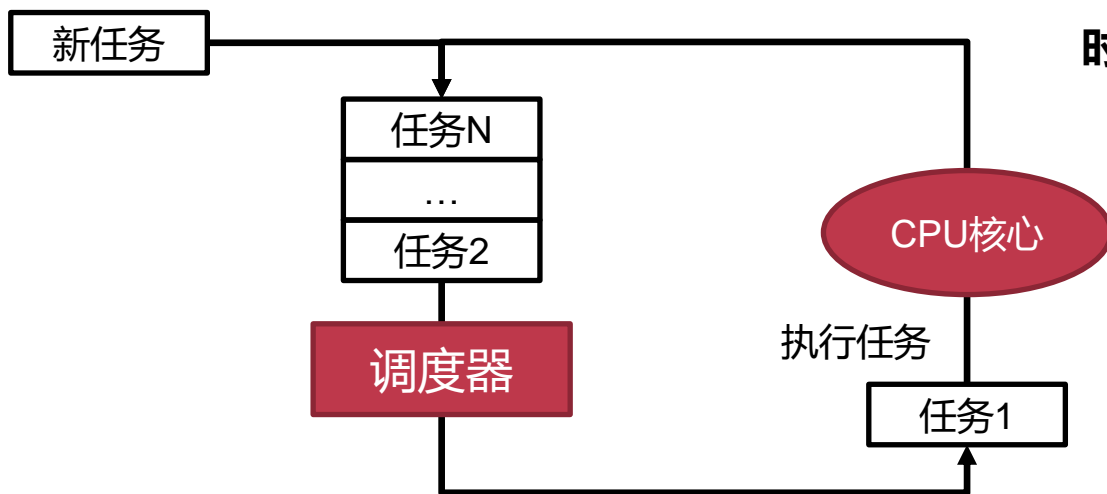


任务 (Task) : 线程、单线程进程

仅有8个处理器，如何运行169个任务？

处理器调度

对象： CPU核心上运行的最小单元，例如线程，统一用“**任务**”（也可表示协程/进程）描述。



时机： 1) 执行时间用尽
2) 等待I/O请求
3) 睡眠
4) 中断
5) ...

决策： 1) 下一个执行的任务
2) 执行该任务的CPU
3) 执行的时长

如果没有调度器



程序员需要等30分钟才能播放他爱听的音乐



调度器的优势

调度器

+

CPU

一个运行30分钟
的机器学习程序



播放音乐



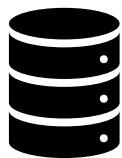
(程序执行片段)

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



不同场景下的调度指标

批处理系统



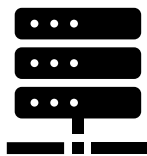
高吞吐量

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

调度指标

- **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
- ...

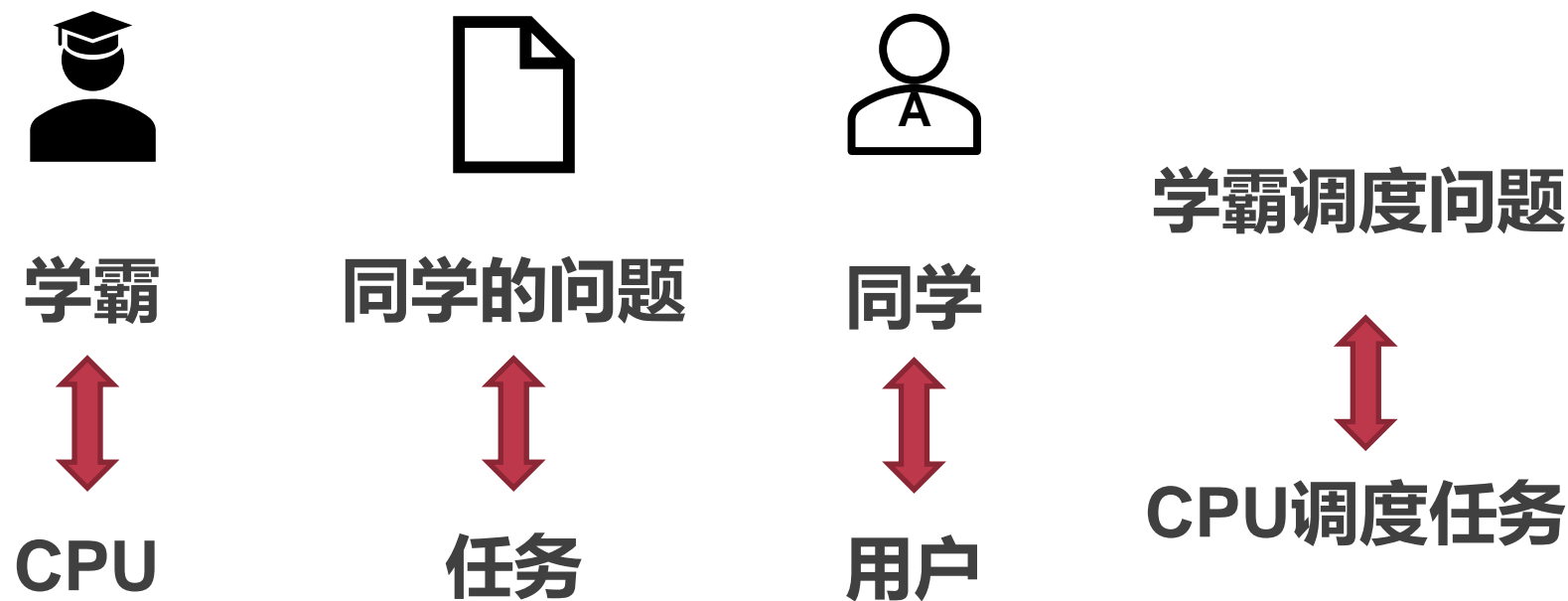
调度的挑战

- 缺少信息（没有**先知**）
 - 工作场景动态变化
- 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
 - ...

Classical Scheduling

经典调度

CPU调度与提问调度

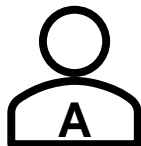


假设每个同学只提一个问题

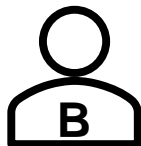
First Come First Served



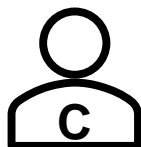
大家排队
先来后到!



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



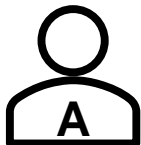
先到先得: 简单、直观

问题: 平均周转、响应时间过长

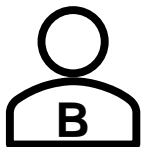
Shortest (Completion Time) Job First



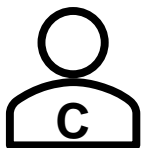
简单的问题先来



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




回顾

处理器上下文 vs. 线程上下文

- 有两个“上下文”概念，如何辨析？
 - 处理器上下文：用于保存切换时的寄存器状态（硬件）
 - 在每个PCB中均有保存
 - 线程上下文：表示目前操作系统正以哪线程的身份运行（软件）
 - 使用一个指向TCB的全局指针（代码中的curr_thread）

```
1 void process_exit_v2(int status)
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->ctx);
5     // 销毁内核栈
6     destroy_kern_stack(curr_proc->stack);
7     // 销毁虚拟地址空间
8     destroy_vmspace(curr_proc->vmspace);
9     // 保存退出状态
10    curr_proc->exit_status = status;
11    // 标记进程为退出状态
12    curr_proc->is_exit = TRUE;
13    // 告知内核选择下个需要执行的进程
14    schedule();
15 }
```

 curr_thread → pcb

小思考：操作系统如何实现curr_thread

- 内核数据

- 全局变量（单核CPU：固定位置）

- 多核CPU

- 需要维护多个curr_thread（每个核心对应一个）
- 如何实现呢？

练习题

Assume we have the following two jobs in the workload and **no I/O issues** are involved. Please fill in the following tables with the execution of CPU when we decide to use different schedule policies respectively. Suppose when a job arrives, it is added to the tail of a work queue. The RR policy selects the next job of the current job in the queue. The **RR** time-slice is 2ms. (**NOTE:** Time 0 means the task running during [0ms,1ms])

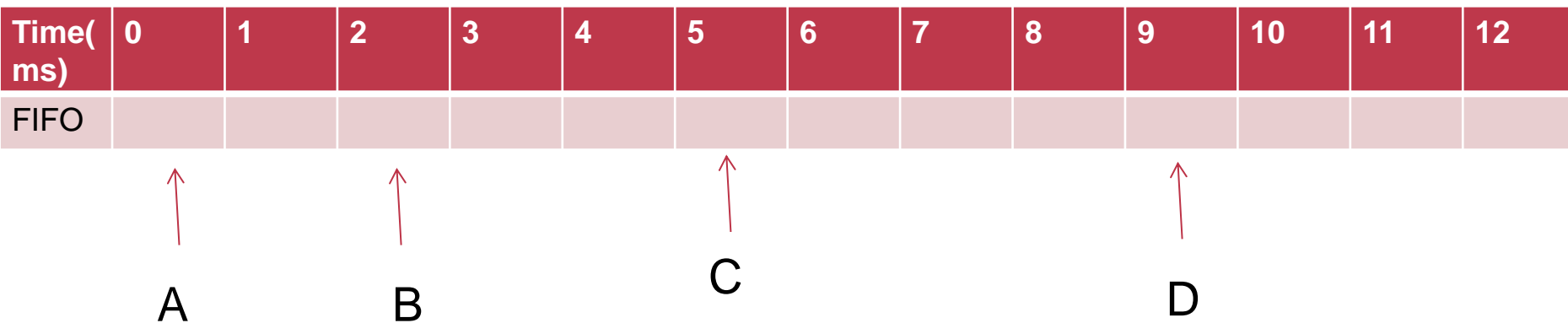
Job	Arrival Time	Length of Run-time
A	0ms	4ms
B	2ms	2ms
C	5ms	3ms
D	9ms	4ms

Schedule

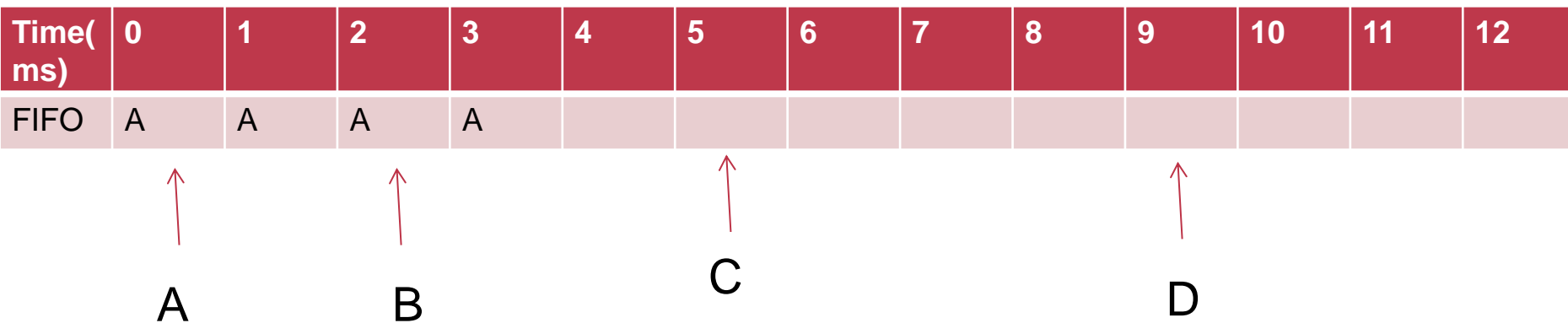
1. Assume we are using different scheduling policies, please calculate the **Average Turnaround Time**, **Average Response Time** and fill the **Job name** in the below table (when the CPU runs at Time 6): (12')

Policy	Average Turnaround Time	Average Response Time	CPU Time: 6ms
FIFO	[1]	[5]	[9]
SJF	[2]	[6]	[10]
STCF	[3]	[7]	[11]
RR	[4]	[8]	[12]

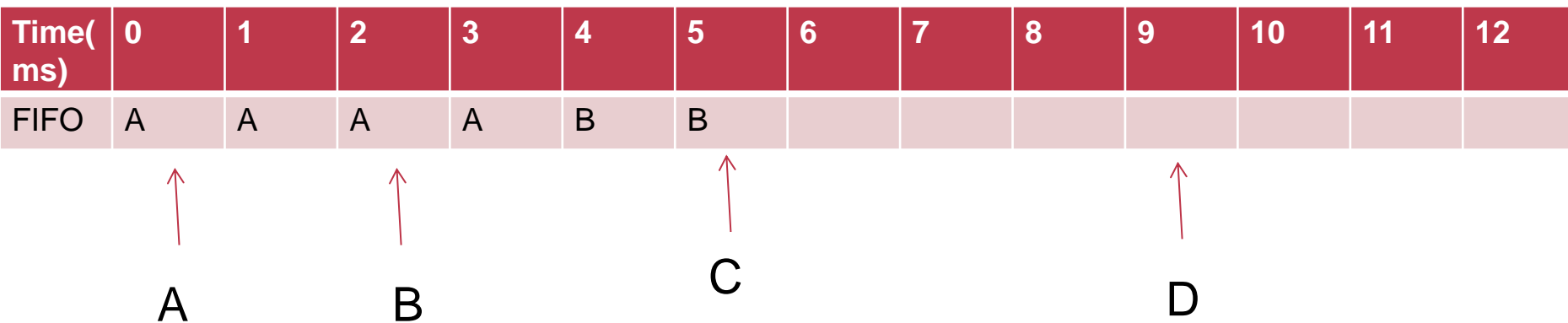
Schedule



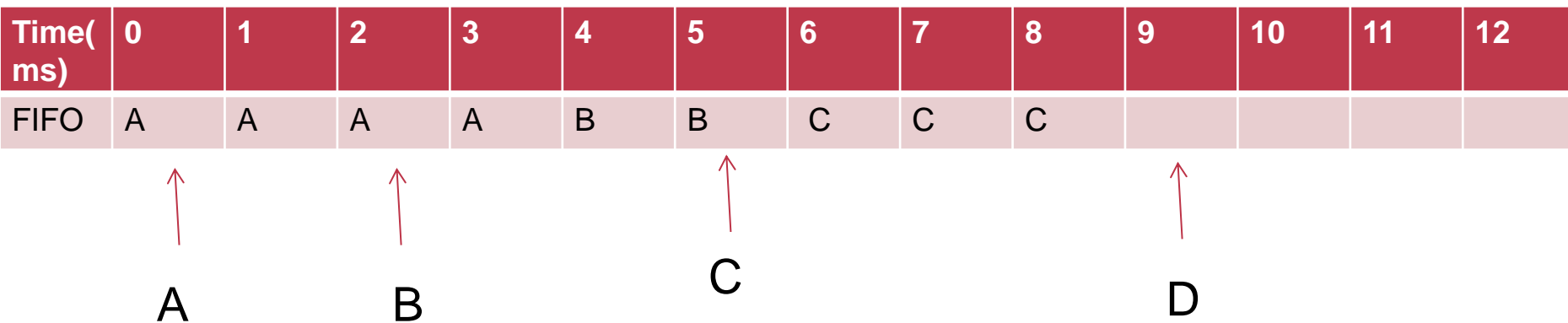
Schedule



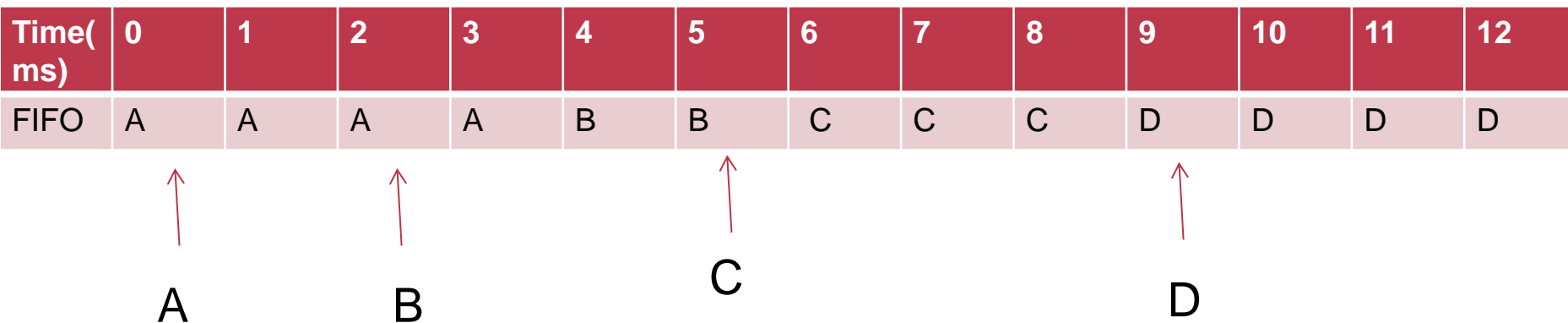
Schedule



Schedule



Schedule



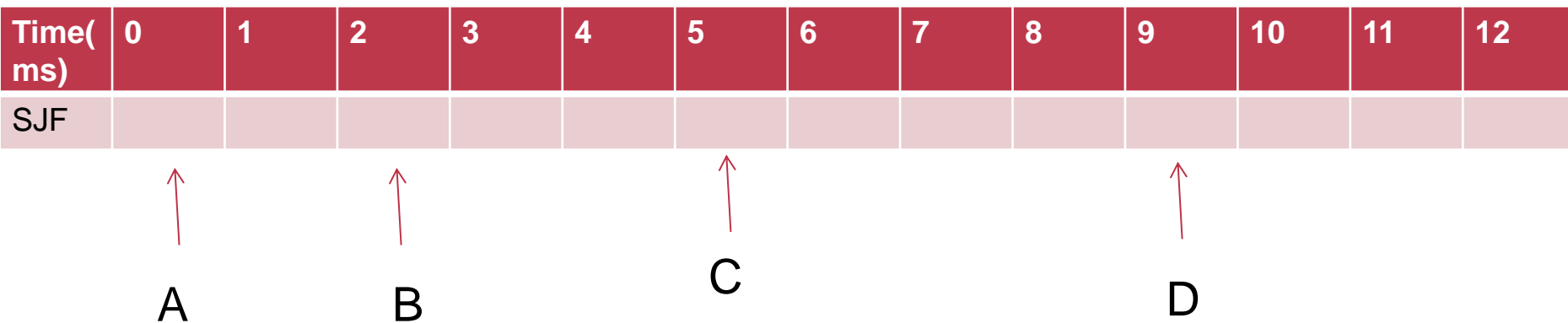
$$\text{Turnaround Time} = ((4-0)+(6-2)+(9-5)+(13-9)) / 4 = 4\text{ms}$$

$$\text{Response Time} = ((0-0)+(4-2)+(6-5)+(9-9)) / 4 = 0.75\text{ms}$$

Schedule

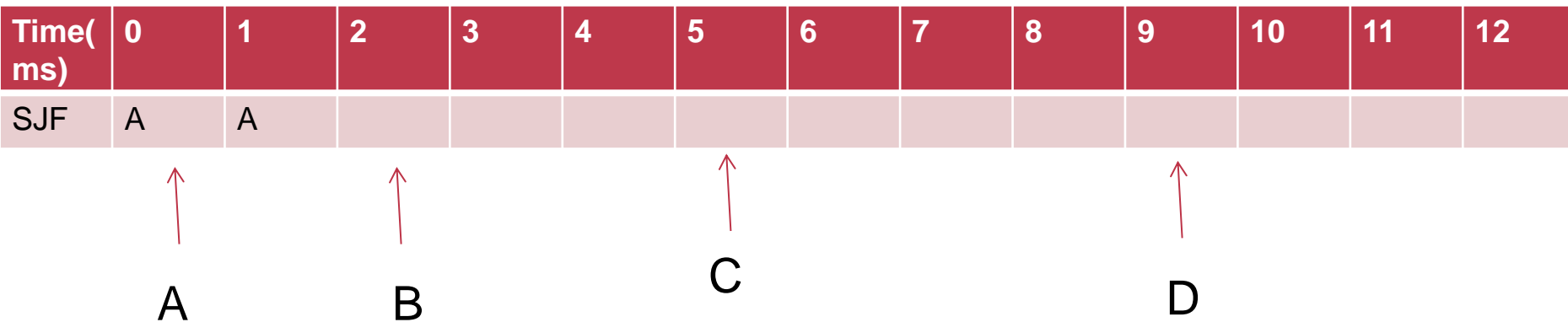
SJF v.s. SCTJF

preemptive!



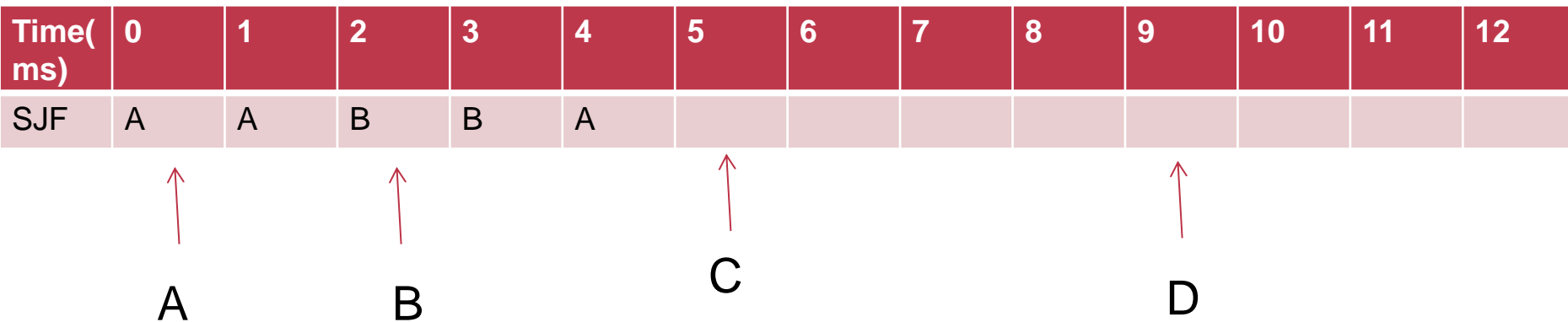
Schedule

preemptive!



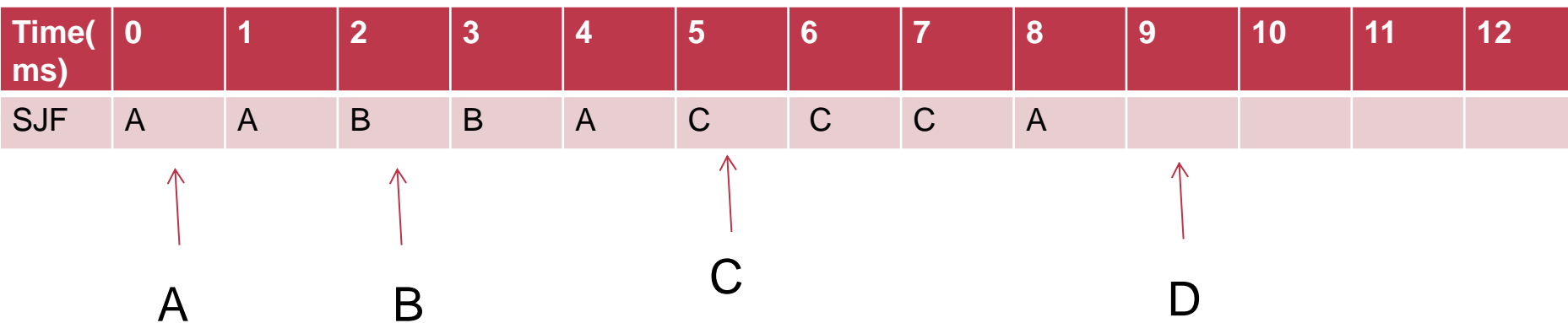
Schedule

preemptive!



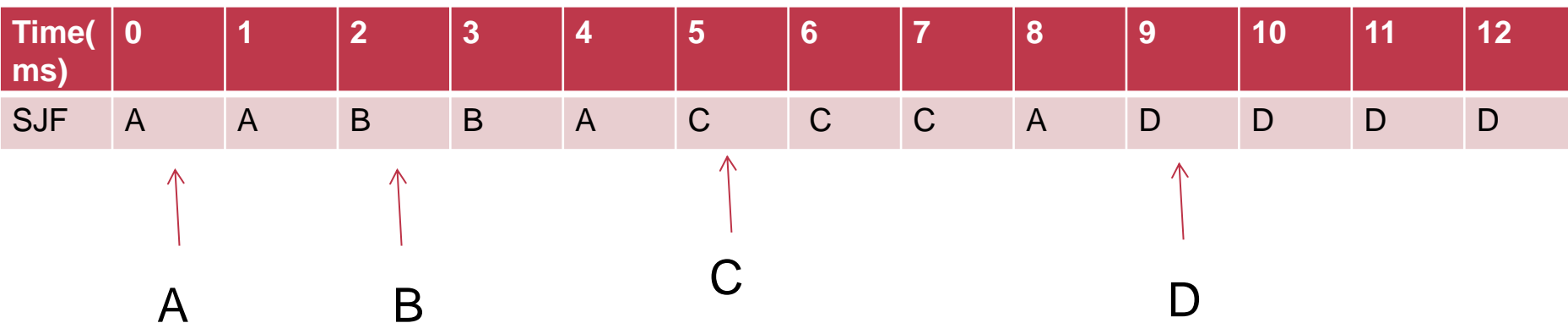
Schedule

preemptive!



Schedule

preemptive!



Turnaround Time = $((9-0)+(4-2)+(8-5)+(13-9)) / 4 = 4.5\text{ms}$ CPU 6ms : C

Response Time = $((0-0)+(2-2)+(5-5)+(9-9)) / 4 = 0\text{ms}$



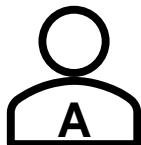
抢占式调度 (Preemptive Scheduling)

- **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**

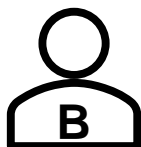
Round Robin (时间片轮转)



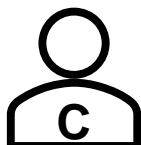
公平起见
每人轮流一分钟!



感觉多等了好久...

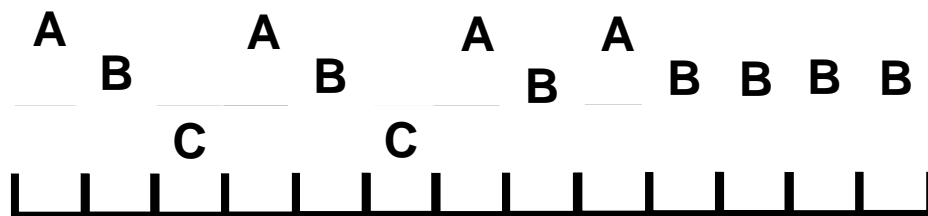


学霸的响应时间短
了好多



学霸的响应得更快了

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

练习题

- 在任务具有什么特征下，RR的**周转时间**问题最为明显？

每个任务的执行时间差不多相同

(举例：10个任务，每个任务需运行1S，RR下每个任务周转时间都接近10S)

- 时间片长短应该如何确定？

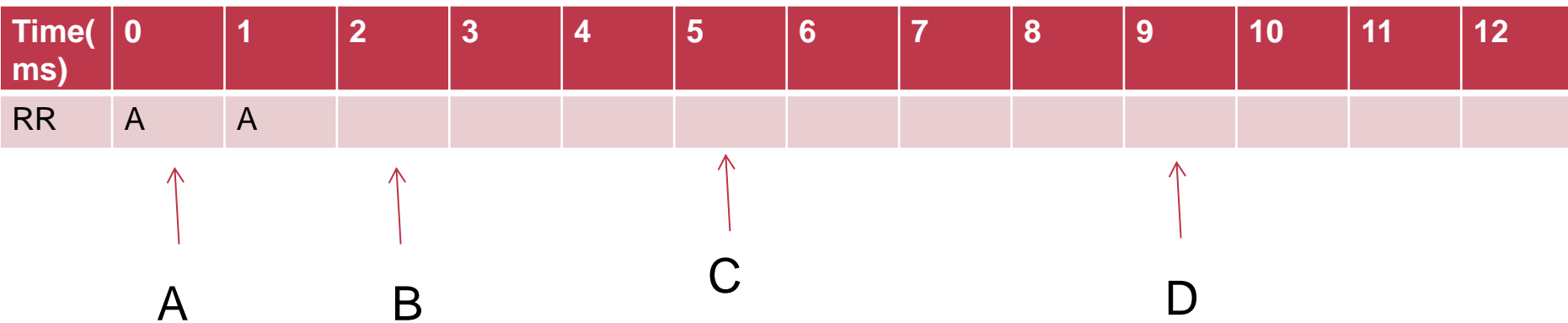
- 过长的时间片会导致什么问题？ FCFS
- 过短的时间片会导致什么问题？ 调度开销

练习题

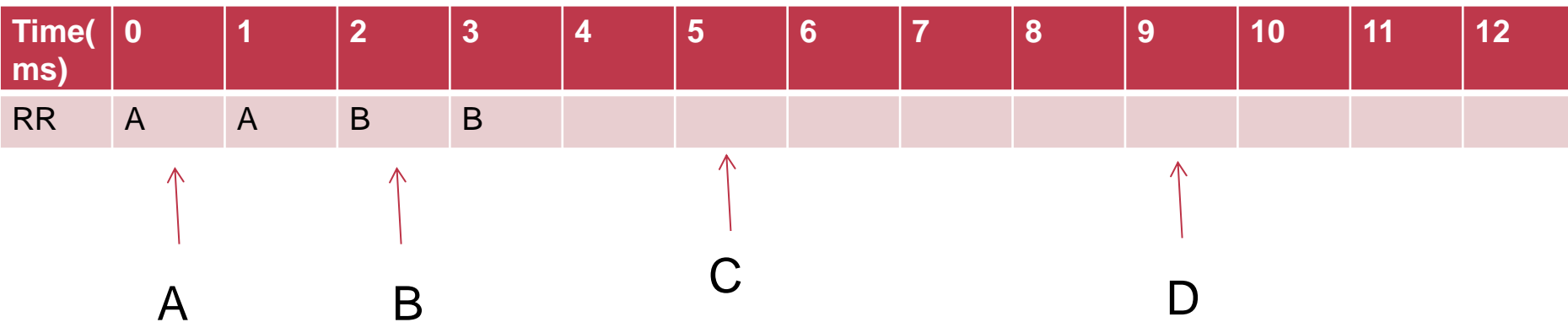
Assume we have the following two jobs in the workload and **no I/O issues** are involved. Please fill in the following tables with the execution of CPU when we decide to use different schedule policies respectively. Suppose when a job arrives, it is added to the tail of a work queue. The RR policy selects the next job of the current job in the queue. The **RR** time-slice is 2ms. (**NOTE:** Time 0 means the task running during [0ms,1ms])

Job	Arrival Time	Length of Run-time
A	0ms	4ms
B	2ms	2ms
C	5ms	3ms
D	9ms	4ms

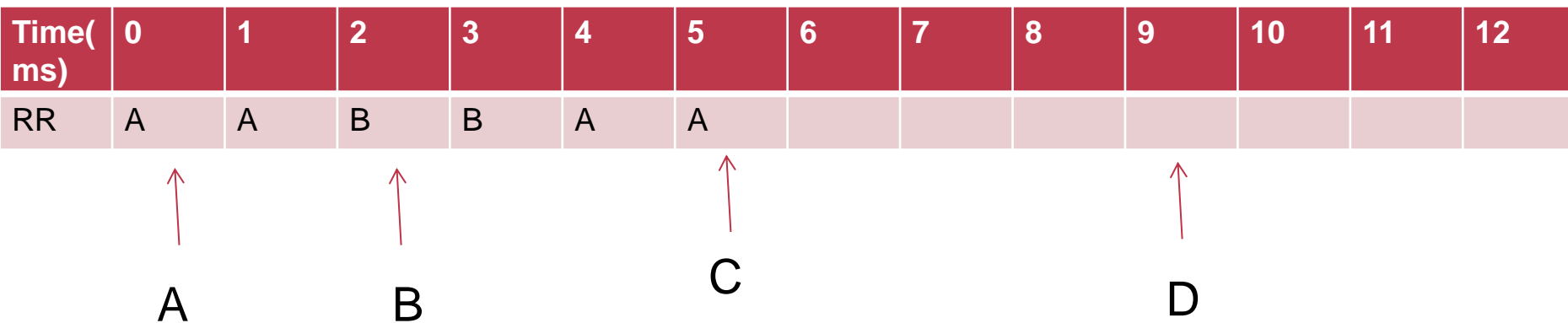
Schedule



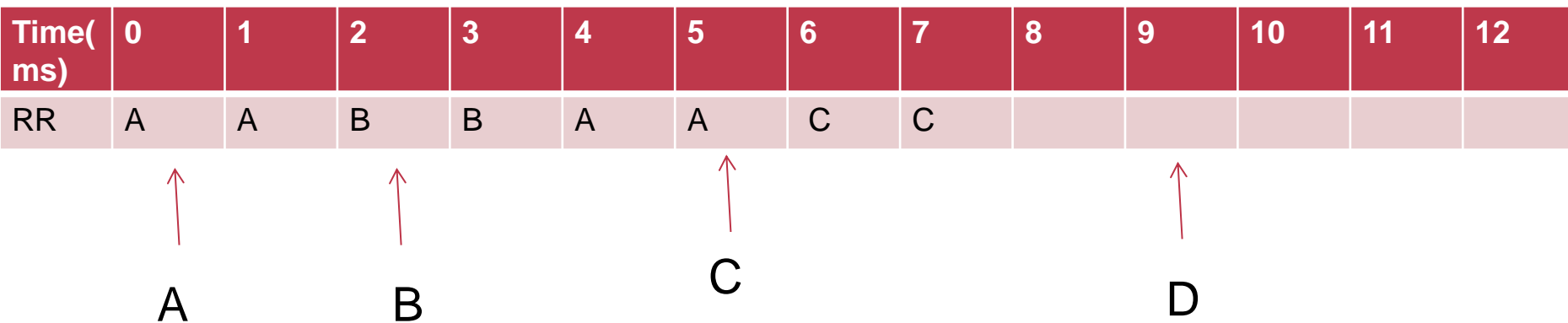
Schedule



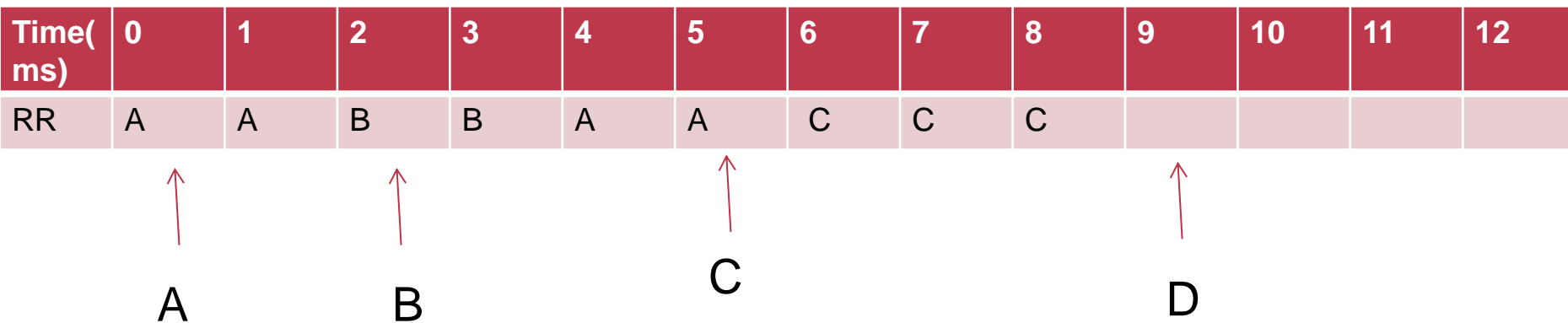
Schedule



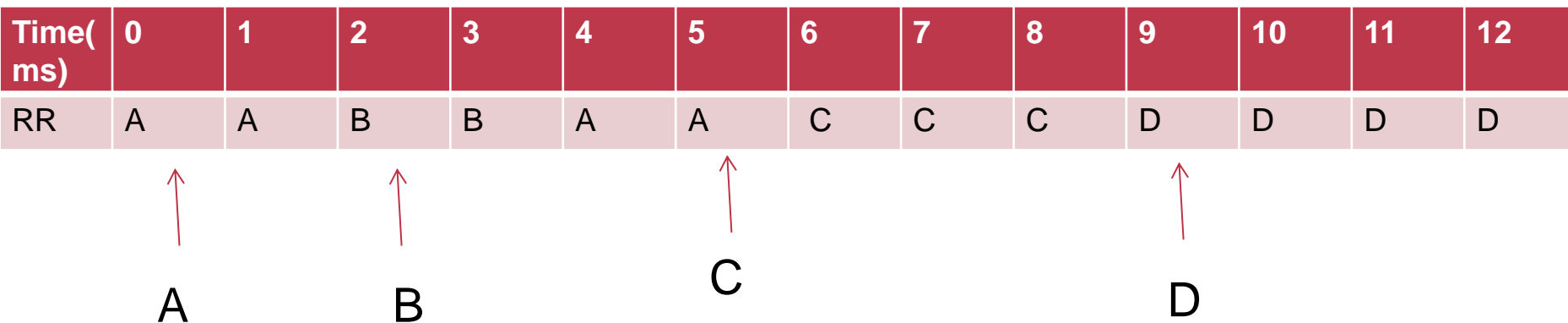
Schedule



Schedule



Schedule

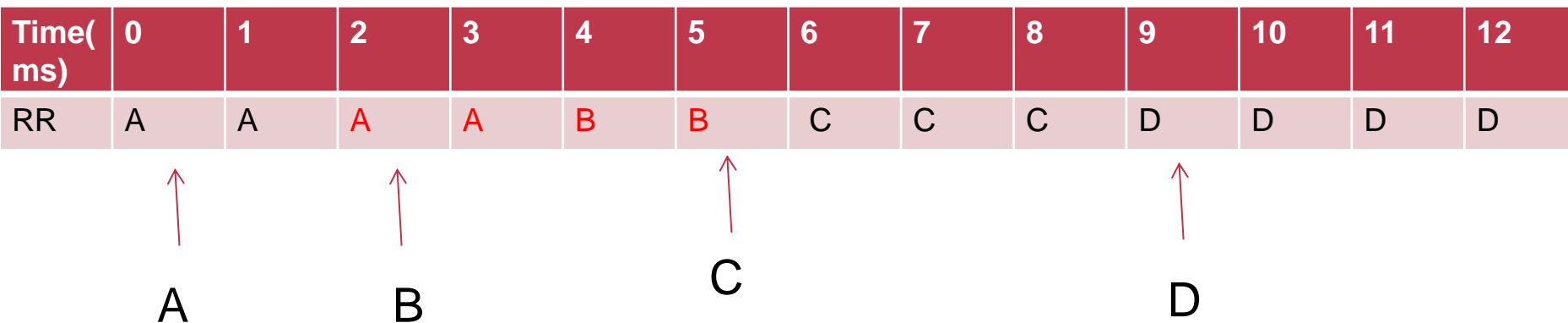


Turnaround Time = $((6-0)+(4-2)+(9-5)+(13-9)) / 4 = 4\text{ms}$ CPU 6ms : C

Response Time = $((0-0)+(2-2)+(6-5)+(9-9)) / 4 = 0.25\text{ms}$

Schedule

if RR movement > accept new job



Turnaround Time = $((4-0)+(6-2)+(9-5)+(13-9)) / 4 = 4\text{ms}$ CPU 6ms : C

Response Time = $((0-0)+(4-2)+(6-5)+(9-9)) / 4 = 0.75\text{ms}$

Priority Scheduling

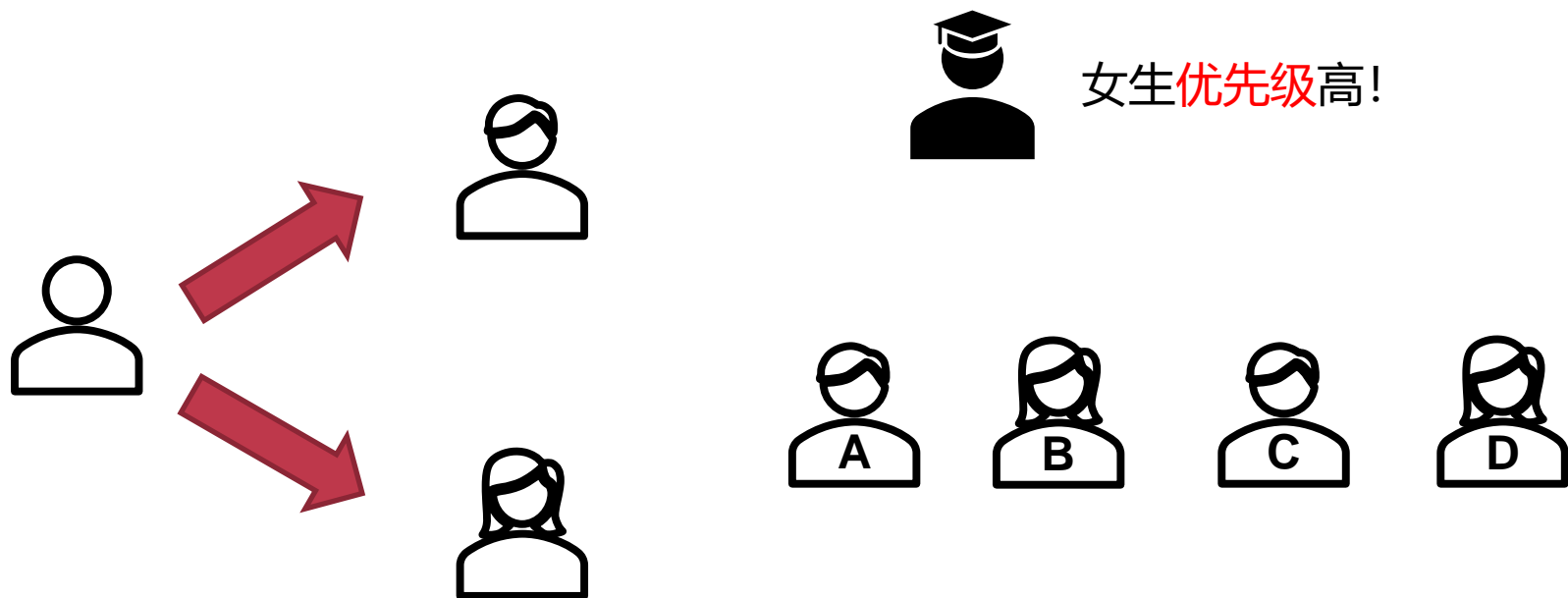


优先级调度

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**

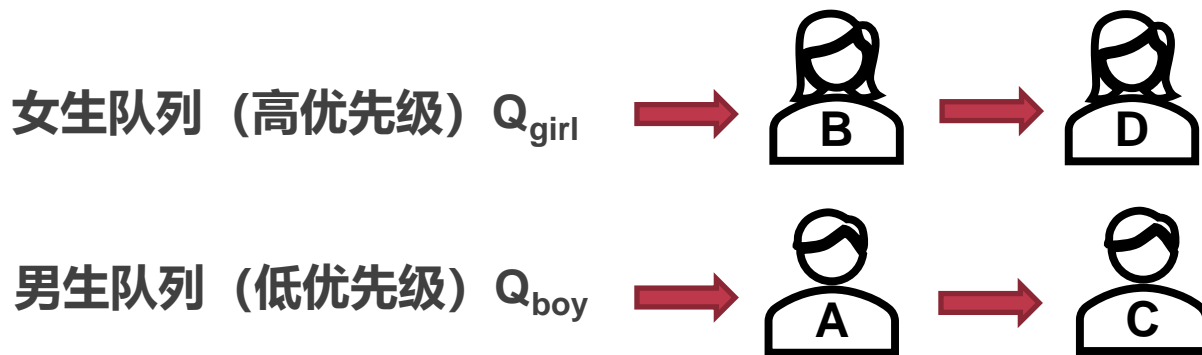
添加条件：优先级



多级队列

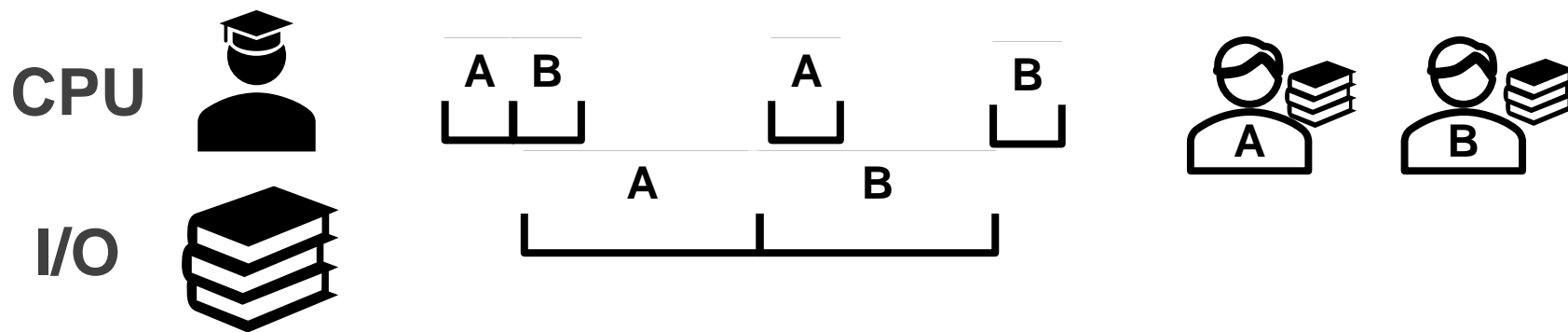
Multi-Level Queue (MLQ)

- 1) 维护多个队列，每个队列设置不同优先级
- 2) 高优先级队列中的任务优先执行
- 3) 同优先级内使用Round Robin调度（也可使用其他调度策略）



添加条件：阅读OS书（类比I/O操作）

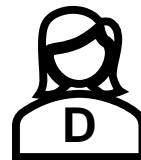
- 学霸告诉同学需要看OS书
 - （学霸只有一本OS书，同一时间只有一个同学能够阅读）
- 阅读完OS书后，同学再和学霸确认知识点



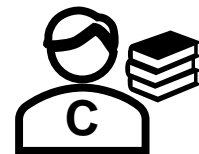
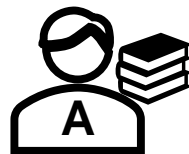
问题：低资源利用率

问题：
多种资源（学霸和OS书）
没有同时利用起来

优先级0（高）



优先级1（低）



资源空闲



资源空闲



资源空闲



思考：优先级的选取

- 什么样的任务应该有高优先级？
 - I/O密集型任务
 - 为了更高的资源利用率
 - 用户主动设置的重要任务
 - 接近DDL的任务（必须在短时间内完成）

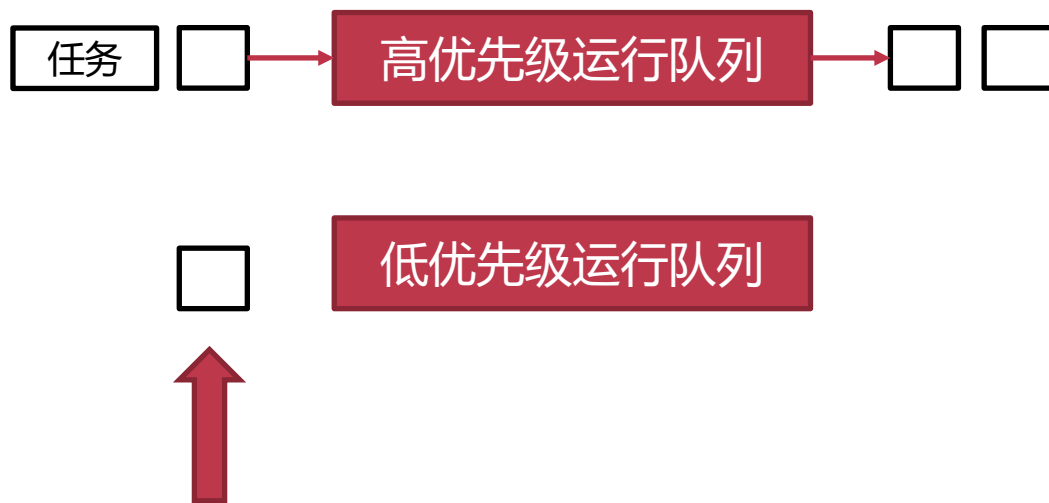
练习题：以下经典调度策略是否属于优先级调度？

- **First Come First Served**
- **Shortest Job First**
- **Round Robin**

优先级的动态调整

- 操作系统中的工作场景是**动态变化**的
- 静态设置的优先级可能导致
 - 资源利用率低
 - 一个CPU密集型动态转变为I/O密集型任务
 - 优先级反转
 - ...
- 某些场景下，任务的优先级需要动态调整

静态优先级的问题：低优先级任务饥饿



被高优先级任务阻塞，长时间无法执行

思考：设计满足以下要求的优先级调度策略

- 是否可以设计一种调度策略，既可以让短任务优先执行，又不会让长任务产生饥饿？
 - 优先级 = $\frac{T_{Waiting}}{T_{Run}}$
 - 如果两个任务等待时间（ $T_{Waiting}$ ）相同，则运行时间越短越优先
 - 如果两个任务运行时间相同，则等待时间越长，越优先
 - 结合FCFS策略和SJF策略，避免了SJF策略在公平性方面的问题

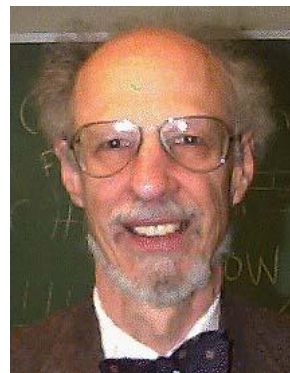
MLFQ: 多级反馈队列

目前介绍的调度策略的限制

- **周转时间、响应时间过长**
 - FCFS
- **依赖对于任务的先验知识**
 - 预知任务执行时间
 - SJF
 - 预知任务是否为I/O密集型任务
 - MLQ（用于设置任务优先级）
 - 适用于静态调度场景（预先知道任务情况）

多级反馈队列

- **Multi-Level Feedback Queue (MLFQ)**
- **Corbato**
 - 于1962年，发表了Compatible Time-Sharing System (CTSS)的相关论文
 - 在该论文中提出了MLFQ等方法
 - 于1990年，获得图灵奖
 - 因CTSS和Multics方面的贡献



MLFQ的主要目标与思路

- 一个**无需先验知识**的通用调度策略
 - 周转时间低、响应时间低
 - 调度开销低
- 通过动态分析任务运行历史，总结任务特征
 - 类似思想的体现：页替换策略、预取
 - 需要注意：如果任务特征变化频繁，效果也不一定理想

基本算法（基于Multi-Level Queue）

- **规则 1:**

- 优先级高的任务会抢占优先级低的任务

- **规则 2:**

- 每个任务会被分配时间片，优先级相同的两个任务使用时间片轮转

如何设置任务优先级？

- **针对混合工作场景**
 - 执行时间短的任务
 - 交互式任务
 - I/O密集型任务
 - 执行时间长的任务
 - CPU密集型计算任务
- **规则 3:**
 - 任务被创建时，假设该任务是短任务，为它分配最高优先级
- **规则 4a:**
 - 一个任务时间片耗尽后，它的优先级会被降低一级
- **规则 4b:**
 - 如果一个任务在时间片耗尽让出CPU，那么它的优先级不变
 - （通常，交互式任务/I/O密集型任务符合该特点）
 - 任务重新执行时，会被分配新的时间片

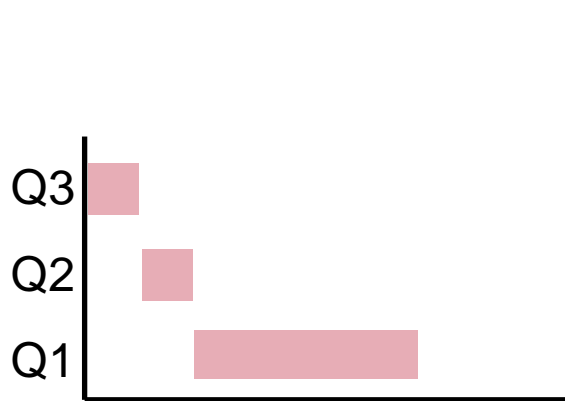
样例执行1、2

对于长任务（红色任务）：

- MLFQ会逐渐降低它的优先级
- 并将它视为长任务

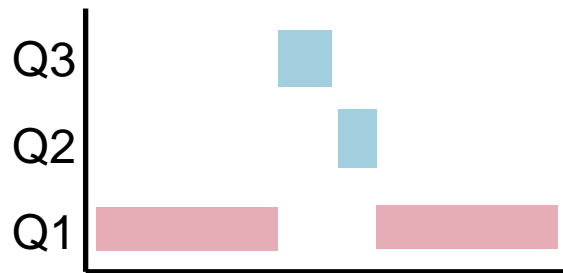
对于短任务（蓝色任务）：

- 它会很快执行完



*Q3优先级最高

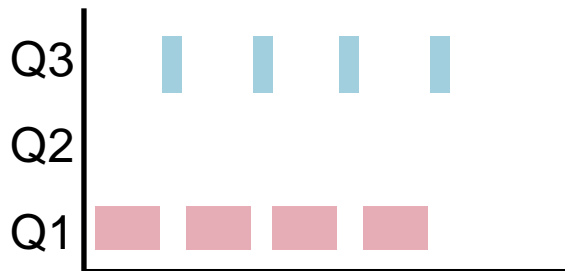
1. 一个长任务的执行



2. 长任务执行时，一个短任务被创建

样例执行3

蓝色任务
红色任务



对于I/O密集型任务：

- 它会在时间片执行完以前放弃CPU
- MLFQ保持它的优先级不变即可

3. 混合CPU密集型（红色任务）与I/O密集型任务（蓝色任务）的执行

基本算法的问题（一）

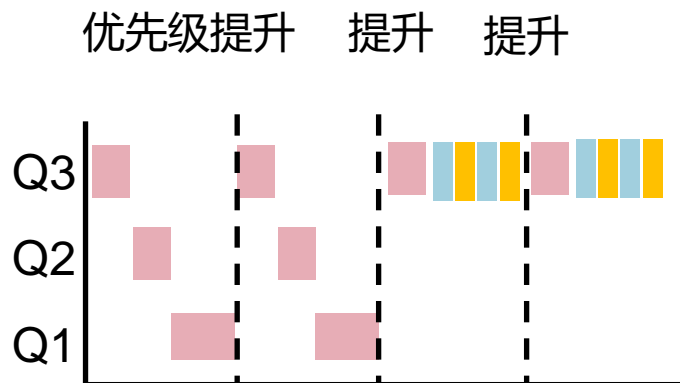
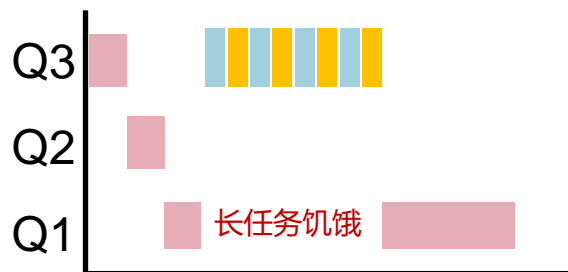
- **长任务饥饿**
 - 过多的短任务、I/O密集型任务可能占用所有CPU时间
- **任务特征可能动态变化**
 - CPU密集型任务→交互式任务， ...

定时优先级提升

思考：为什么要提升全部的优先级？

- **规则 5:**
 - 在某个时间段 S 后，将系统中所有任务优先级升为最高
- **效果1：避免长任务饿死**
 - 所有任务的优先级会定时地提升最高
 - 最高级队列采用RR，长任务一定会被调度到
- **效果2：针对任务特征动态变化的场景**
 - MLFQ会定时地重新审视每个任务

样例执行4

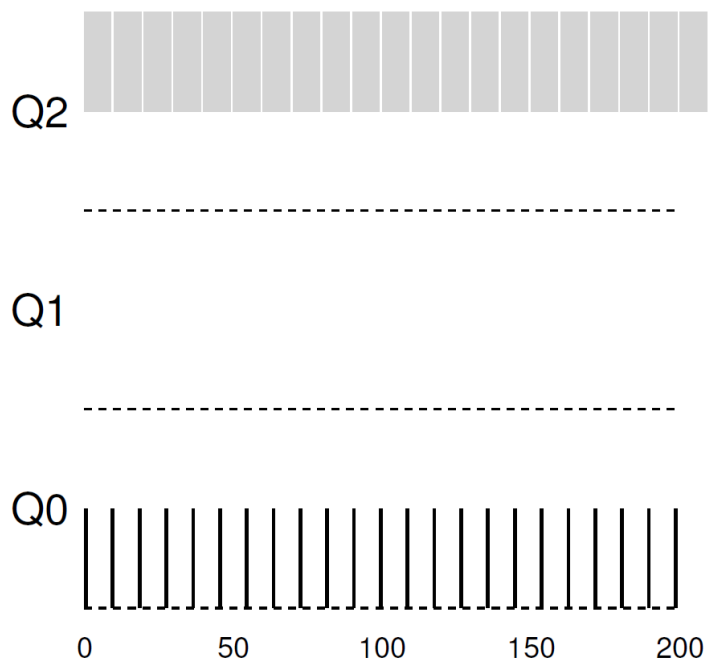


4. 采用定时优先级提升的前后对比（左为采用前，右为采用后）

基本算法的问题（二）

- 无法应对抢占CPU时间的攻击
 - 恶意任务在时间片用完前发起I/O请求
 - 避免MLFQ将该任务的优先级降低
 - 并且每次重新执行时间片会被重置
 - 几乎独占CPU!

攻击示例



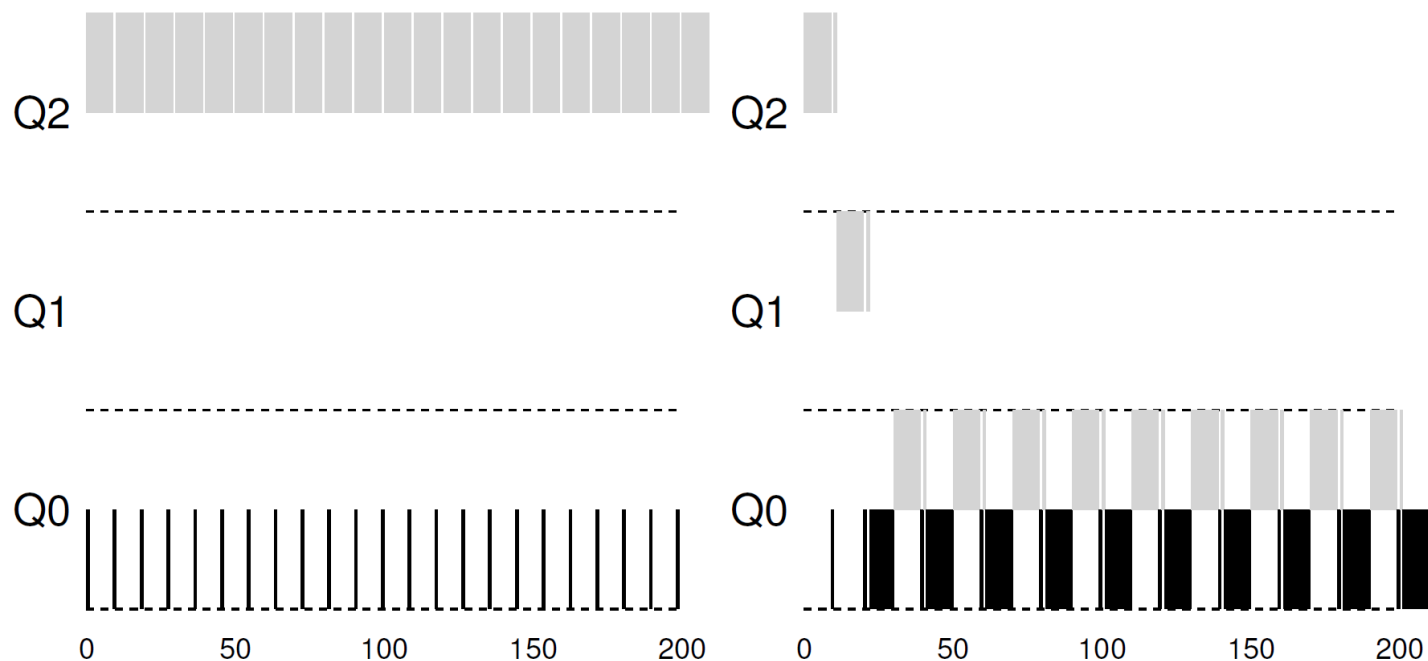
攻击者任务始终享有高优先级

低优先级任务仅拥有很少的CPU时间片

更准确地记录执行时间

- **规则 4:**
 - 一个任务时间片耗尽后（无论它期间放弃了多次CPU，它的时间片不会被重置），它的优先级会被降低一级
- **更新策略**
 - 记录每个任务在当前优先级使用的时间片
 - 当累计一个完整时间片被用完后，降低其优先级

样例执行5



5.使用准确记录执行时间的前后对比（左为采用前，右为采用后）

MLFQ的参数调试

- **如何确定MLFQ的各种参数?**
 - 优先级队列的数量
 - 不同队列的时间片长短
 - 定时优先级提升的时间间隔
- **每个参数都体现了MLFQ的权衡**
 - 对于不同的工作场景，不同的参数会导致不一样的表现

MLFQ各个队列时间片长短的选择

- 为不同队列选择不同的时间片
 - 高优先级队列时间片较短，针对短任务
 - 降低响应时间
 - 低优先级队列时间片较长，针对长任务
 - 降低调度开销



多级反馈队列总结

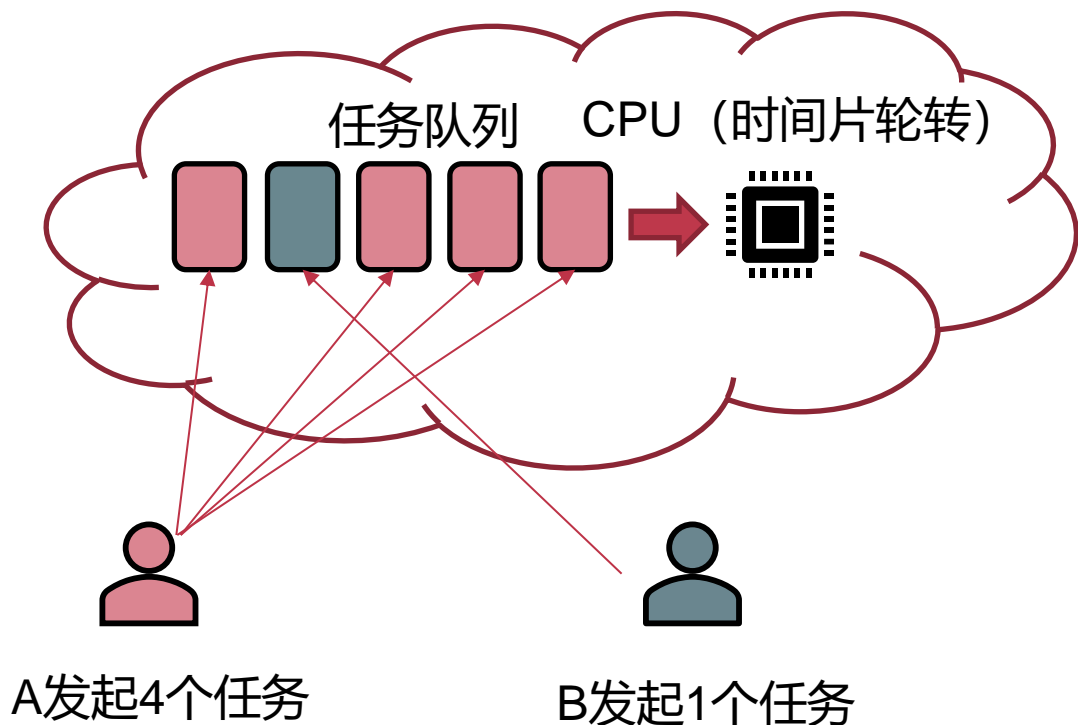
- **Multi-Level Feedback Queue**
 - 通过观察任务的历史执行，动态确定任务优先级
 - 无需任务的先验知识
 - 同时达到了周转时间和响应时间两方面的要求
 - 对于短任务，周转时间指标近似于SJF
 - 对于交互式任务，响应时间指标近似于RR
 - 可以避免长任务的饿死
- **许多著名系统的调度器是基于MLFQ实现的**
 - BSD, Solaris, Windows NT 和后续Windows操作系统

Fair-Share Scheduling

公平共享调度

场景：共享服务器

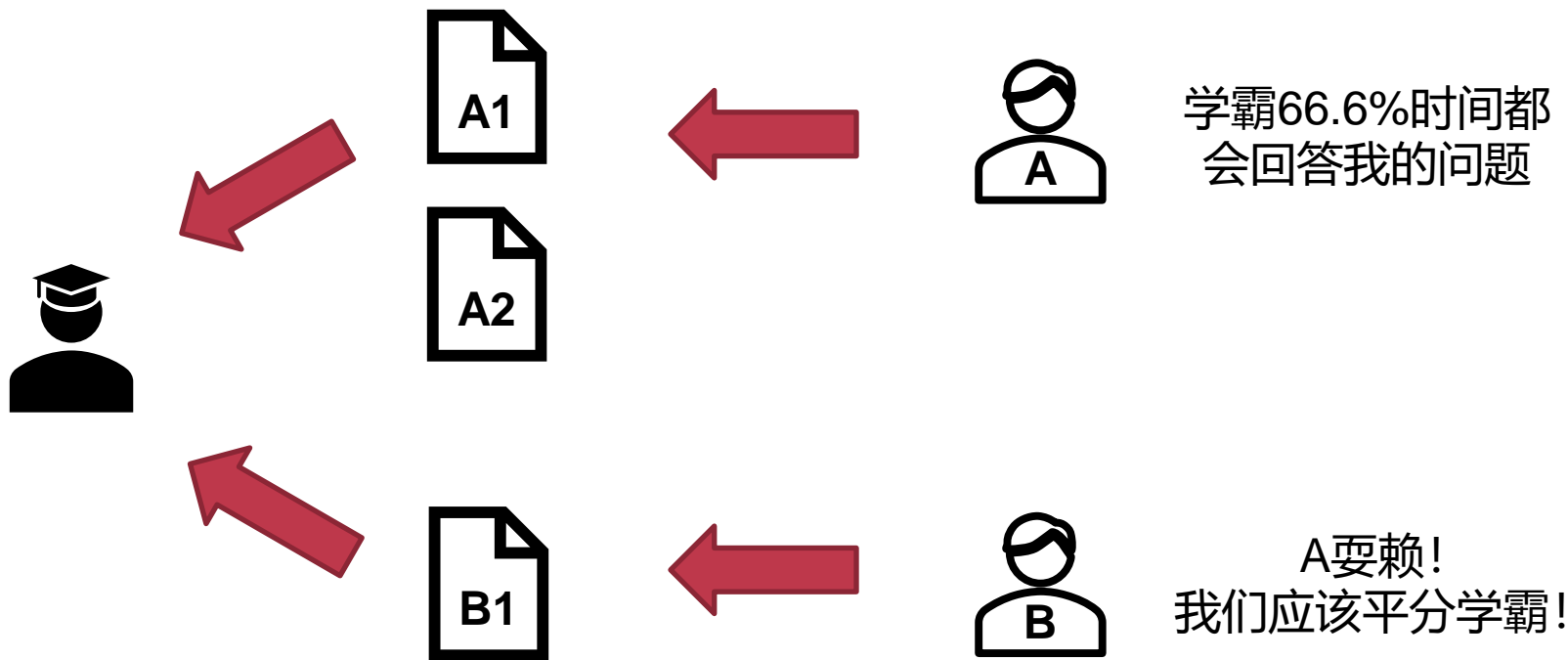
- A和B两位同学合资买了一台服务器，他们每人负担了一半的费用
- 两人应**均分**CPU时间
 - 而非被发起的任务数量决定
- 如果CPU使用时间片轮转调度
 - A占用80%CPU时间
 - B占用20%CPU时间
- 实例：容器CPU cgroup



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“份额”以确定相对比例
 - 例：份额为4的用户使用资源，是份额为2的用户的2倍

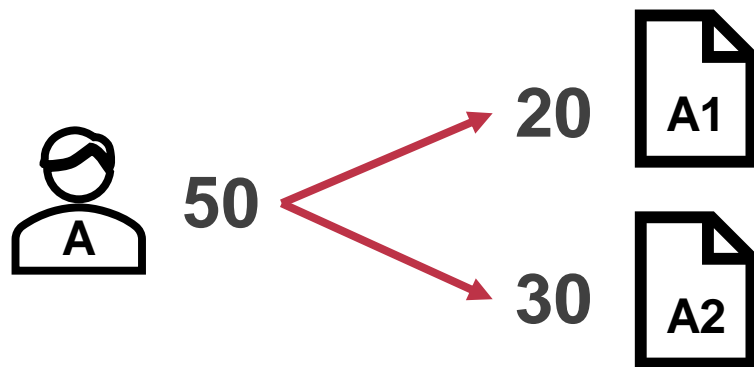
添加条件：一个同学会问多个问题



方法：使用“ticket”表示任务的份额

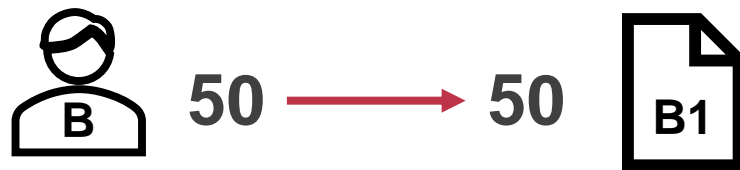
- *ticket*: 每个问题对应的份额
- T : ticket的总量
- 问题A1可占用学霸时间的比例

$$- \frac{ticket_{A1}}{T} = \frac{20}{100} = \frac{1}{5}$$



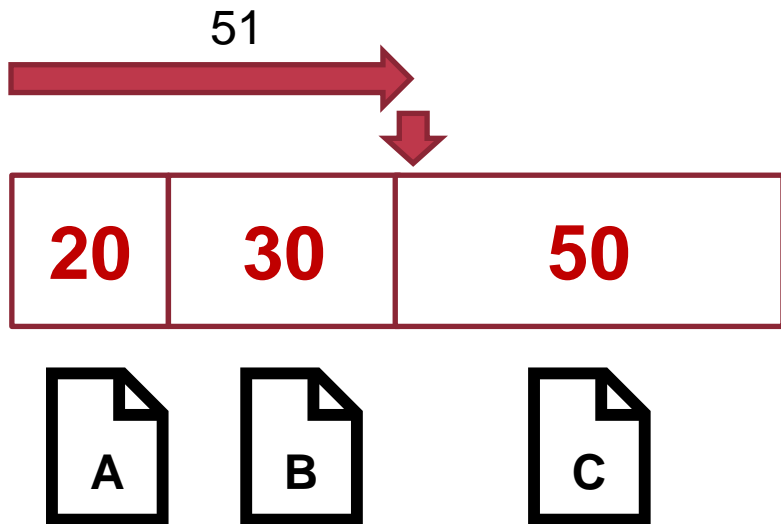
- 同学A可占用学霸时间的比例

$$- \frac{ticket_A}{T} = \frac{ticket_{A1} + ticket_{A2}}{T} = \frac{50}{100} = \frac{1}{2}$$



一种公平共享的实现：彩票调度 (Lottery Scheduling)

- 每次调度时，生成随机数 $R \in [0, T)$
- 根据 R ，找到对应的任务
 - $R=51 \rightarrow$ 调度C



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

思考：份额与优先级的异同？

- **份额影响任务对CPU的占用比例**
 - 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

思考：随机的利弊

- 随机的的好处是？
 - 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

步幅调度 (Stride Scheduling)

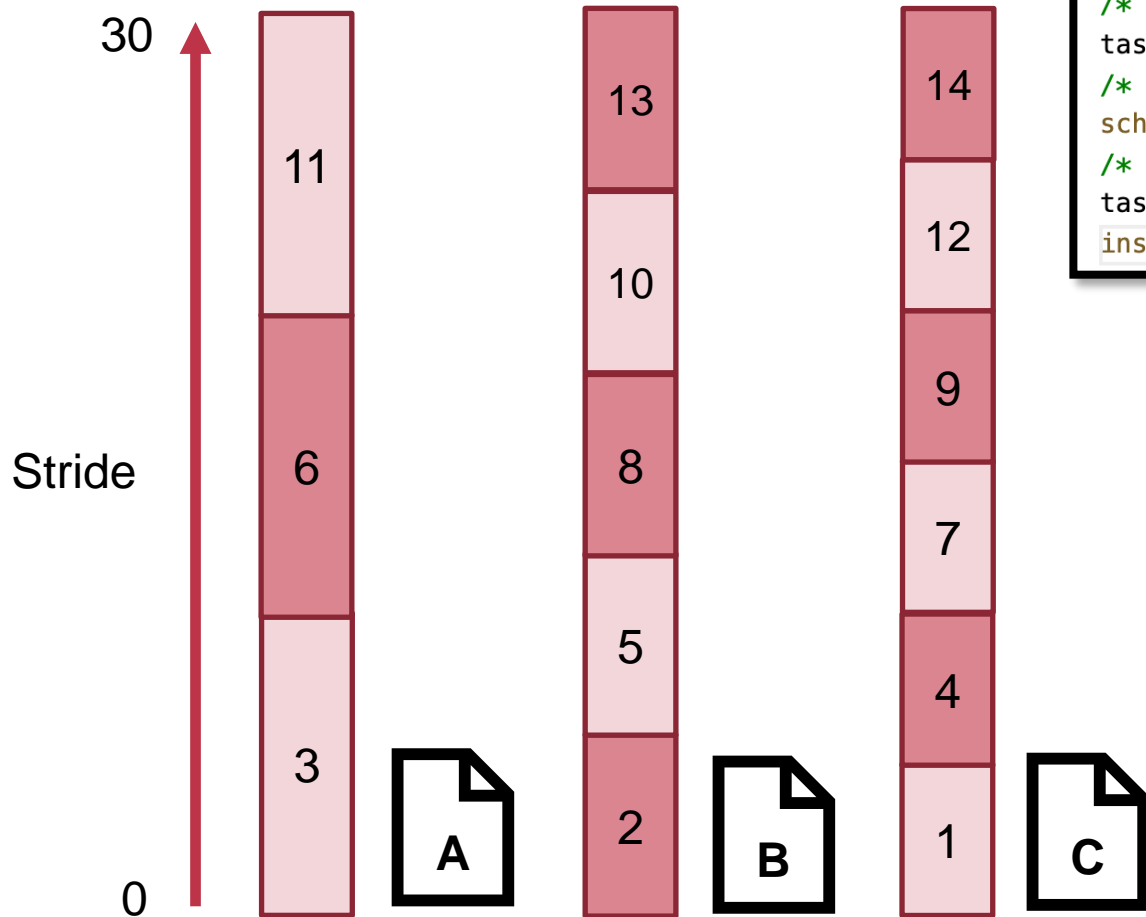
- **确定性版本**的Lottery Scheduling
 - 可以沿用tickets的概念
- **Stride——步幅，任务一次执行增加的虚拟时间**
 - $stride = \frac{MaxStride}{ticket}$
 - MaxStride是一个足够大的整数
 - 本例中设为所有tickets的最小公倍数
- **Pass——累计执行的虚拟时间**

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5

MaxStride = 300

步幅调度 (Stride Scheduling)

调度次序: 1-14



```
/* select client with minimum pass value */  
task = remove_queue_min(q);  
/* use resource for quantum */  
schedule(task);  
/* compute next pass using stride */  
task->pass += task->stride;  
insert_queue(q, current);
```

挑选pass最小的任务优先执行

	Ticket	Stride
A	30	10
B	50	6
C	60	5

公平共享调度

	Lottery Scheduling	Stride Scheduling
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

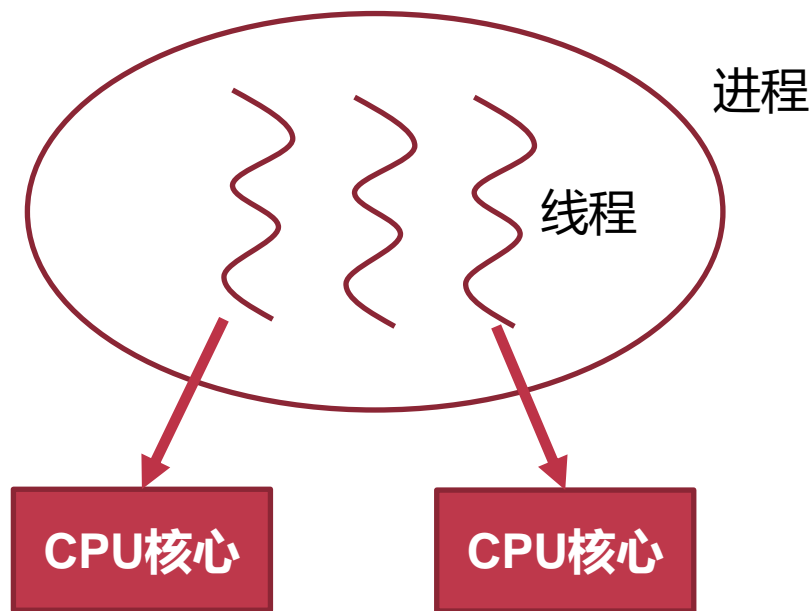
预期——根据任务份额ticket计算的执行时间期望

Multicore Scheduling Policy

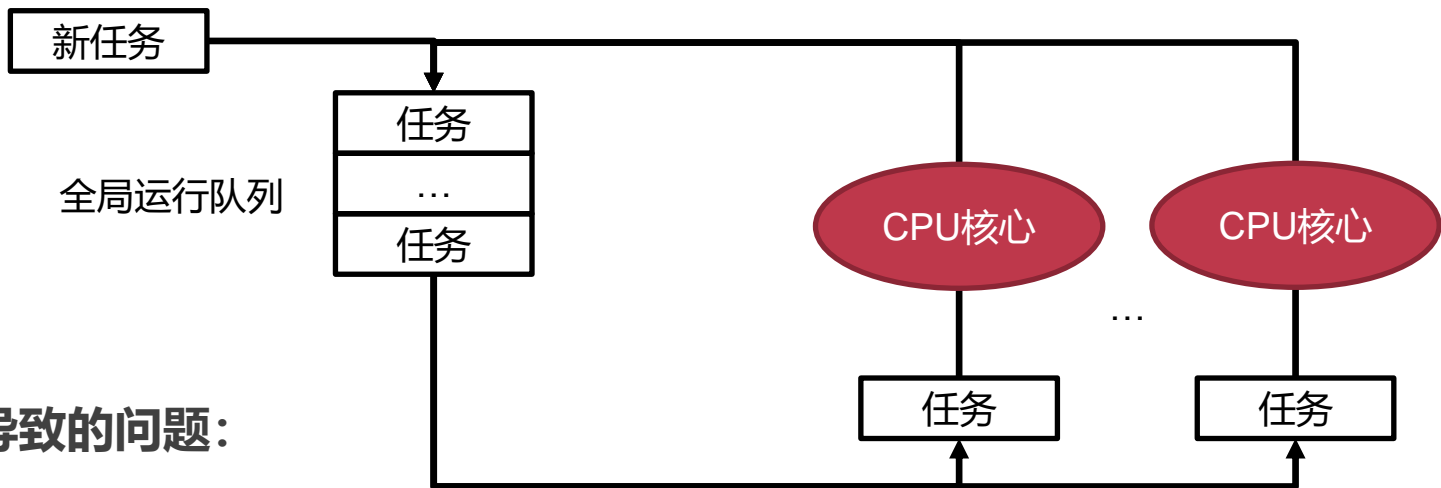
多核调度策略

多核调度需要考虑的额外因素

- 不同线程可以在不同CPU核心上同时运行

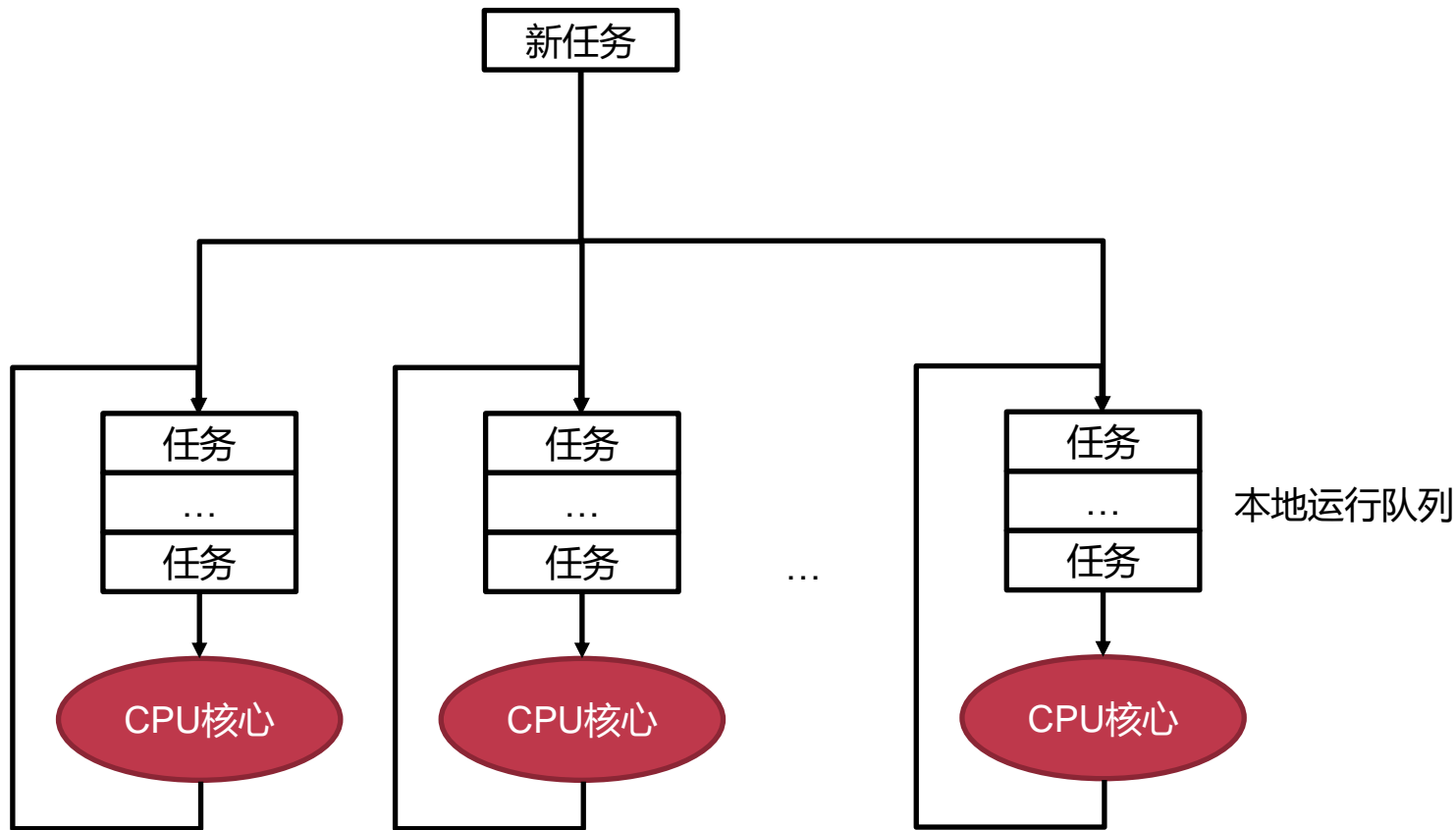


全局运行队列



- 可能导致的问题：
 - 所有CPU核心竞争全局调度器
 - 同一个线程可能在不同CPU上切换
 - 切换开销大：Cache、TLB、...
 - 缓存局部性差

每个CPU核心维护本地运行队列（任务池）



亲和性 (Affinity)

- 程序员如何控制自己程序的行为？
 - 例如，他希望某个线程独占一个CPU核心
- 通过操作系统暴露的任务亲和性接口，可以指定任务能够使用的CPU核心

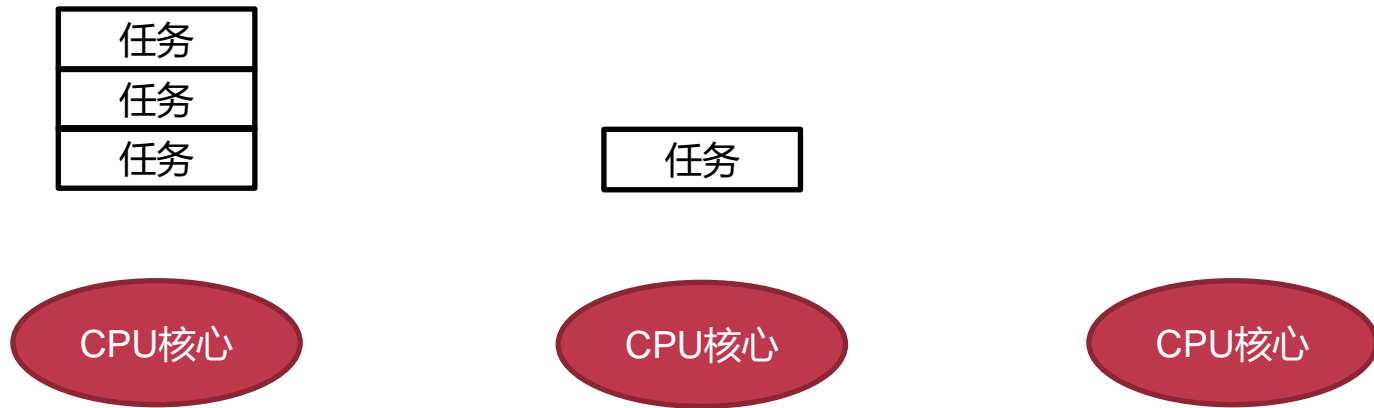
```
1  #include <sched.h>
2
3  int sched_setaffinity(pid_t pid, size_t cpusetsize,
4  |      const cpu_set_t *mask);
5  int sched_getaffinity(pid_t pid, size_t cpusetsize,
6  |      cpu_set_t *mask);
```



指定目标CPU集合的bitmask

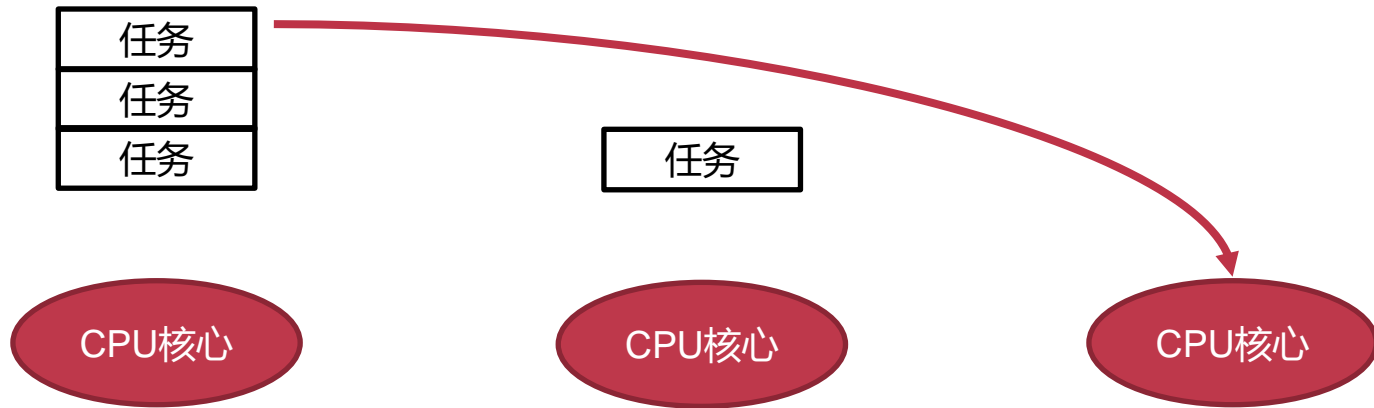
思考：指定线程在CPU上执行的问题

- 负载不均衡



负载均衡 (Load Balance)

- 需要追踪CPU的负载情况
- 将任务从负载高的CPU迁移到负载低的CPU



思考：如何定义任务的负载？

- 根据任务负载定义的不同，负载均衡的效果也不尽相同
- 请思考如下任务负载定义的优劣：
 - 每个CPU核心本地运行队列的长度
 - 优势：实现简单
 - 劣势：不能准确反应当前CPU的负载情况
 - 每个任务单位时间内使用的CPU资源
 - 优势：直观反映当前CPU的负载情况
 - 劣势：引入额外负载追踪开销

调度小结

- 调度指标
- 调度策略
 - FCFS、SJF、RR
 - 优先级调度、MLFQ
 - 公平共享调度
- 多核调度