IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

进程/线程调度

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

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

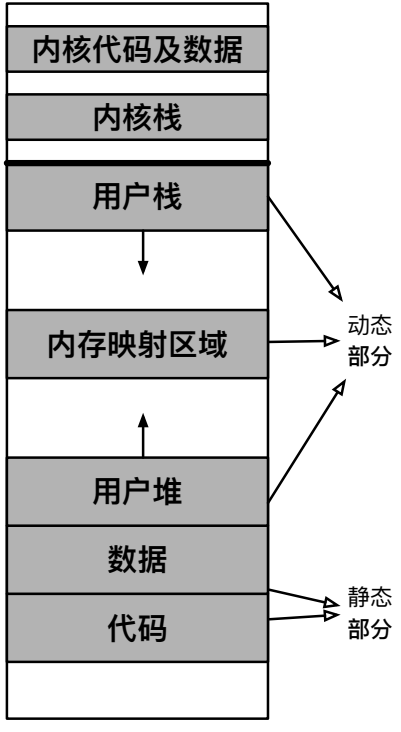
版权声明

- 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

回顾：进程

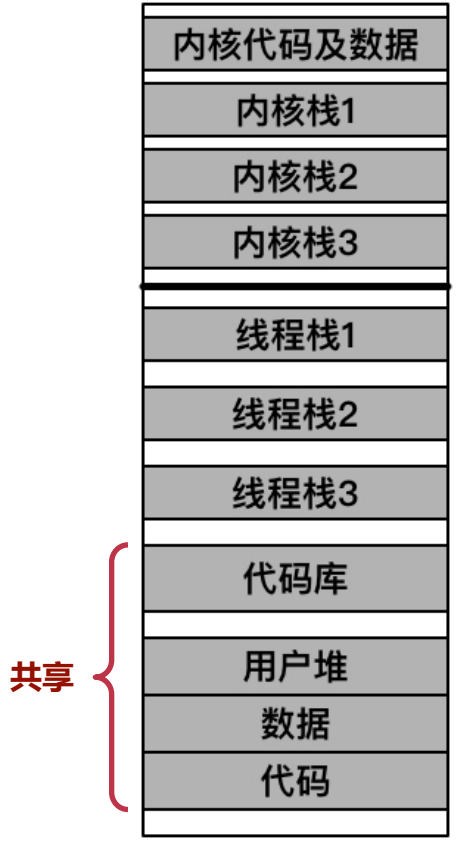
- **进程是计算机程序运行时的抽象**
  - 静态部分：程序运行需要的代码和数据
  - 动态部分：程序运行期间的状态
    （程序计数器、堆、栈……）

- **进程具有独立的虚拟地址空间**
  - 每个进程都具有"独占全部内存"的假象
  - 内核中同样包含内核栈和内核代码、数据



回顾：线程

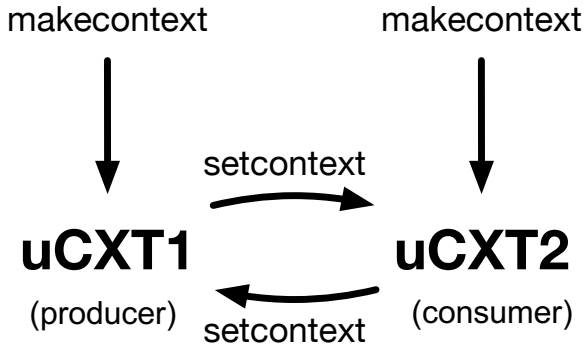
- **线程只包含运行时的状态**
  - 静态部分由**进程**提供
  - 包括了执行所需的**最小**状态（主要是寄存器和栈）
- **一个进程可以包含多个线程**
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行



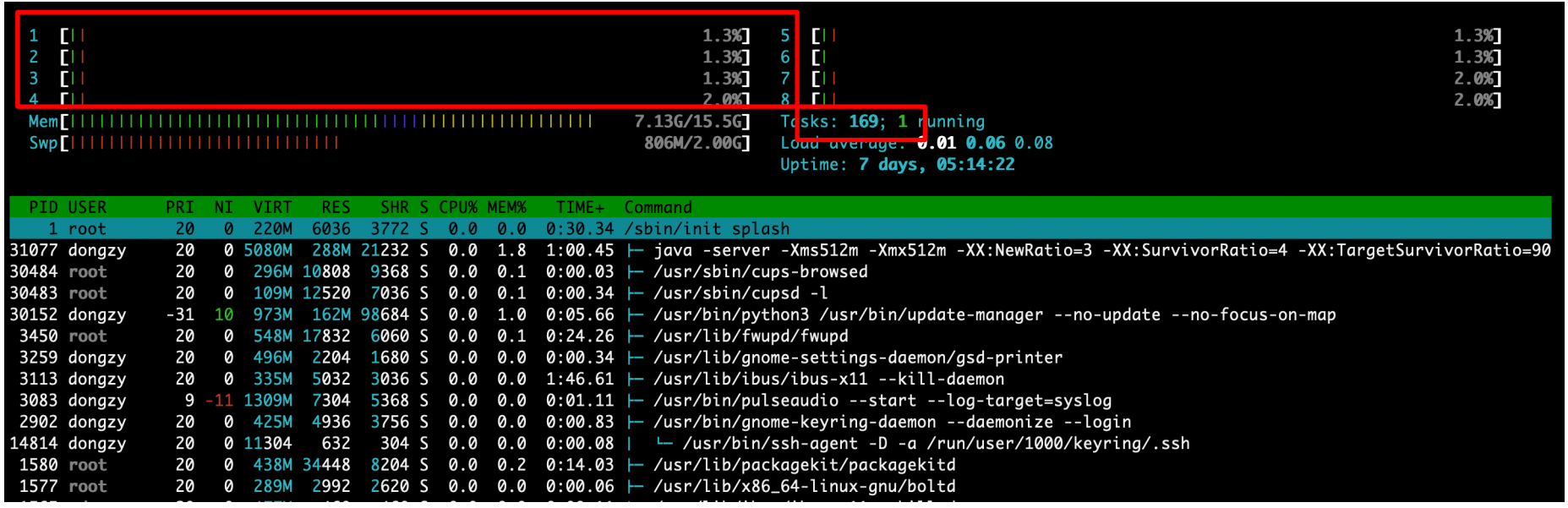
回顾：纤程

- **纤程切换及时**
  - 当生产者完成任务后，可直接用户态切换到消费者
  - 对该线程来说是最优调度（内核调度器很难做到）

- **高效上下文切换**
  - 切换不进入内核态，开销小
  - 即时频繁切换也不会造成过大开销



系统中的任务数远多于处理器数

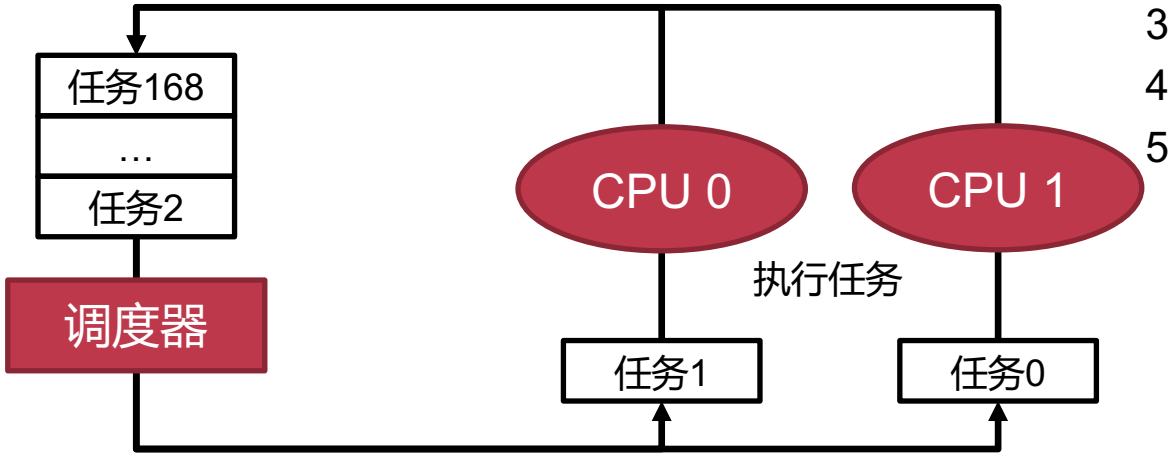


任务（Task）：线程、单线程进程

仅有8个处理器，如何运行169个任务？

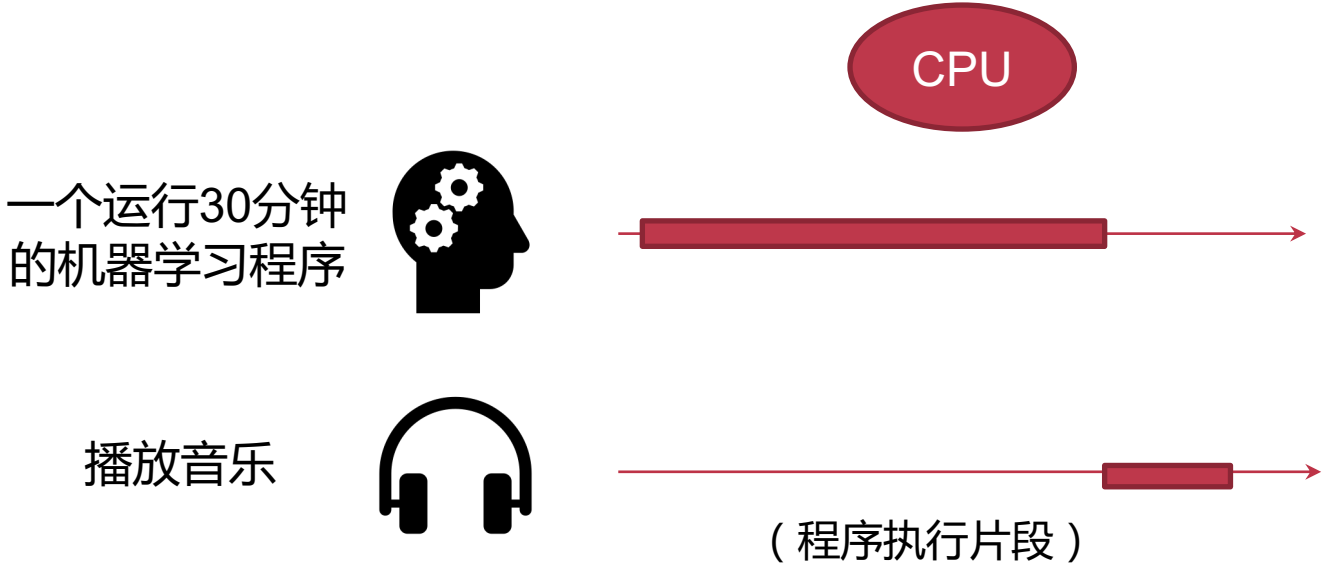
进程/线程调度

执行结束：1）执行时间用尽
2）等待I/O请求
3）睡眠
4）中断
5）…



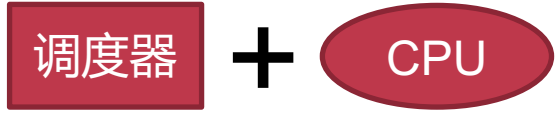
调度决策：1）下一个执行的任务
2）执行该任务的CPU
3）执行的时长

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

调度器让生活更美好



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

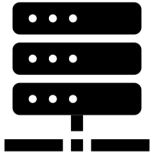
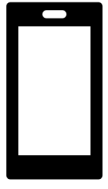
什么是调度？

协调请求对于资源的使用

思考：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	交互式系统	网络服务器	移动设备	实时系统
				
高吞吐量	低响应时间	可扩展性	低能耗	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Oracle☹）**
  - 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
  - ...

策略 V.S. 机制

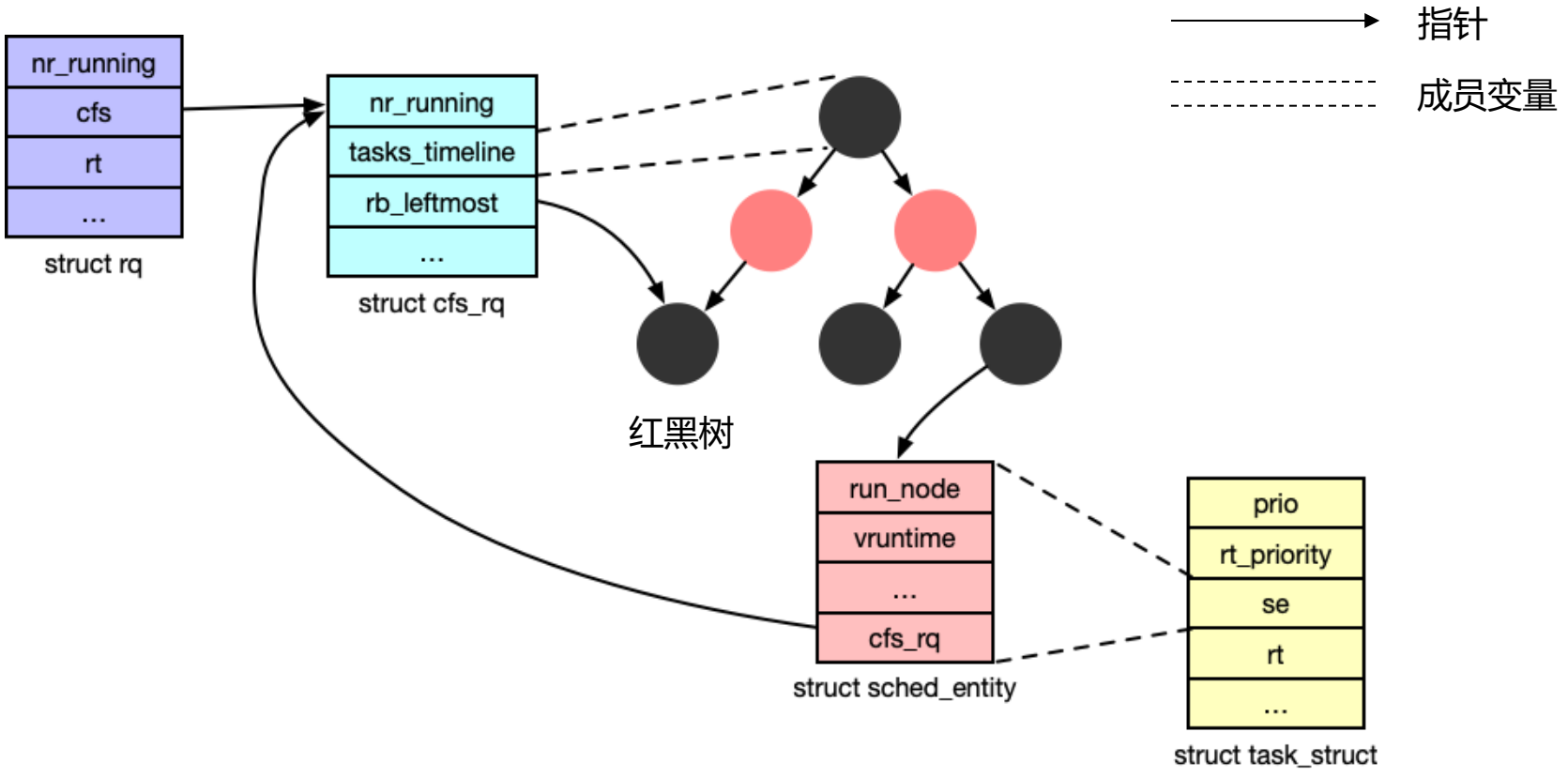
- **策略**
  - 做什么
  - 从上层去分析、解决问题
- **机制**
  - 怎么做
  - 实现某一策略、功能

主题	策略	机制
上课	OS《章节8：调度》	课堂、网课
上课	签到	二维码、点名
科研	写C++代码	VSCode、Sublime
科研	写论文（Latex）	VSCode、Sublime

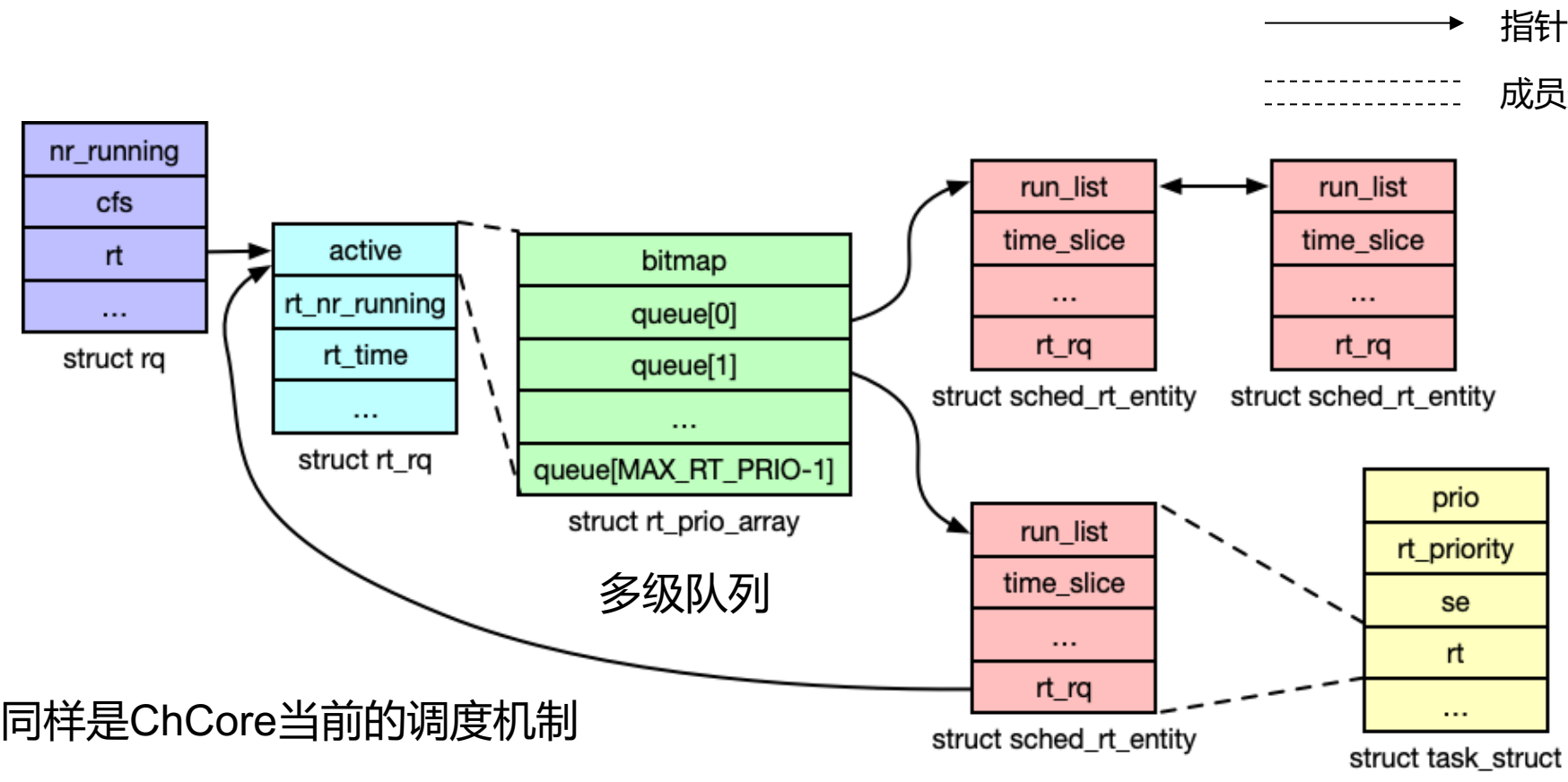
Linux中的调度策略

- **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**
  - Complete Fair Scheduler (CFS)
    - SCHED_OTHER
    - SCHED_BATCH
    - SCHED_IDLE
  - Real-Time Scheduler (RT)
    - SCHED_FIFO
    - SCHED_RR

Linux调度机制：CFS Run Queue



Linux调度机制：RT Run Queue



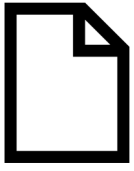
同样是ChCore当前的调度机制

Classical Scheduling

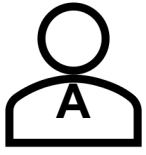
经典调度

CPU调度与提问调度

学霸



同学的问题



同学

学霸调度问题

CPU

任务

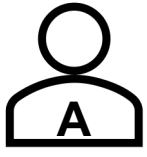
用户

CPU调度任务

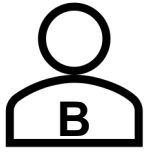
当前假设每个同学只提一个问题

First Come First Served

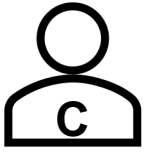
大家排队
先来后到！



得嘞，我第一

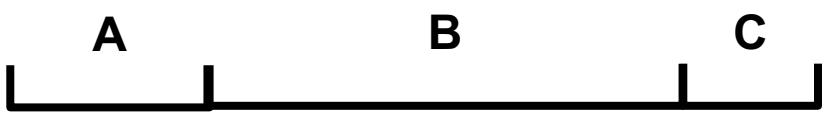


C,先来后到！



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

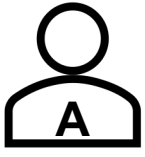


先到先得：简单、直观

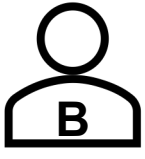
问题：平均周转、响应时间过长

Shortest Job First

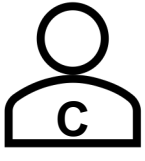
简单的问题先来



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

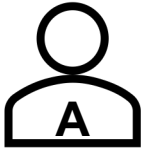
抢占式调度 (Preemptive Scheduling)

- **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

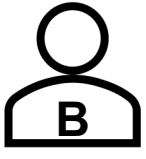
- **通过定时触发的时钟中断实现**

Round Robin (时间片轮转)

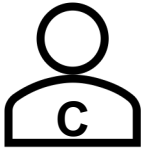
公平起见
每人轮流一分钟！



感觉多等了好久…

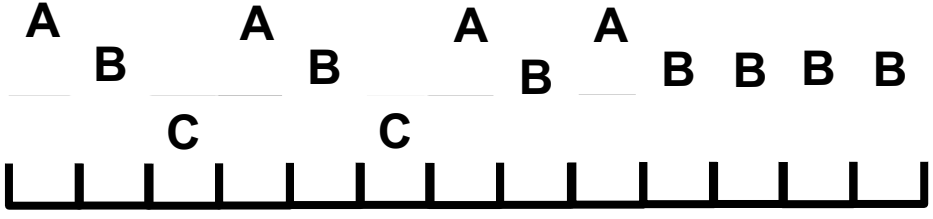


学霸的响应时间短了好多



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

思考：

- **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？

Priority Scheduling

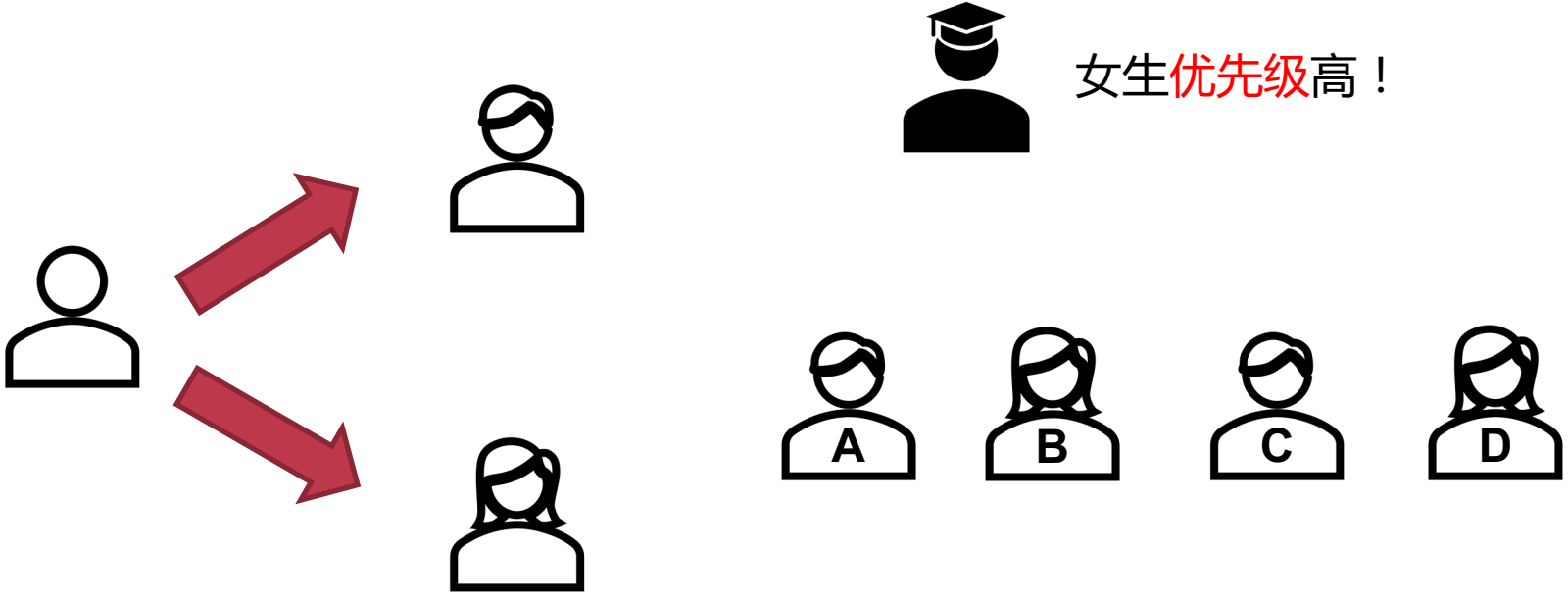
优先级调度

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**

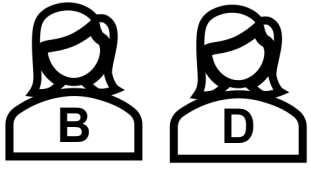
添加条件：优先级



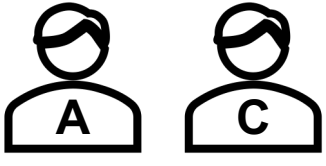
Multi-level Queue

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：

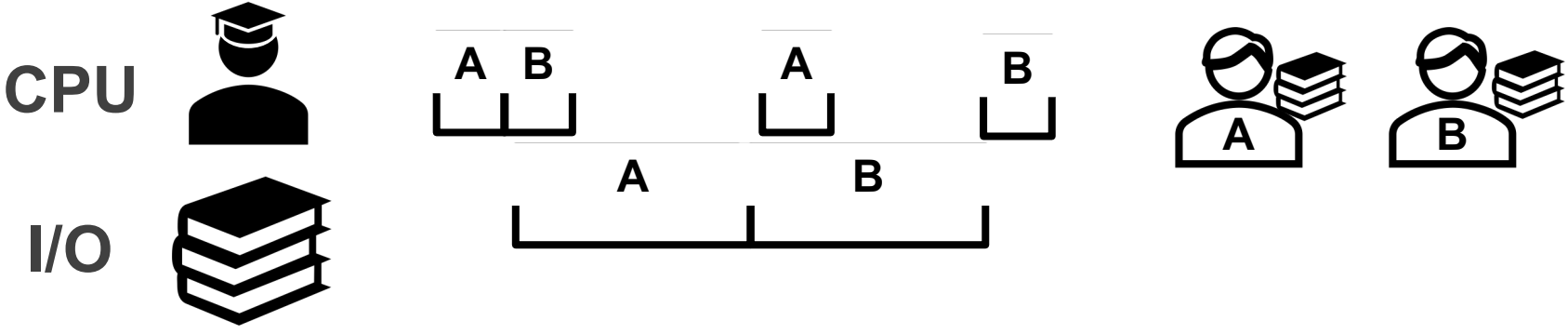
1）维护多个优先级队列

2）高优先级的任务优先执行

3）同优先级内使用Round Robin调度

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**

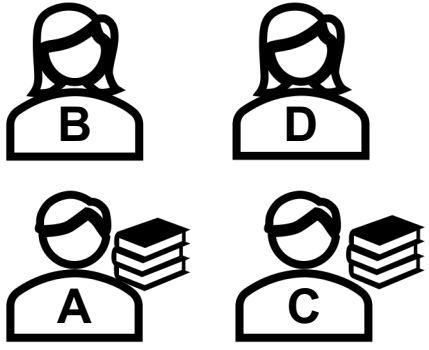


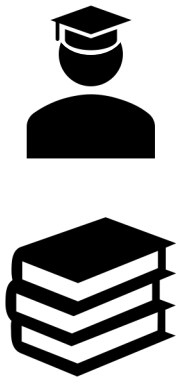
问题1：低资源利用率

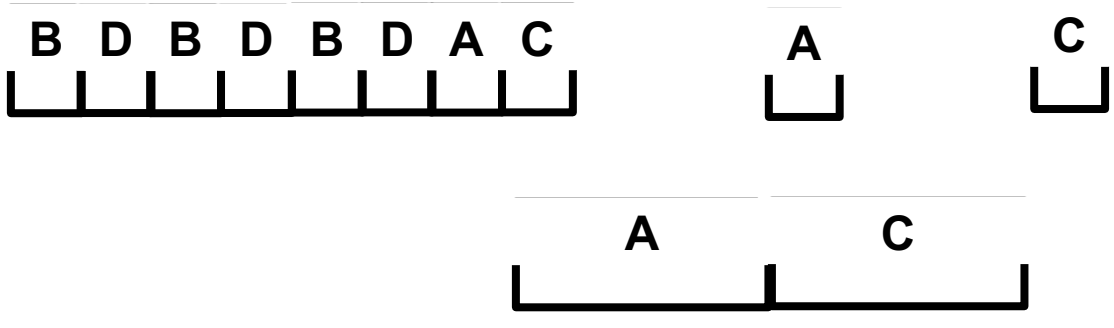
问题：
多种资源（学霸和OS书）
没有同时利用起来

优先级0（高）

优先级1（低）







思考：优先级的选取

- **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

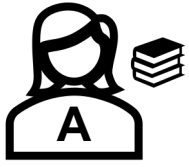
问题2：优先级反转

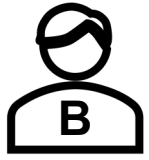
- **高、低优先级任务都需要独占共享资源**
  - 共享资源
    - 存储
    - 硬件
    - OS书
    - …
  - 通常使用信号量、互斥锁实现独占
- **低优先任务占用资源 -> 高优先级任务被阻塞**

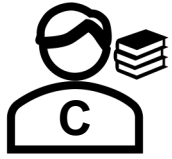
问题2：优先级反转

优先级：A>B>C

问题：
A被C占有的资源阻塞
优先级较低的B先于A学习







1.申请OS书成功

2. 抢占C
申请OS书失败
等待

3.B优先级高于C
可以向学霸学习

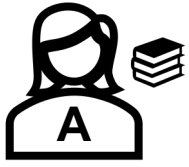
解决方法：优先级继承

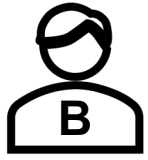
优先级：A>B>C

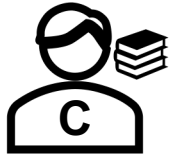
解决方案：

A暂时将优先级转移给C

让C尽快归还OS书







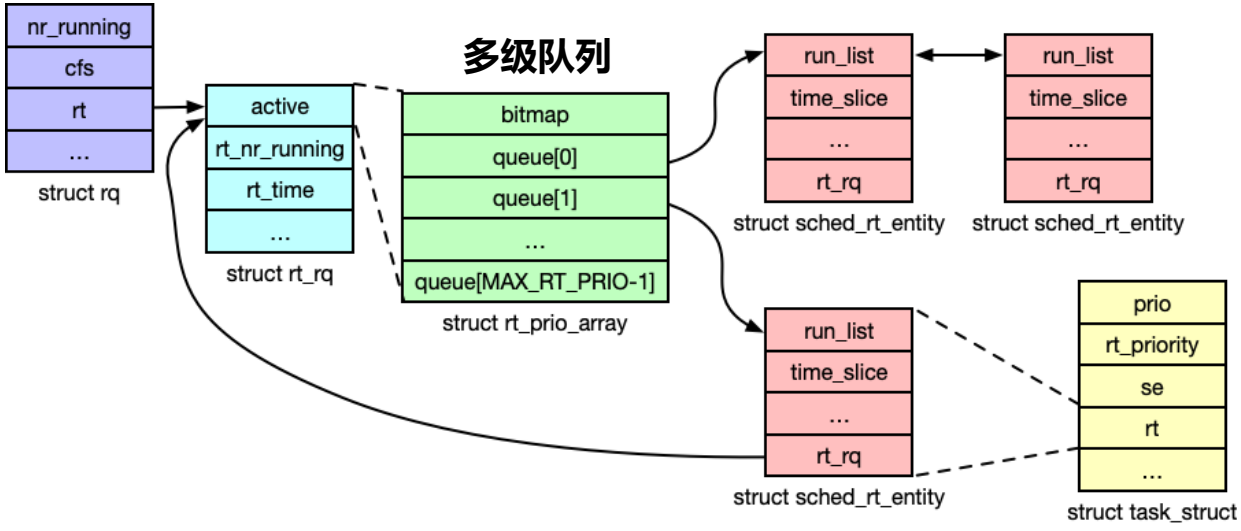
1.申请OS书成功

2. 抢占C
转移优先级给C

3.归还OS书
返回优先级

4. 申请OS书成功
继续学习

Linux Real-Time Scheduler



- **Linux Real-Time Scheduler，使用Multi-level Queue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

思考：优先级

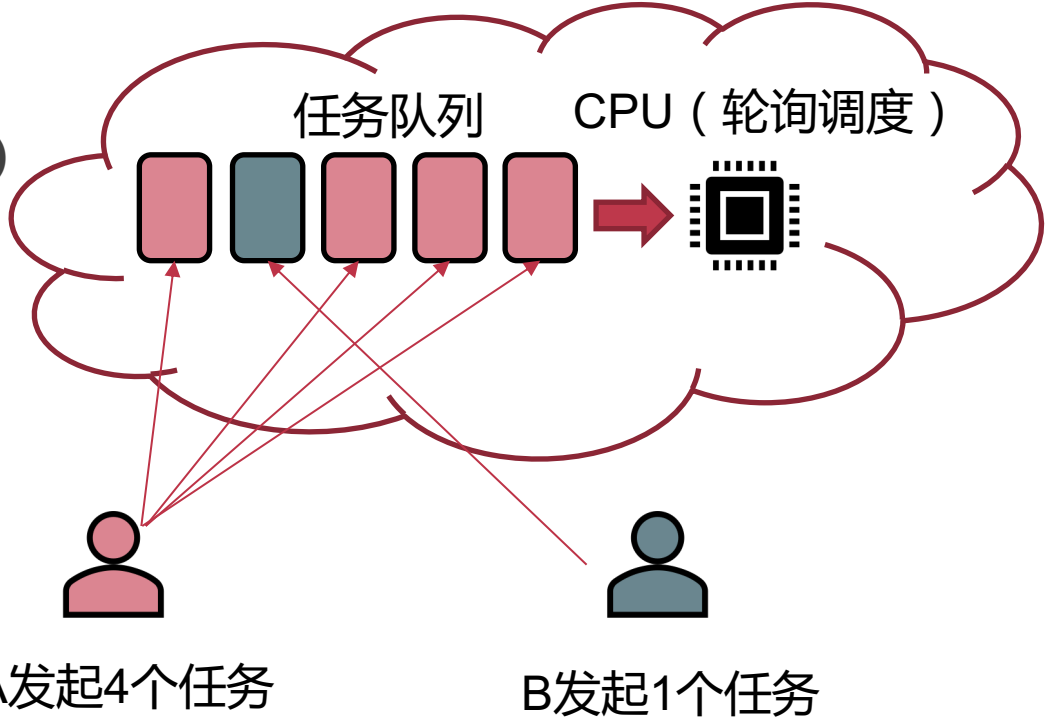
- **以下这些调度策略算不算优先级调度？**
  - First Come First Served
  - Shortest Job First
  - Round Robin

Fair-Share Scheduling

公平共享调度

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Round Robin调度**
  - A占用80%CPU时间
  - B占用20%CPU时间

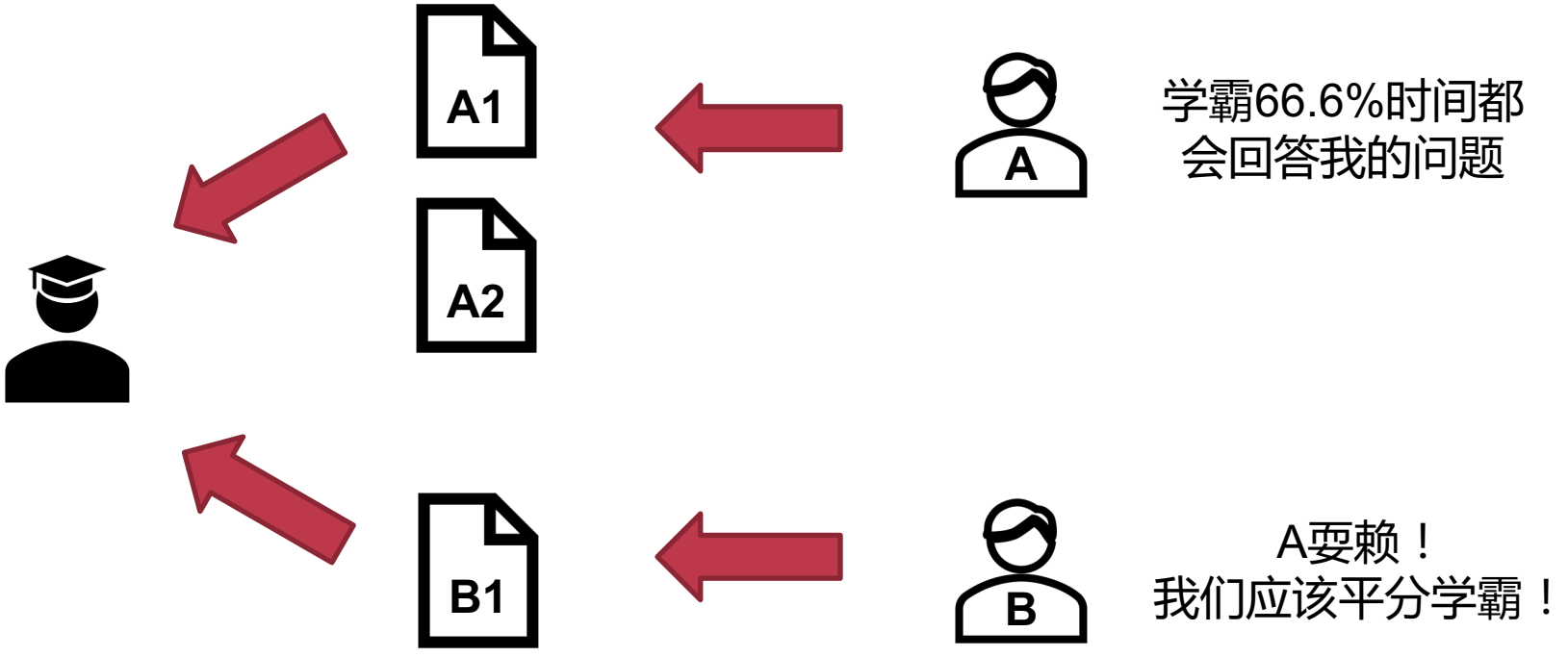


公平共享

- **每个用户占用的资源是成比例的**
  - 而非被任务的数量决定

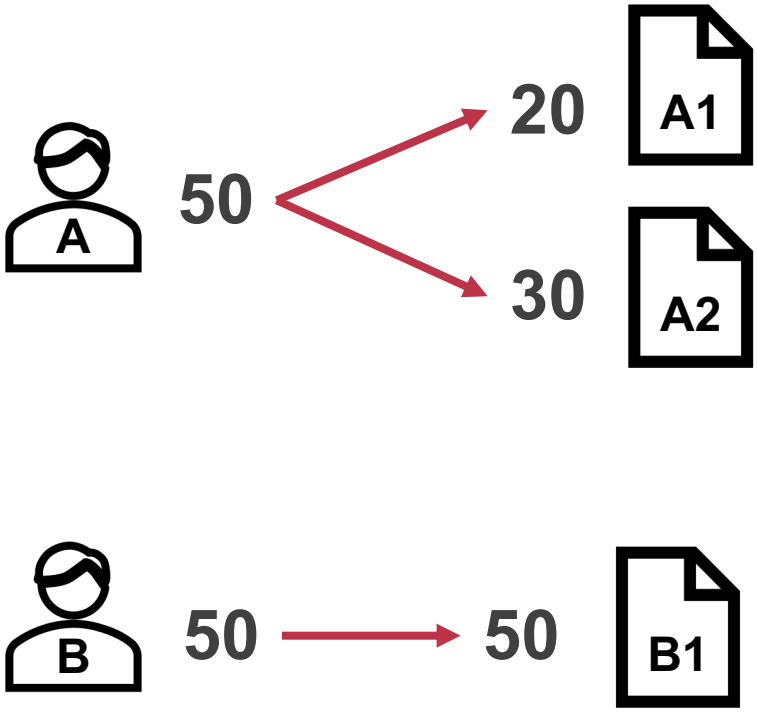
- **每个用户占用的资源是可以被计算的**
  - 设定"权重值"以确定相对比例（绝对值不重要）
  - 例：权重为4的用户使用资源，是权重为2的用户的2倍

添加条件：一个同学会问多个问题



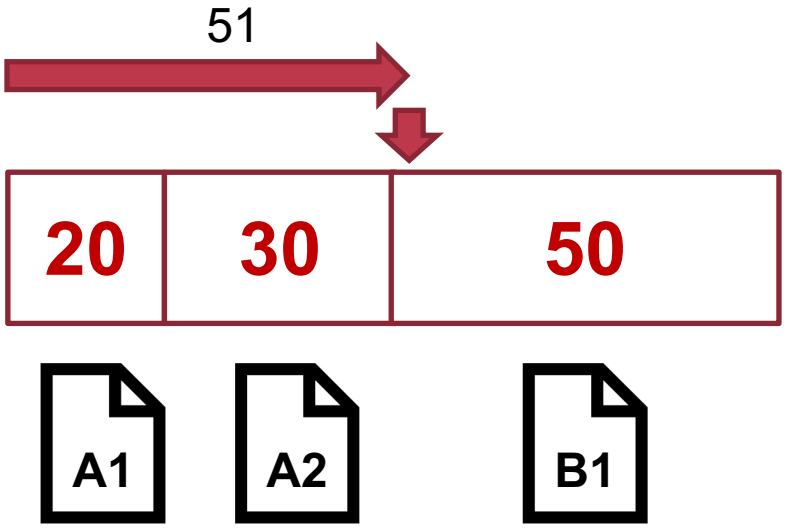
方法：使用"ticket"表示任务的权重

- $ticket$：每个问题对应的权重
- T ：ticket的总量
- 问题A1可占用学霸时间的比例
  - $\frac{ticket_{A1}}{\mathrm{T}} = \frac{20}{100} = \frac{1}{5}$
- 同学A可占用学霸时间的比例
  - $\frac{ticket_{A}}{\mathrm{T}} = \frac{\mathrm{ticket_{A1}+ticket_{A2}}}{\mathrm{T}} = \frac{50}{100} = \frac{1}{2}$



一种公平共享的实现：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 → 调度B1

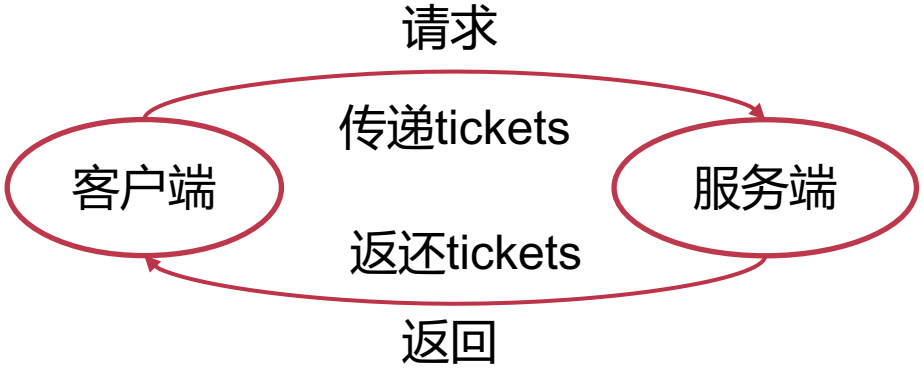



```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task.ticket
    if (R < sum) {
        break
    }
}
schedule()
```


Lottery Scheduling: Flexible Proportional-Share Resource Management. https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf

Ticket Transfer

- **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**



思考：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

思考：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

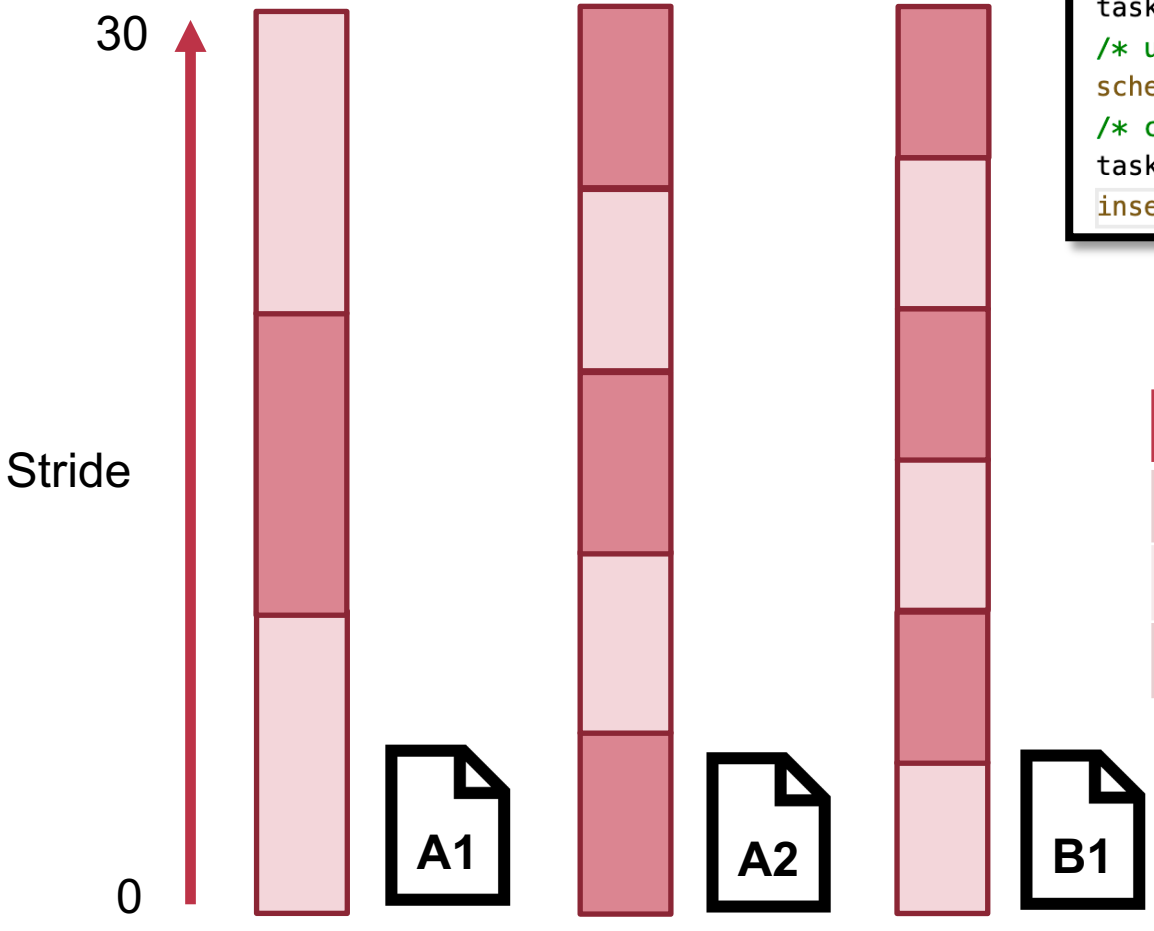
Stride Scheduling

- **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

Stride Scheduling




```
/* 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);
```

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5

公平共享调度

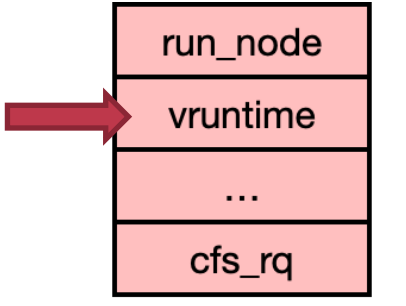
	Lottery Scheduling	Stride Scheduling
调度决策生成	随机	确定性计算
任务实际执行时间与预期的差距	大	小

预期——根据任务权重计算的执行时间期望

Linux Complete Fair Scheduler

- CFS（完全公平调度器）
- 使用类似Stride Scheduling的公平共享调度策略

CFS的 vruntime 等价于
Stride Scheduling的pass



可参考： https://www.linuxjournal.com/node/10267

Real-Time Scheduling

实时调度

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
  - 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

Earliest Deadline First

- **任务**
  - C：所需执行时间
  - P：任务触发的时间周期
    - 假设P同时是任务截止时间

$$\left(\sum_{i=1}^{n}\frac{C_i}{P_i} \leq 1\right)$$

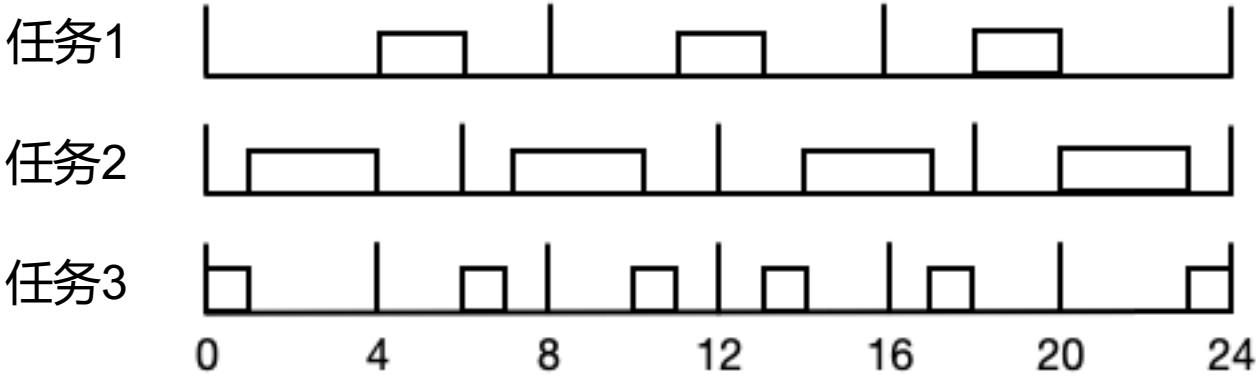
反映了CPU利用率

- **当实时任务满足条件：**
  - 这些任务对EDF是可调度的
    - 所有任务在截止时间前完成

Earliest Deadline First

- **每次调度截止时间最近的任务**
- **EDF是动态算法**
  - 无需预知执行时间、任务周期
- **在任务可调度的情况下能够实现最优调度**

	执行时间	任务周期/截止时间
任务1	2	8
任务2	3	6
任务3	1	4

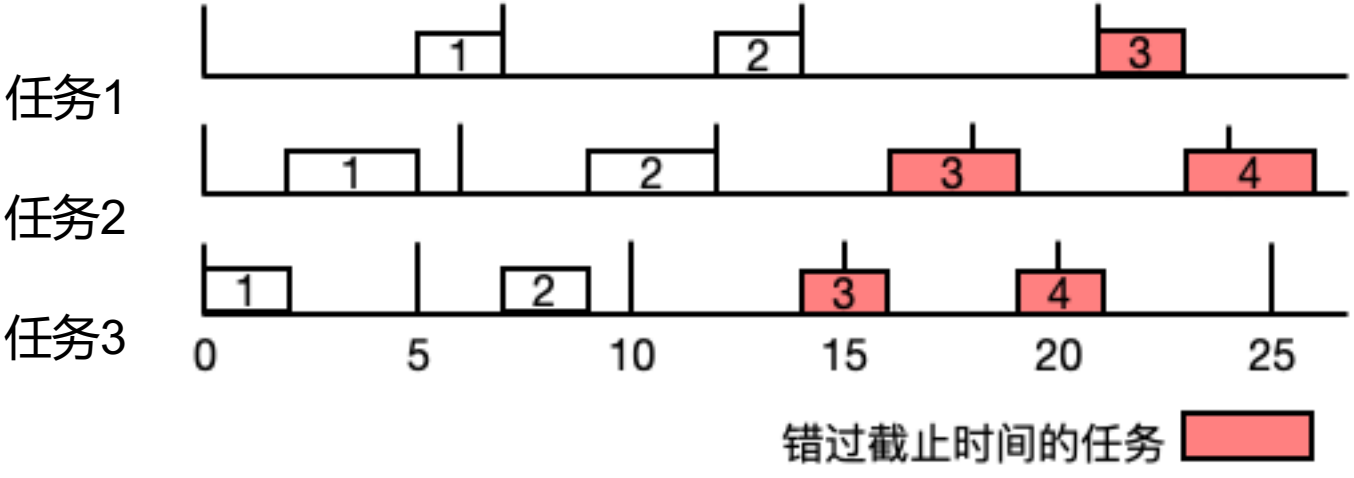


$$\sum_{i=1}^{n} \frac{C_i}{P_i} = 1$$

多米诺效应：需要进行可调度性分析

- 在任务不可调度时，会造成多数任务都错过截止时间

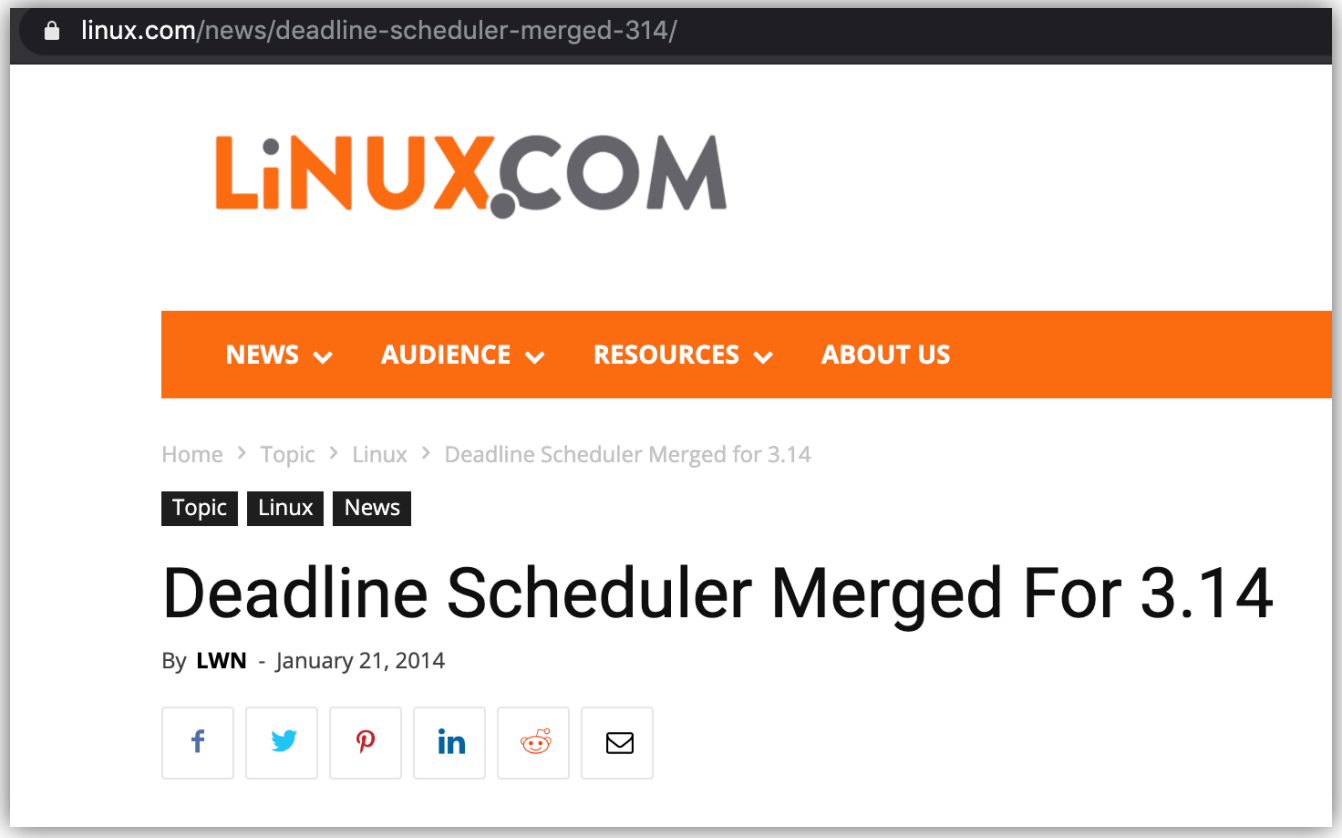
	执行时间	任务周期/截止时间
任务1	2	7
任务2	3	6
任务3	2	5



$$\sum_{i=1}^{n} \frac{C_i}{P_i} > 1$$

方框中数字代表第n个周期的任务

基于Deadline的实时调度策略被Linux 3.14整合

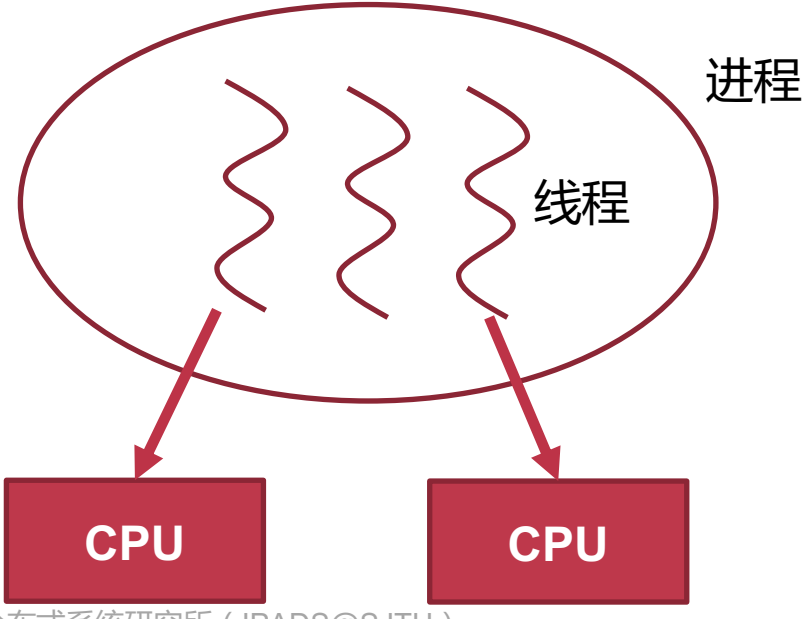


Multicore Scheduling Policy

多核调度策略

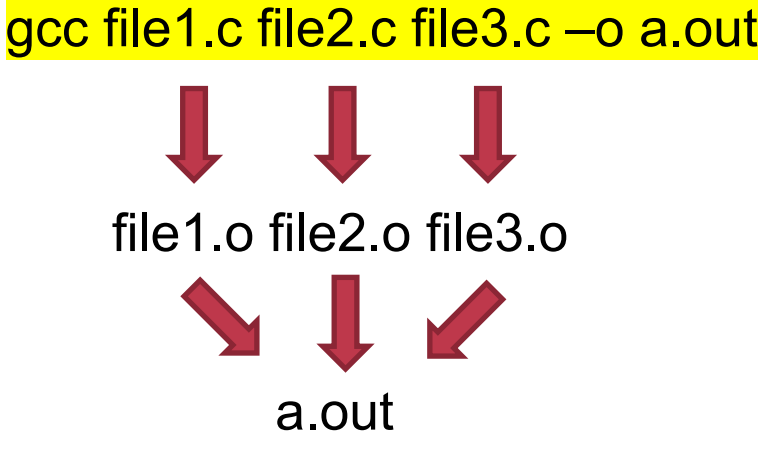
多核调度需要考虑的额外因素

- 一个进程的不同线程可以在不同CPU上同时运行



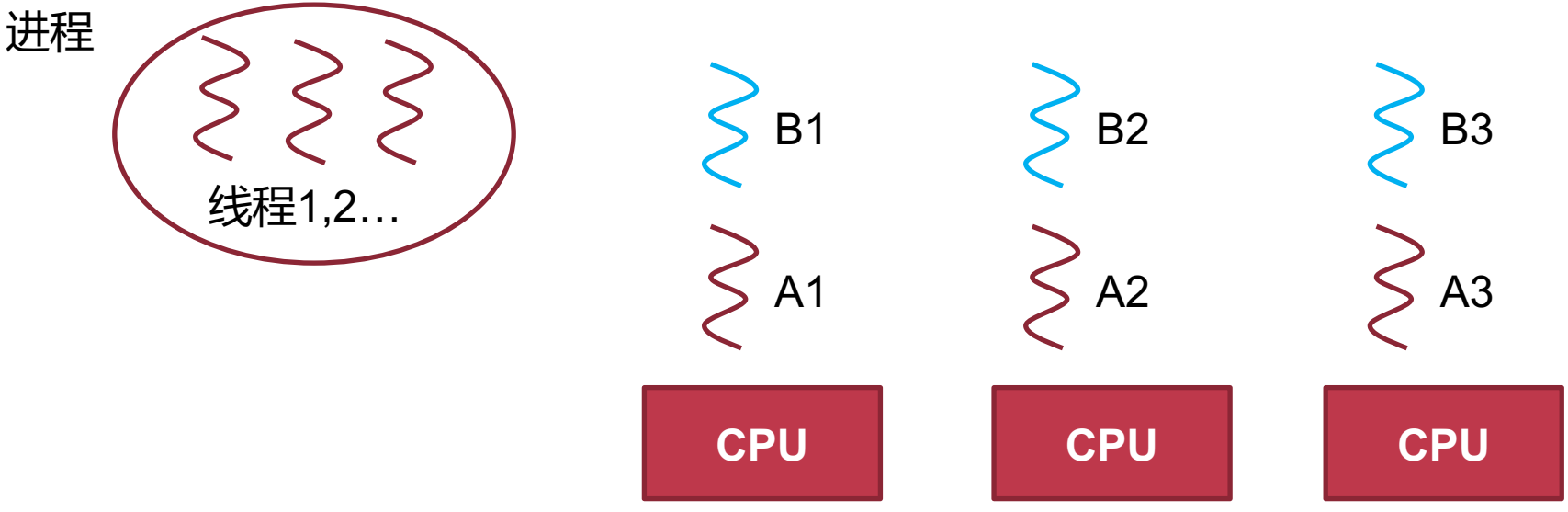
同一个进程的线程很可能有依赖关系

- **例：GCC 编译a.out文件**
- **每个线程编译一个文件（.c到.o）**
- **线程间依赖：**
  - 所有.o生成完才能进入下一步操作



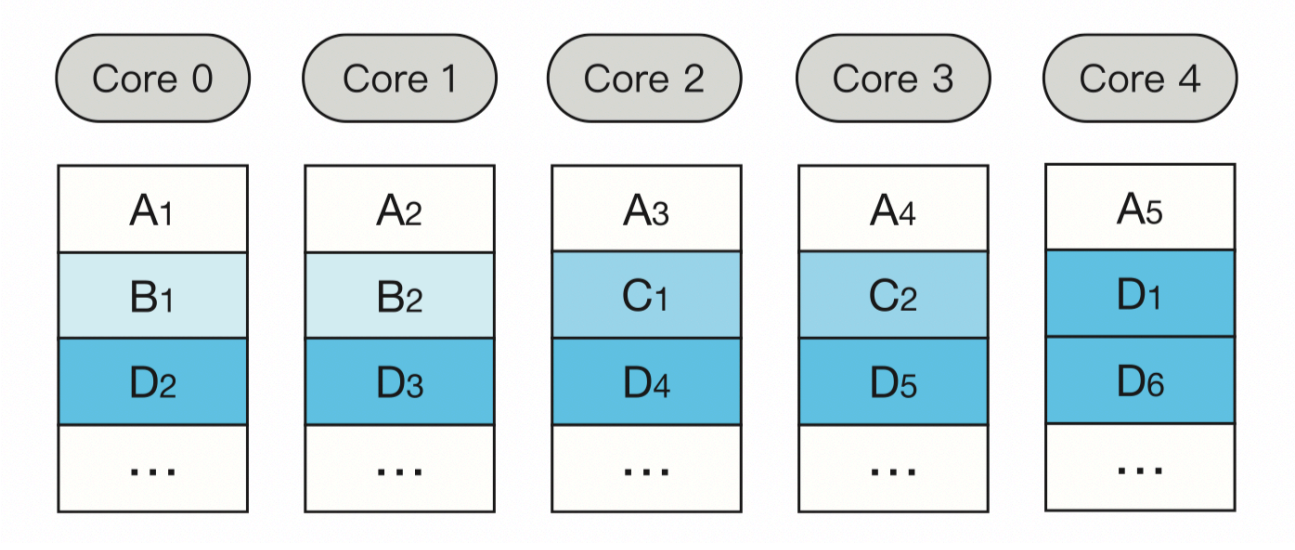
群组调度：Gang Scheduling

- 在多个CPU上同时执行一个进程的多个线程

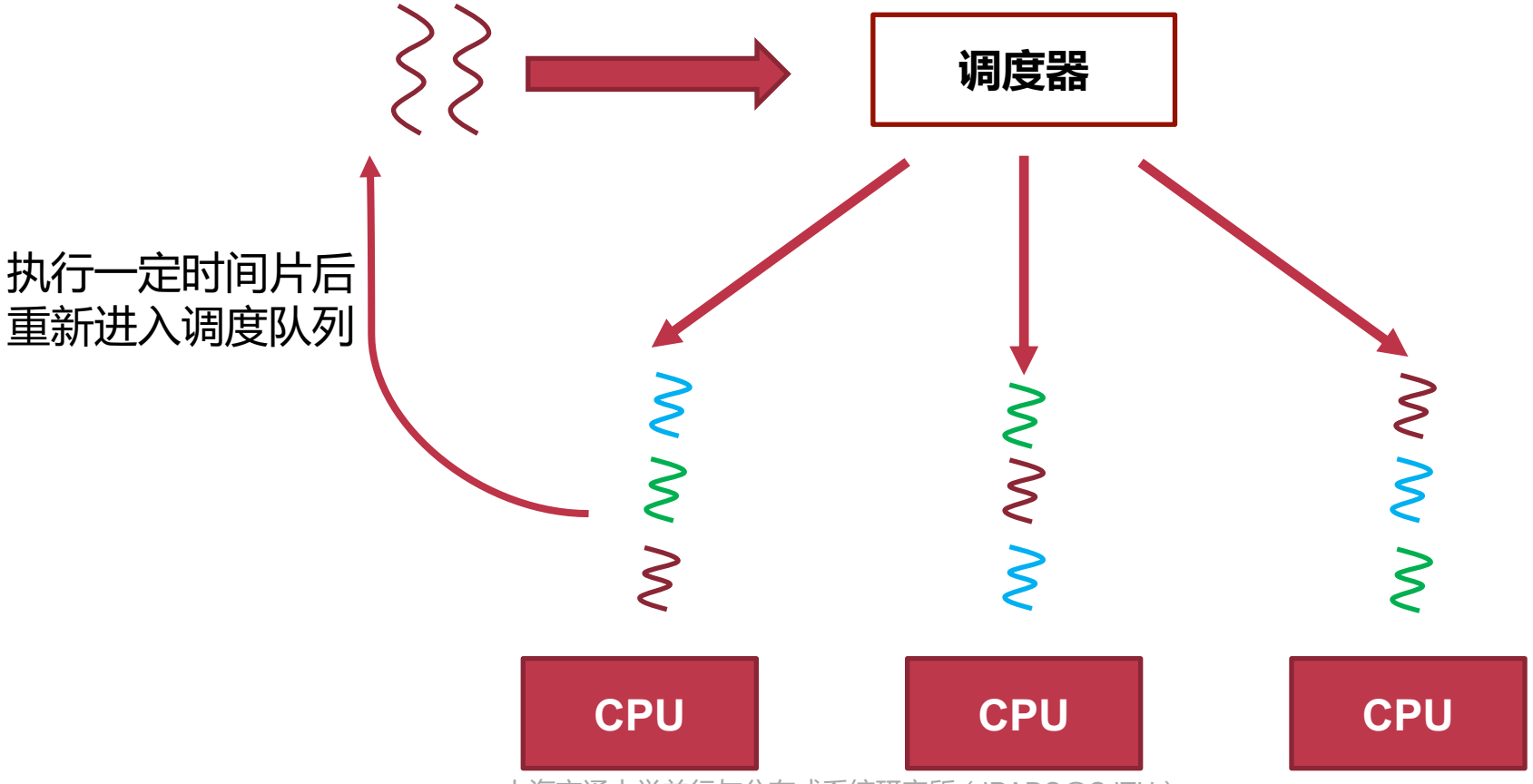


群组调度：Gang Scheduling

- **4组任务A、B、C、D**
  - 组内任务都是关联任务，需要尽可能同时执行

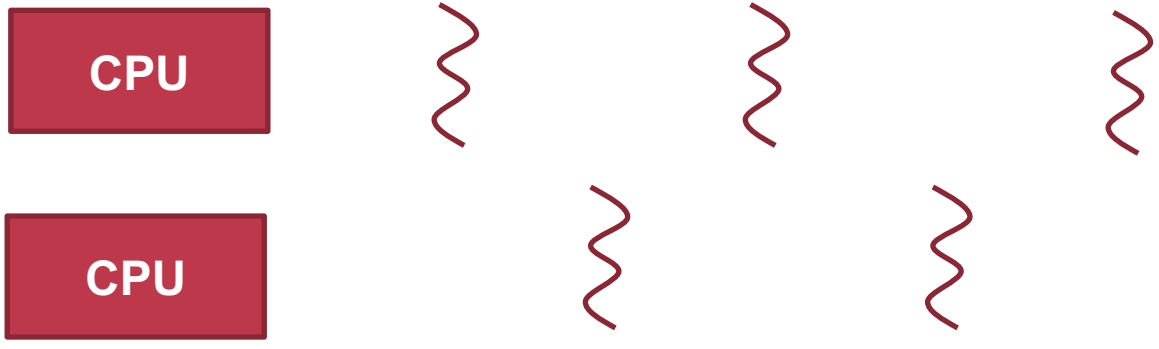


全局使用一个调度器的问题

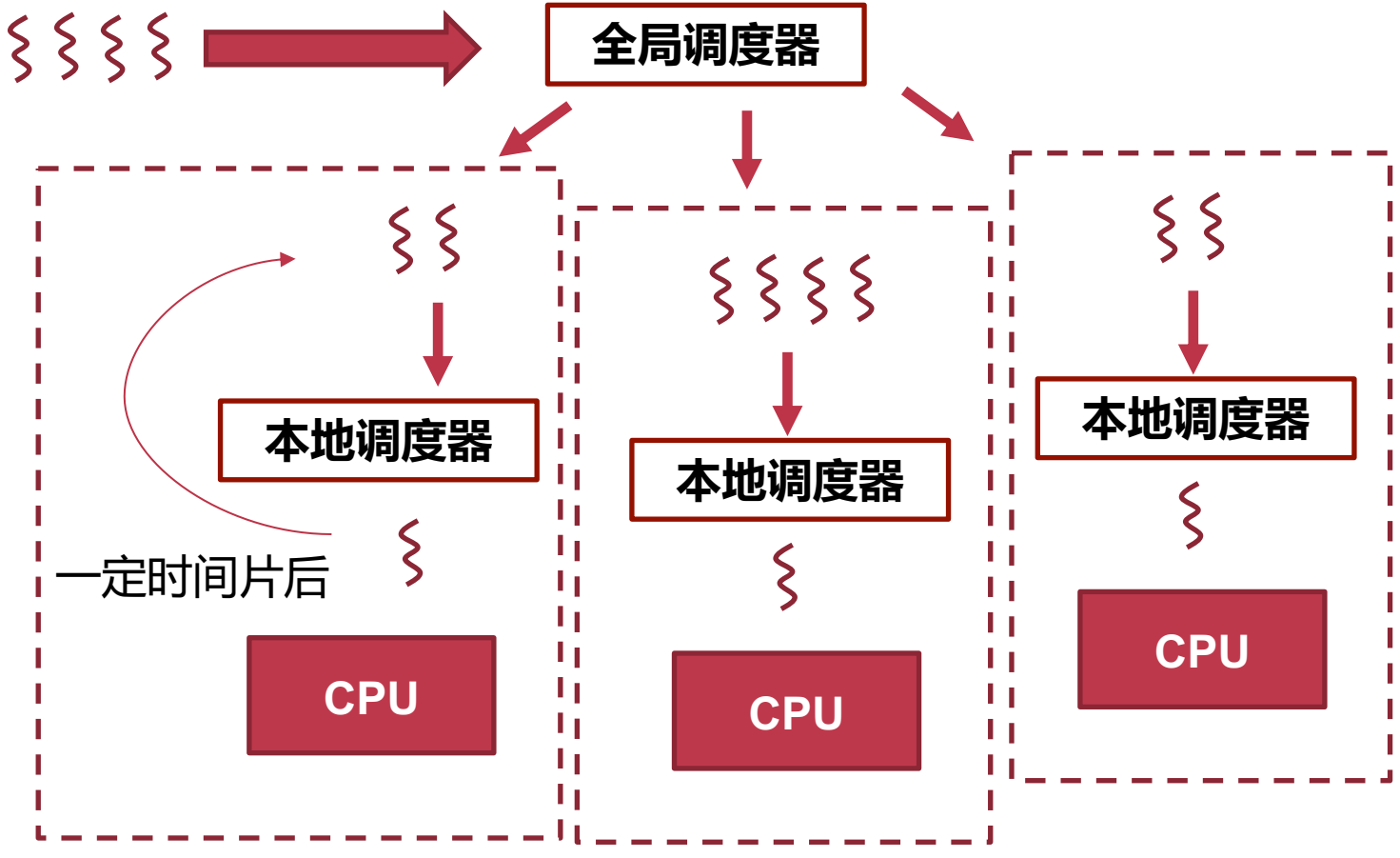


全局使用一个调度器的问题

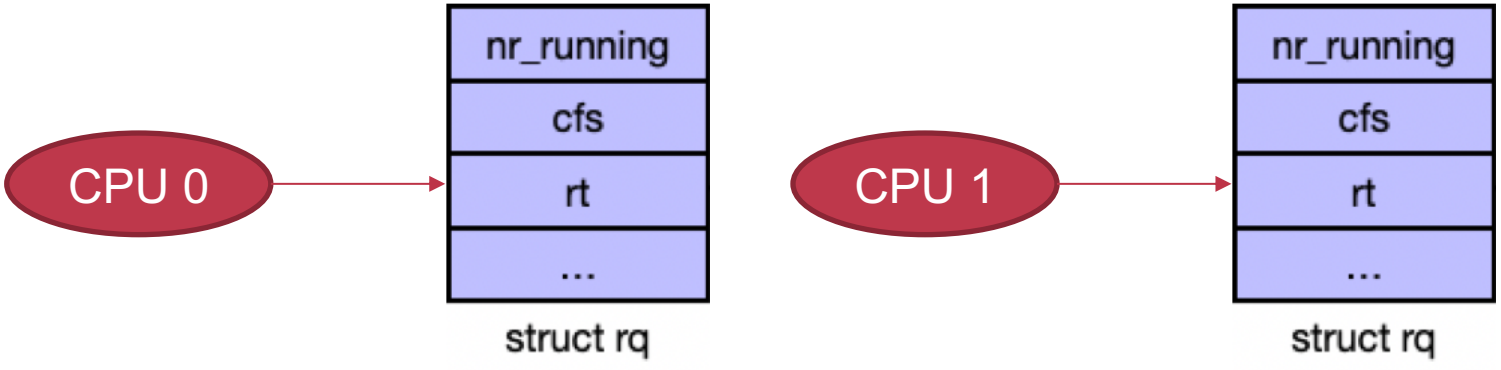
- **所有CPU竞争全局调度器**
- **同一个线程可能在不同CPU上切换**
  - 切换开销大：Cache、TLB、…
  - 缓存局部性差



Two-level Scheduling



ChCore & Linux的多核调度



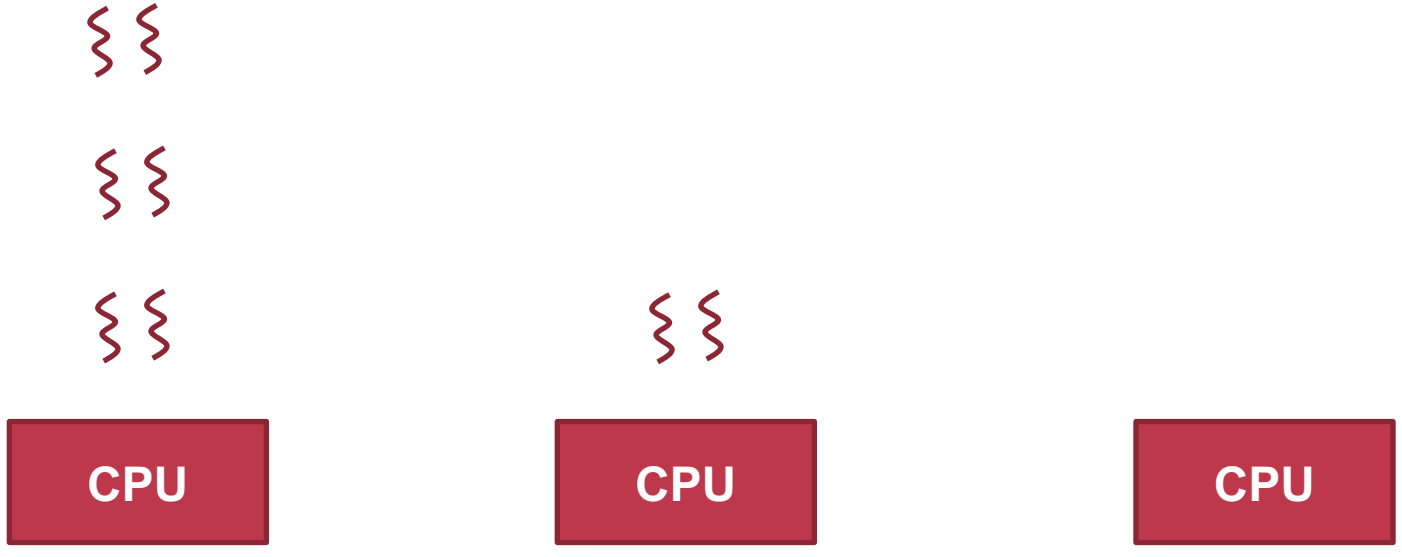
ChCore & Linux同样使用Two-level Scheduling的架构

每个CPU有各自的本地调度器和runq

负载追踪与负载均衡

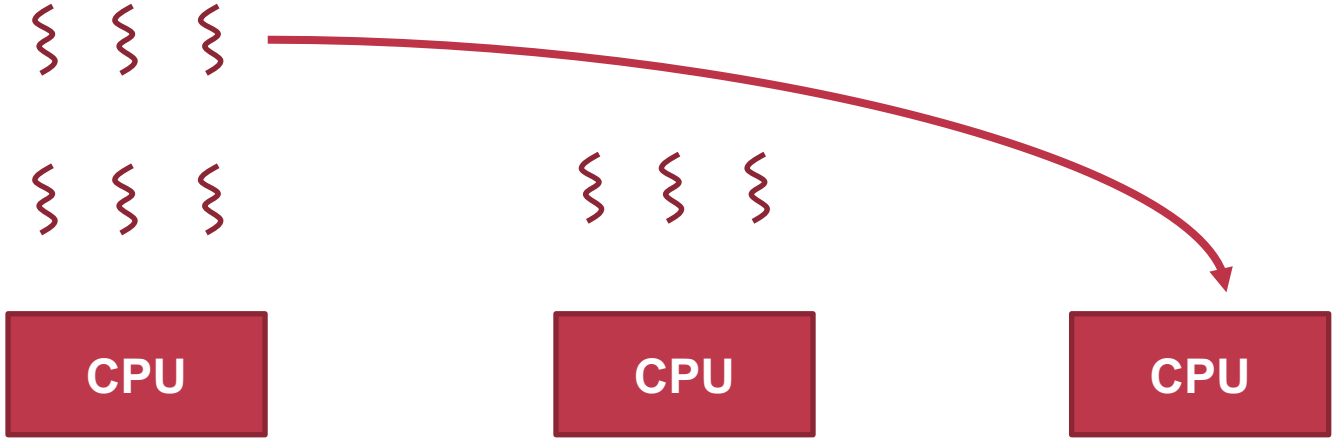
思考：指定线程在CPU上执行的问题

- 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



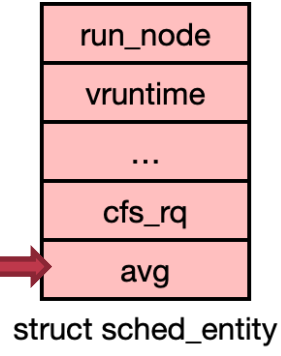
如何进行准确的负载追踪？

- **方法一：以运行队列为粒度追踪负载**
  - 运行队列长，意味着负载高
  - 但从一个高负载的运行队列中，到底选取哪个任务迁移呢？没有相关信息，因此不够准确
- **方法二：以调度实体为粒度追踪负载**
  - 以调度实体（单个任务）为粒度记录负载
  - Linux's PELT: Per Entity Load Tracking，从Linux 3.8开始

PELT: Per Entity Load Tracking

- **记录每个调度实体对负载的贡献**
  - 每1024μs周期，记录线程处于可运行状态的时间x
  - 线程t在第i个周期内的负载为:
    - $L_{t,i} = \frac{x}{1024} * CpuScaleFactor$
    - 为什么需要$CpuScaleFactor$ ？
      - 负载是在每个CPU上独立计算的，需要有一个统一的标准
      - 性能高的CPU，Factor也更高

每个调度实体都会在avg成员中记录自己的负载



任务总负载的计算

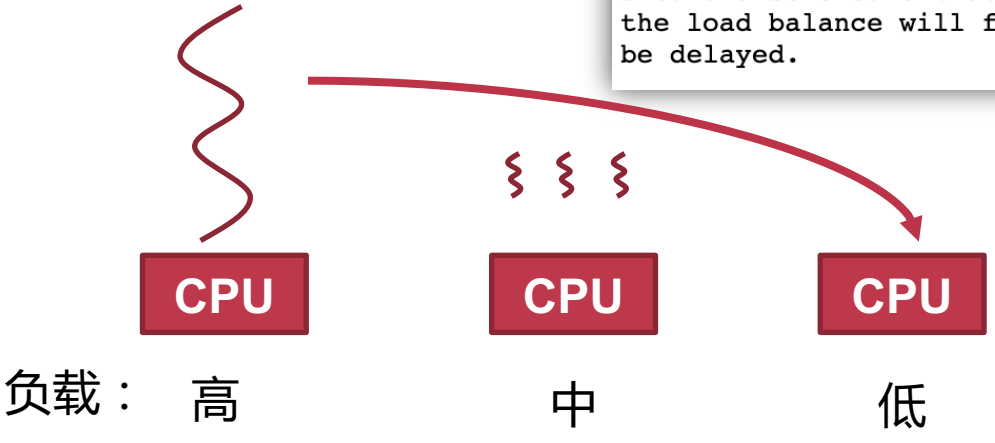
- $Load_t = L_{t,i} + L_{t,i-1} + \cdots + L_{t,0}$?
  - 不合适，任务的负载是变化的
    - 不能因为任务连续运行一个星期，就说它当前负载高
  - 最近的负载权重应该更大

- $Load_t = L_{t,i} + \gamma * L_{t,i-1} + \gamma^2 * L_{t,i-2} \ldots + \gamma^i L_{t,0}$
  - Linux设置衰减因子$\gamma^{32} = 0.5$
    - 当前负载在任务经过32个周期后减半
  - 实际计算时：$Load_t = L_{t,i} + \gamma * Load_t$

实现高效的调度器绝非易事

```
From: Vincent Guittot <vincent.guittot@linaro.org>
To: mingo@redhat.com, peterz@infradead.org, juri.lelli@redhat.com,
        dietmar.eggemann@arm.com, rostedt@goodmis.org,
        bsegall@google.com, mgorman@suse.de,
        linux-kernel@vger.kernel.org
Cc: Vincent Guittot <vincent.guittot@linaro.org>
Subject: [PATCH] sched/fair: improve spreading of utilization
Date: Thu, 12 Mar 2020 17:54:29 +0100
Message-ID: <20200312165429.990-1-vincent.guittot@linaro.org> (raw)

During load_balancing, a group with spare capacity will try to pull some
utilizations from an overloaded group. In such case, the load balance
looks for the runqueue with the highest utilization. Nevertheless, it
should also ensure that there are some pending tasks to pull otherwise
the load balance will fail to pull a task and the spread of the load will
be delayed.
```



负载均衡

从负载最高的CPU拉取任务执行

但如果该CPU当前只有一个任务呢？

实现高效的调度器绝非易事

```
diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index c7aaae2..783356f 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -9313,6 +9313,14 @@ static struct rq *find_busiest_queue(struct lb_env *env,
                case migrate_util:
                        util = cpu_util(cpu_of(rq));

+                       /*
+                        * Don't try to pull utilization from a CPU with one
+                        * running task. Whatever its utilization, we will fail
+                        * detach the task.
+                        */
+                       if (nr_running <= 1)
+                               continue;
+
                        if (busiest_util < util) {
                                busiest_util = util;
                                busiest = rq;
```

```
Below are the average results for 15 iterations on an arm64 octo core:
sysbench --test=cpu --num-threads=8  --max-requests=1000 run

                              tip/sched/core  +patchset
total time:                   172ms           158ms
per-request statistics:
         avg:                   1.337ms         1.244ms
         max:                  21.191ms        10.753ms
```

调度小结

- 策略 V.S. 机制
- 经典调度：FCFS、SJF、RR
- 优先级调度：Multi-level Queues、优先级的选取
- 公平共享调度：Lottery、Stride
- 实时调度：EDF

下次课内容

- 进程间通信