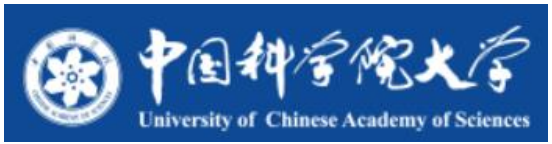
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

处理器调度

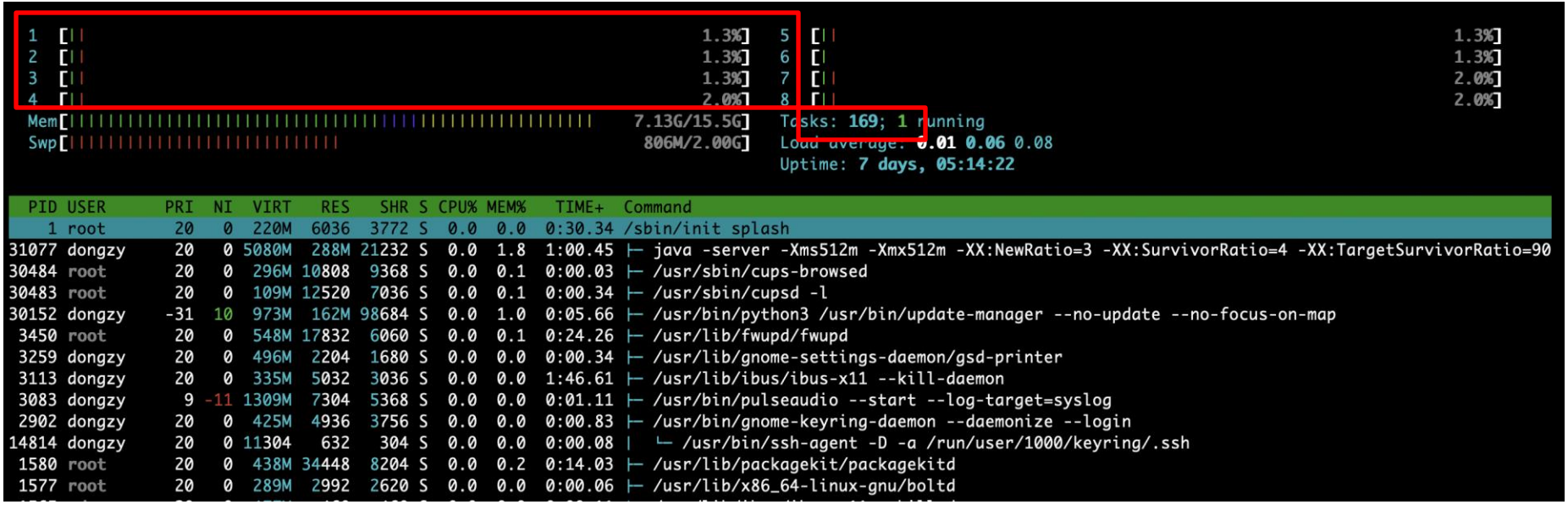
郑晨

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



系统中的任务数远多于处理器数

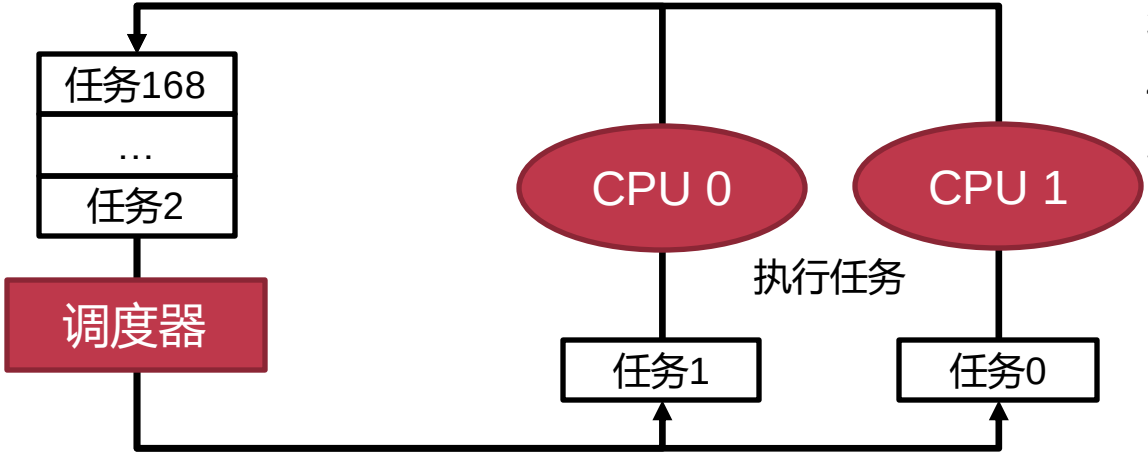


任务（Task）：线程、单线程进程

仅有8个处理器，如何运行169个任务?

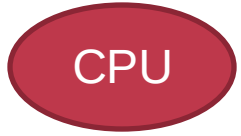
进程/线程调度

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



一个运行30分钟
的机器学习程序

播放音乐

（程序执行片段）

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

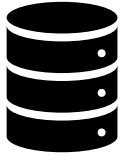
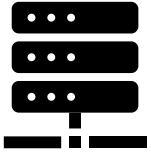
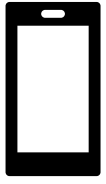
什么是调度?

协调对资源的使用请求

思考：还有哪些调度适用的场景?

- 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
- …

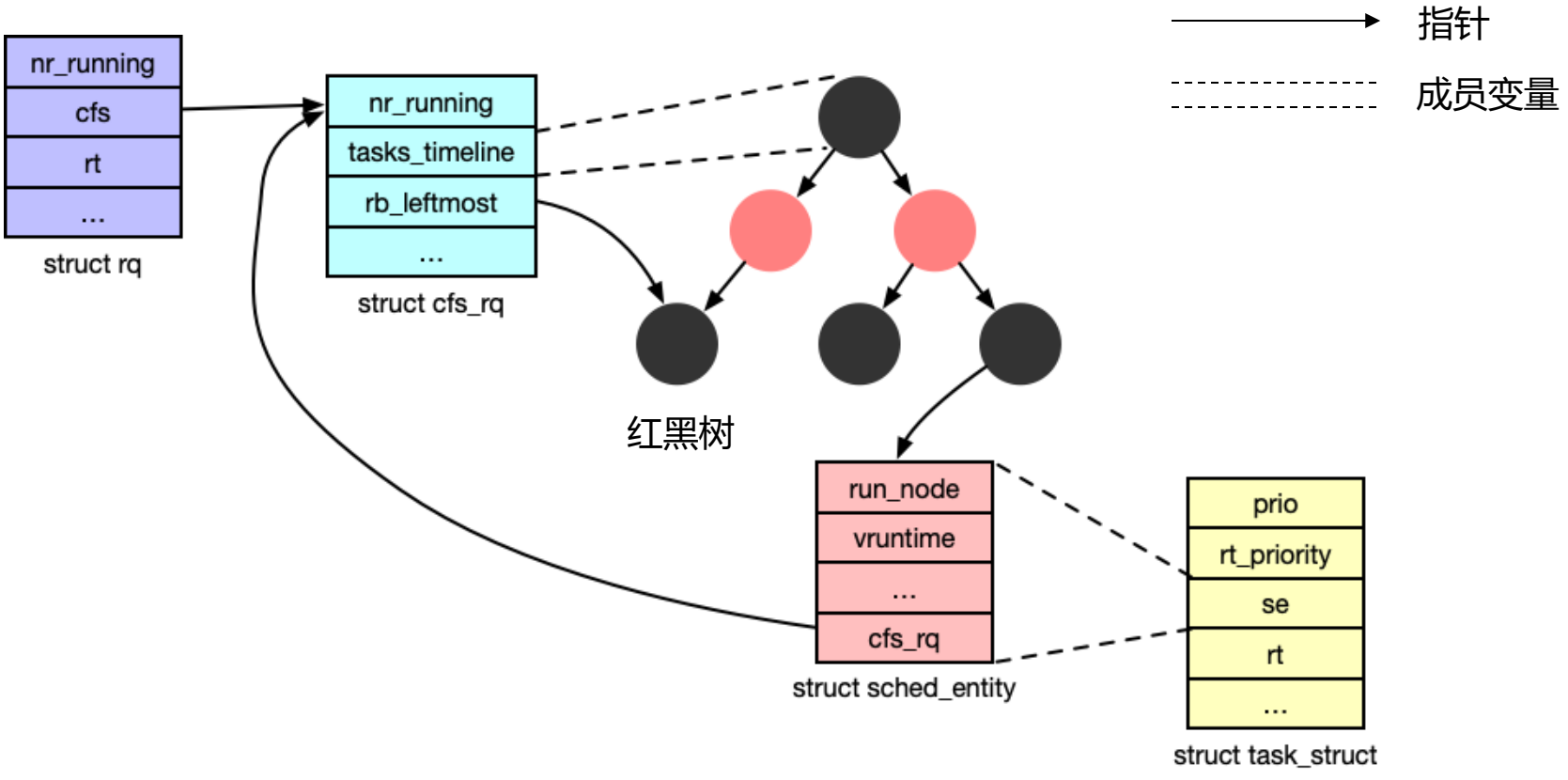
调度的挑战

- **缺少信息（没有先知）**
  - 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
  - ...

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



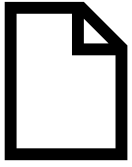
Classical Scheduling

经典调度

CPU调度与提问调度

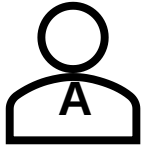
学霸

CPU



同学的问题

任务



同学

用户

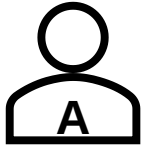
学霸调度问题

CPU调度任务

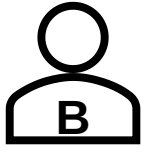
当前假设每个同学只提一个问题

First Come First Served

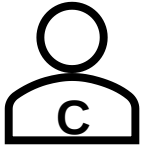
大家排队
先来后到!



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…

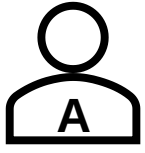
问题	到达时间	解答时间（工作量）
A	0	4
B	1	7
C	2	2

先到先得：简单、直观

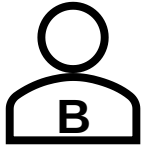
问题：平均周转、响应时间过长

Shortest Job First

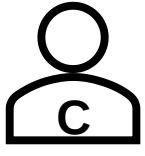
简单的问题先来



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

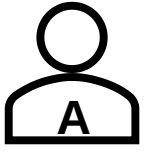
抢占式调度 (Preemptive Scheduling)

- **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

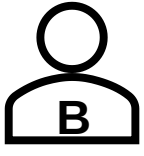
- **通过定时触发的时钟中断实现**

Round Robin (时间片轮转)

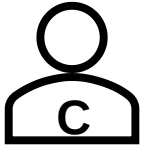
公平起见
每人轮流一分钟！



感觉多等了好久…

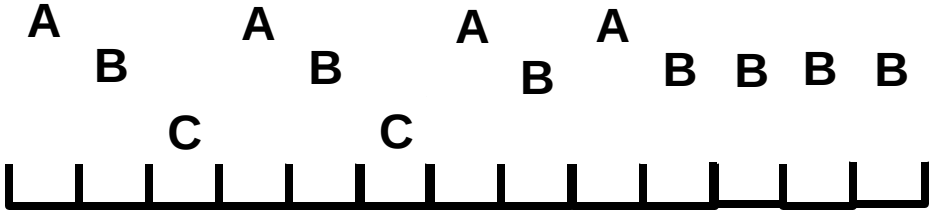


学霸的响应时间短
了好多



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？**

  每个任务的执行时间差不多相同

- **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

时钟中断与周期性工作

- **时钟中断是OS的心跳**
  - 周期性触发事件和工作的执行

- **大量的OS机制是时间驱动的**
  - 更新系统在线时间:uptime
  - 更新日历时间
  - SMP系统中，对运行队列进行负载均衡
  - 运行已到期的定时器
  - 更新资源和进程统计信息
    - 如time ls;对时间进行剖析

HZ与jiffies

- **系统中断的频率:HZ**
  - 在编译时静态确定，早期系统为100，现在为250或1000
  - 每秒产生HZ次时钟中断
- **HZ对系统的影响**
  - 越大:计时精度更高，时钟中断处理开销越大
    - 例如，进程只剩余2ms的时间片，若HZ为100,则可能10ms后才退出执行;若HZ为1000，则可精确在2ms后暂停执行。
- **jiffies:Linux中的全局变量**
  - 标识系统启动至今的tick数量(时钟中断)
  - 每秒jiffies增加HZ次
  - 系统启动至今的时间(秒数)=(jiffies-预设偏移)/HZ
    - 预设偏移offset由内核在启动时设置

```
extern unsigned long volatile jiffies;
extern u64 jiffies_64;
```

Jiffies相关的宏

用于判断时序的先后关系

```
#define time_after(unknown, known) ((long)(known) - (long)(unknown) < 0)
#define time_before(unknown, known) ((long)(unknown) - (long)(known) < 0)
#define time_after_eq(unknown, known) ((long)(unknown) - (long)(known) >= 0)
#define time_before_eq(unknown, known) ((long)(known) - (long)(unknown) >= 0)
```

用于延时，常用于设备驱动

```
void udelay(unsigned long usecs)
void ndelay(unsigned long nsecs)
void mdelay(unsigned long msecs)
```

时钟中断的处理例程

```
static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&xtime_lock);
        /* Keep track of the next tick event */
        tick_next_period = ktime_add(tick_next_period, tick_period);
        do_timer(1);                    //do_timer更新系统时间等状态
        write_sequnlock(&xtime_lock);
    }
    //update_process_times更新进程状态
    update_process_times(user_mode(get_irq_regs()));
    profile_tick(CPU_PROFILING); 更新进程状态
}
```

相关函数实现

```
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;
    update_wall_time();
    calc_global_load();
}
```

更新系统的jiffies值

更新系统全局负载统计信息

```
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id();
    /* Note: this timer irq context must be accounted for as well. */
    account_process_tick(p, user_tick);
    run_local_timers();
    rcu_check_callbacks(cpu, user_tick);
    printk_tick();
    scheduler_tick();
    run_posix_cpu_timers(p);
}
```

判断是否有到期的定时器需要运行

在scheduler_tick中判断当前进程时间片是否用完，决定是否设置调度标志

定时器应用示例：schedule_timeout

```
set_current_state(TASK_INTERRUPTIBLE);
/* take a nap and wake up in “s” seconds */
schedule_timeout(s * HZ);
```

调用schedule_timeout函数前，将当前进程状态设置为TASK_INTERRUPTIBLE

```
signed long schedule_timeout(signed long timeout)
{
    expire = timeout + jiffies;
    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long) current;
    timer.function = process_timeout;
    add_timer(&timer);
    schedule();
    …
}
```

设置定时器处理函数为process_timeout，之后进入睡眠

```
void process_timeout(unsigned long data)
{
    wake_up_process((task_t *) data);
}
```

唤醒睡眠进程

内核代码分析

- **简单的Monte Carlo算法**
  - 每次时钟中断发生时，判断当前状态在用户态还是内核态
  - 如果在内核态，根据程序PC，判断当前的函数位置
    - 定于热点区域，进行算法优化
  - Time命令，top命令
    - 用户态时间和核心态时间由时钟中断的统计函数完成

- **强大的内核分析工具：perf、Oprofile**
  - 基于PMU
  - 采用类似的定时或定量采样机制
  - 粒度更细，可观测时间更多
  - 如perf top可以直接观察内核和应用的执行热点区域

时钟相关的系统调用

System call	Description
clock_gettime()	Gets the current value of a POSIX clock
clock_settime( )	Sets the current value of a POSIX clock
clock_getres( )	Gets the resolution of a POSIX clock
timer_create( )	Creates a new POSIX timer based on a specified POSIX clock
timer_gettime( )	Gets the current value and increment of a POSIX timer
timer_settime( )	Sets the current value and increment of a POSIX timer
timer_getoverrun( )	Gets the number of overruns of a decayed POSIX timer
timer_delete( )	Destroys a POSIX timer
clock_nanosleep()	Puts the process to sleep using a POSIX clock as time source

Priority Scheduling

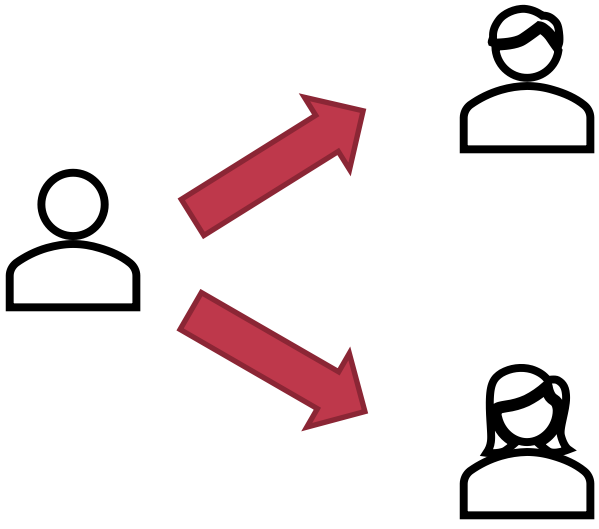
优先级调度

调度优先级

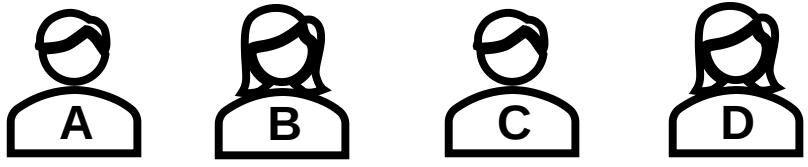
- **操作系统中的任务是不同的，例如：**
  - 系统 V.S. 用户、前台 V.S. 后台、…
- **如果不加以区分**
  - 系统关键任务无法及时处理
  - "后台运算"导致"视频播放"卡顿

- **优先级用于确保重要的任务被优先调度**

添加条件：优先级



女生优先级高！



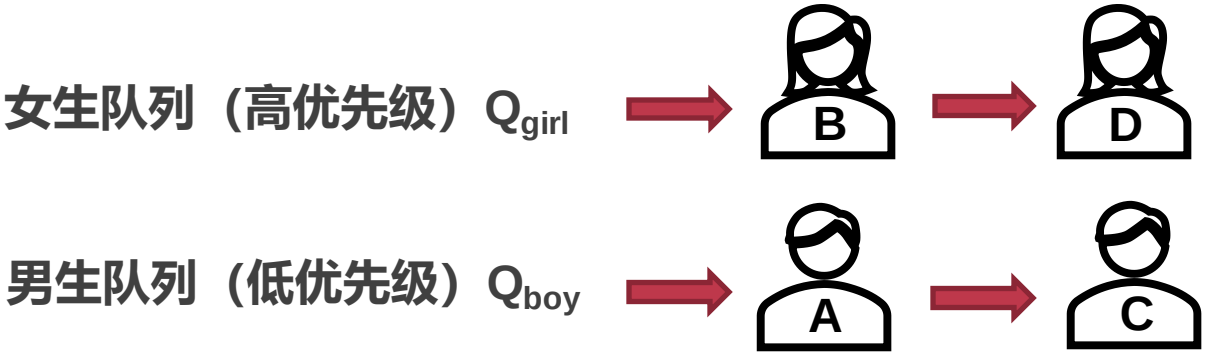
多级队列

Multi-Level Queue (MLQ)

1）维护多个队列，每个对应静态设置好的优先级

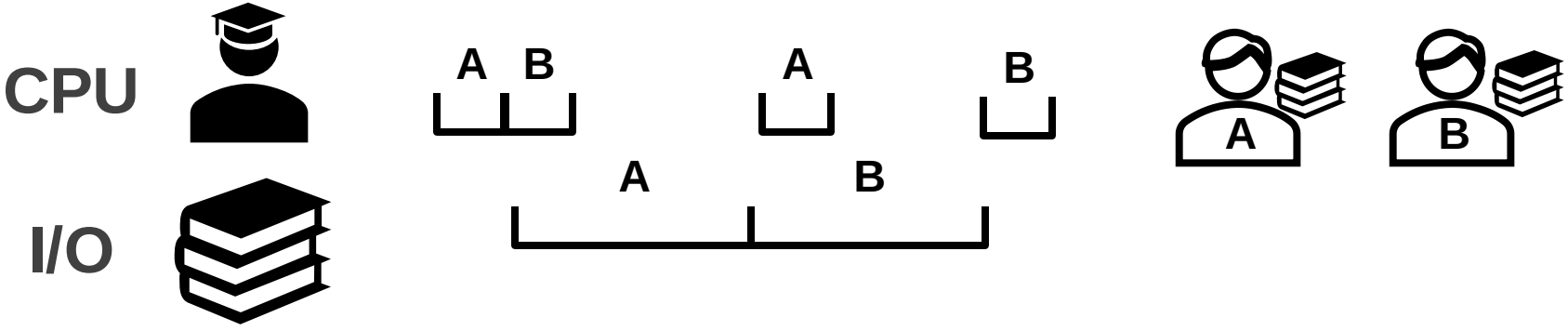
2）高优先级的任务优先执行

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**

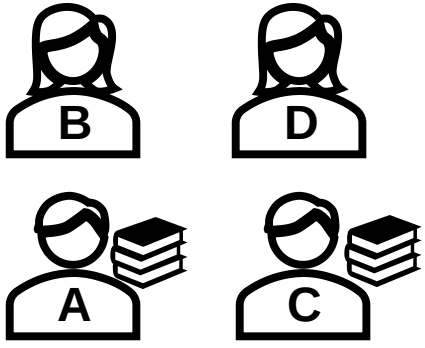


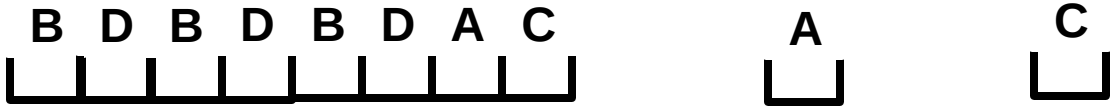
问题1：低资源利用率

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

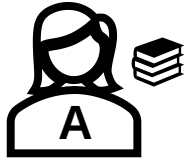
问题2：优先级反转

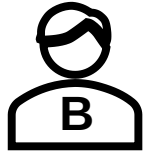
- **高、低优先级任务都需要独占共享资源**
  - 共享资源
    - 存储
    - 硬件
    - OS书
    - …
  - 通常使用信号量、互斥锁实现独占
- **低优先任务占用资源 -> 高优先级任务被阻塞**

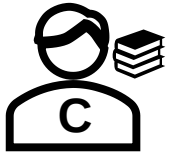
问题2：优先级反转

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

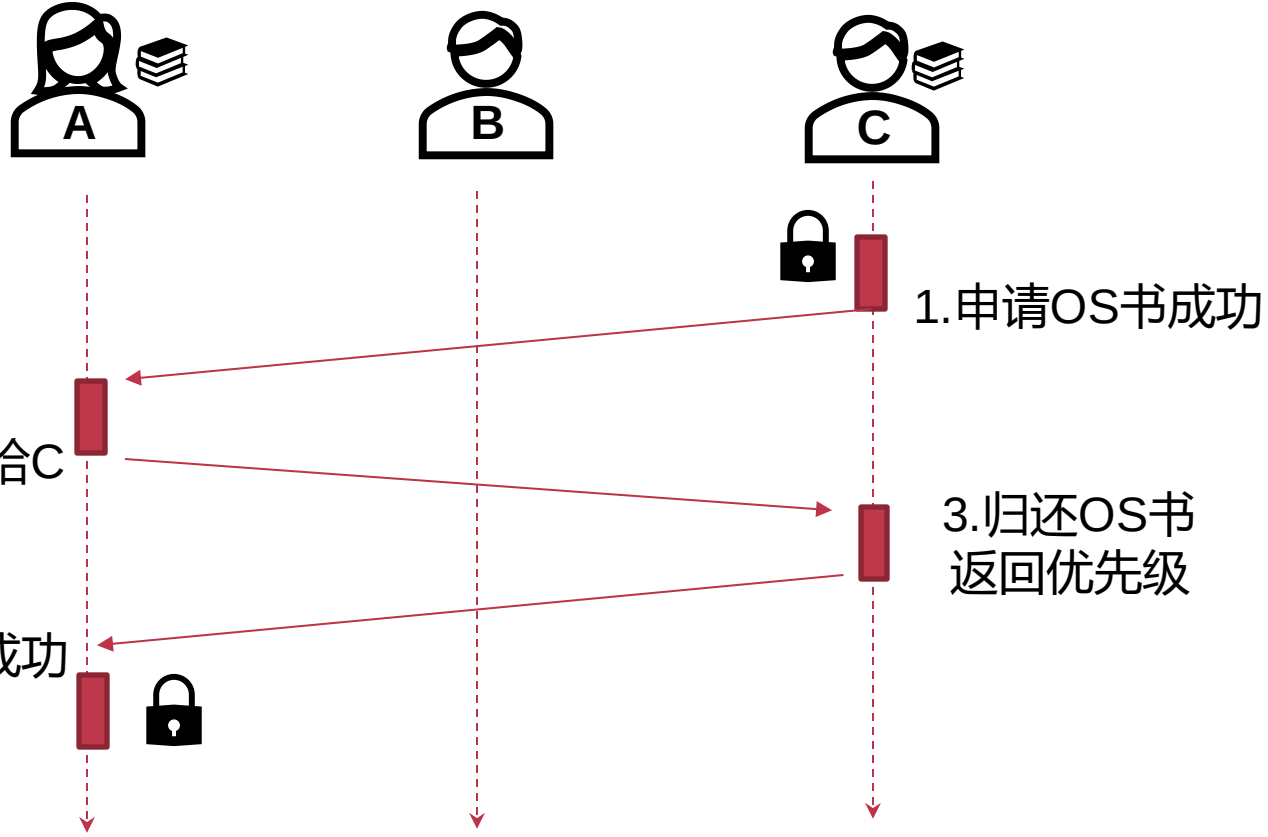
2. 抢占C
申请OS书失败
等待

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



思考：优先级

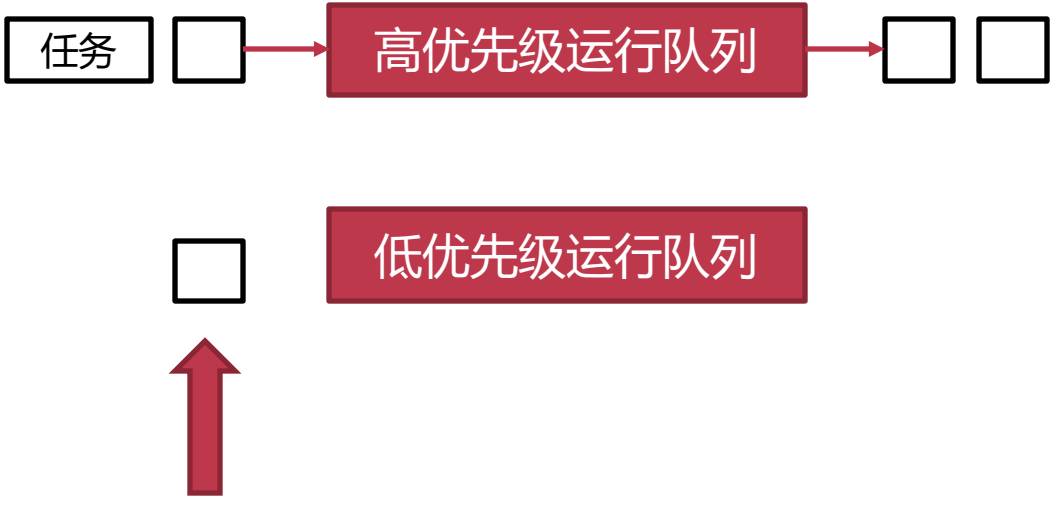
- **以下这些调度策略算不算优先级调度?**
  - First Come First Served
  - Shortest Job First
  - Round Robin

MLFQ

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）
- **假设调度没有开销**
  - RR（将时间片设置过短会导致调度开销过大）

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

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
  - …

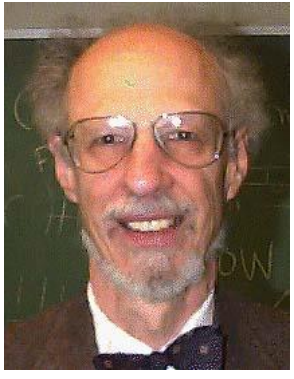
- **某些场景下，任务的优先级需要动态调整**

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？

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以及其它概念
  - 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：如果工作场景变化频繁，效果会很差

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?

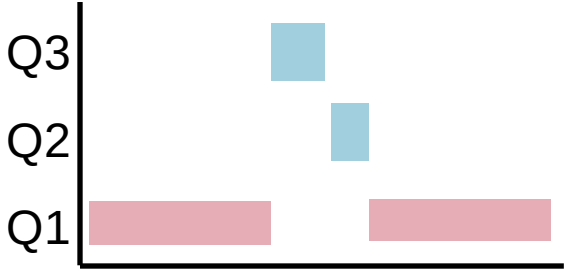
- **针对混合工作场景**
  - 执行时间短的任务
    - 交互式任务
    - I/O密集型任务
  - 执行时间长的任务
    - CPU密集型计算任务

- **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前放弃CPU，那么它的优先级不变
  - 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

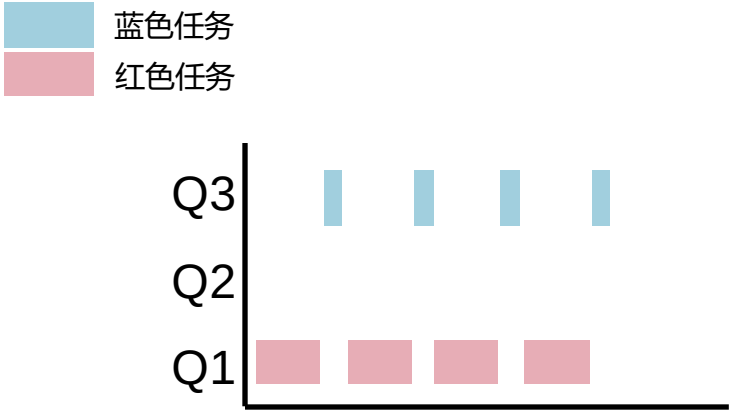


*Q3优先级最高

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



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，…

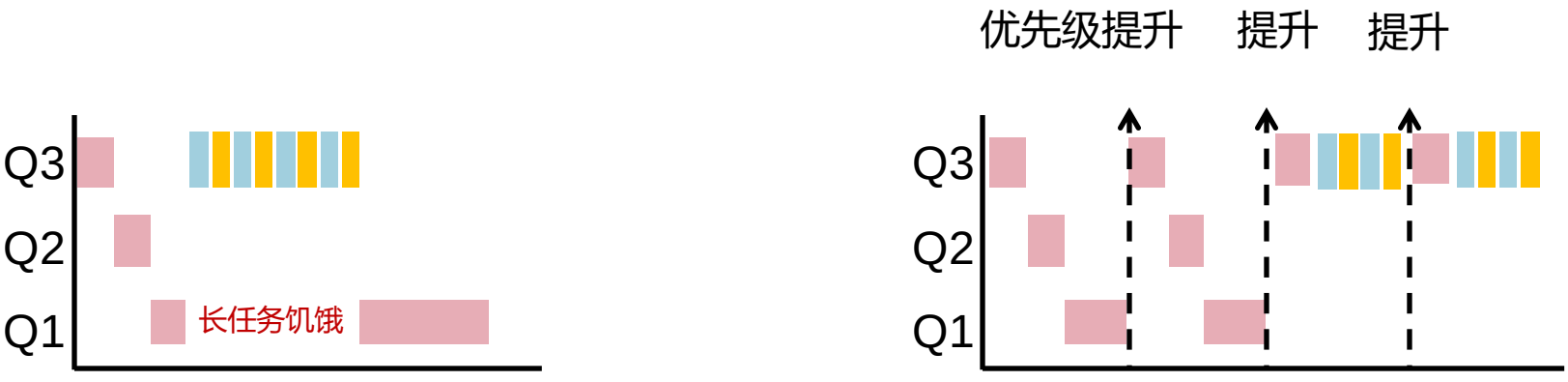
定时优先级提升

- **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

思考：为什么要提升全部的优先级？

- **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

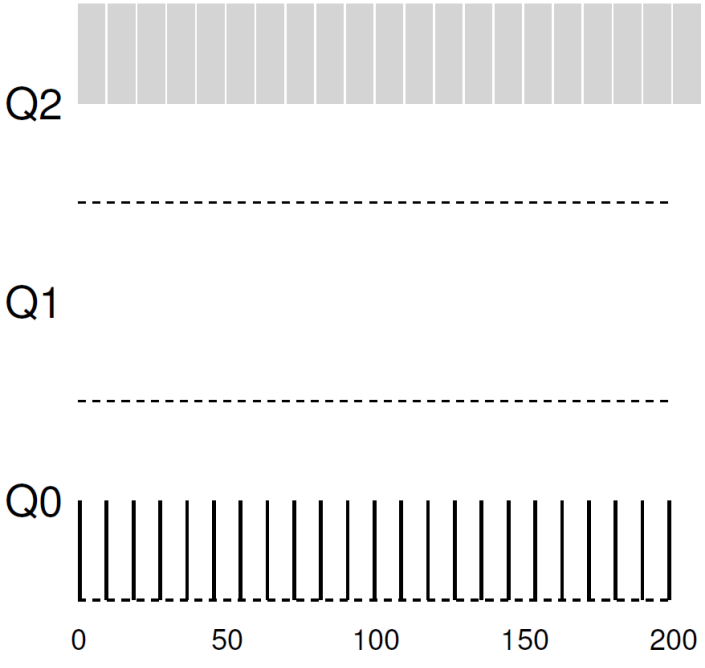


4. 采用定时优先级提升的前后对比（左为采用前，右为采用后）

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）

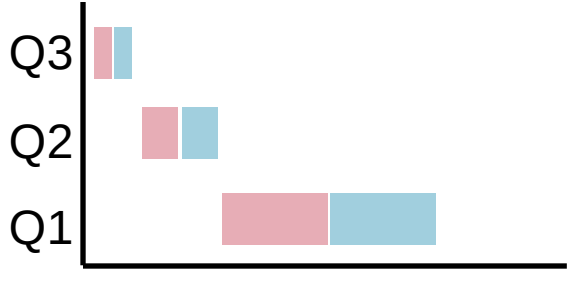
MLFQ的参数调试

- **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
    - 提升响应时间
  - 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

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？

高响应比优先（Highest Response Ratio Next）

响应比（Response Ratio）是一个任务的响应时间（$T_{Response}$）与其运行时间（T_{Run}）的比值

- 优先级 $= \text{Response Ratio} = \frac{T_{Response}}{T_{Run}} = \frac{T_{Waiting}+T_{\text{Run}}}{T_{Run}} = \frac{T_{Waiting}}{T_{Run}} + 1$
- **选择就绪队列中响应比R值最高的进程**
- 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

HRRN 策略通过结合FCFS策略和SJF策略，避免了SJF策略在公平性方面的问题。

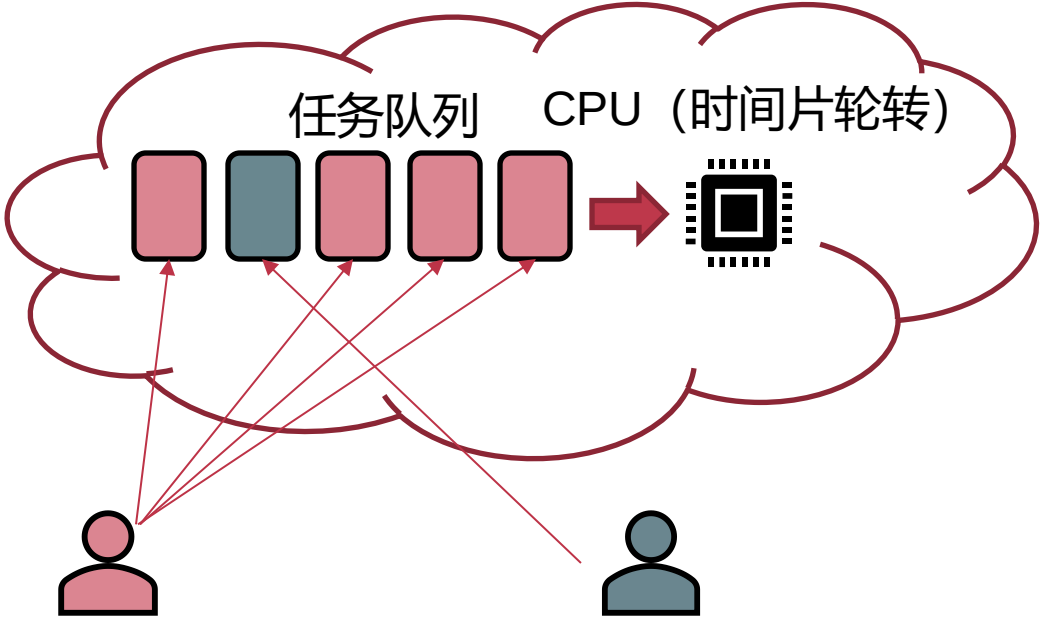
- ✓ 关注进程的等待时间
- ✓ 防止无限期等待

Fair-Share Scheduling

公平共享调度

场景：共享服务器

- **A和B两位同学合租一台服务器（5天）**
- **两人希望按比例分配CPU时间**
  - 按时间分担租赁费用
- **考虑任务响应时间**
  - 不允许FCFS、SJF

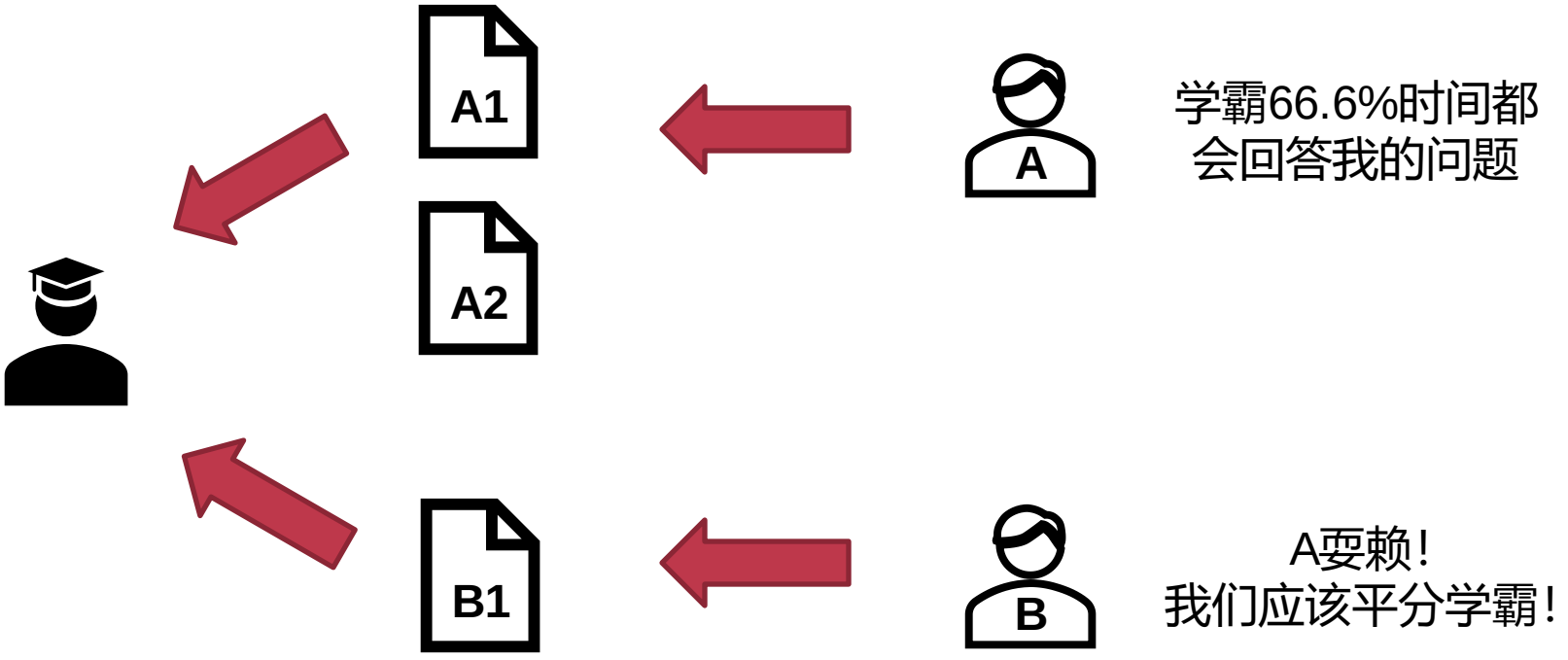


公平共享

- **每个用户占用的资源是成比例的**
  - 而非被任务的数量决定

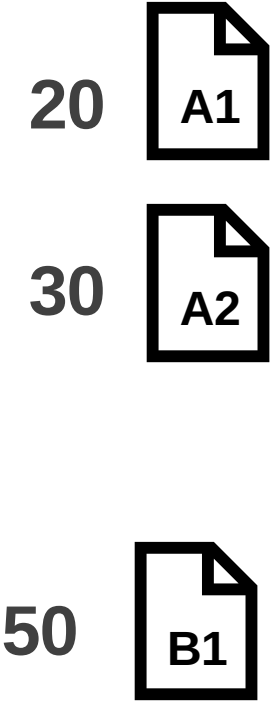
- **每个用户占用的资源是可以被计算的**
  - 设定“份额"以确定相对比例（绝对值不重要）
  - 例：份额为4的用户使用资源，是份额为2的用户的2倍

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

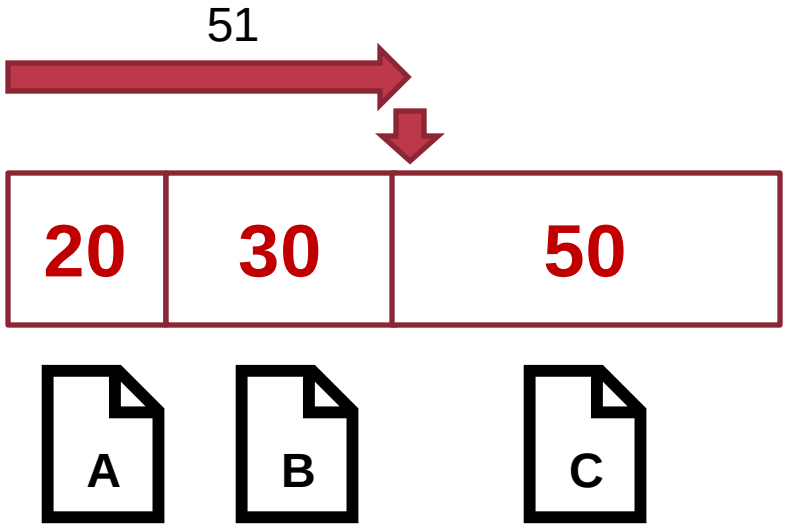


方法：使用“ticket”表示任务的份额

- $ticket$：每个任务对应的份额
- T：ticket的总量
- 任务A：ticket 20
- 任务B：ticket 30
- 任务C：ticket 50
- 则A:B:C占用的CPU执行时间
  - 20:30:50

例：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 → 调度C

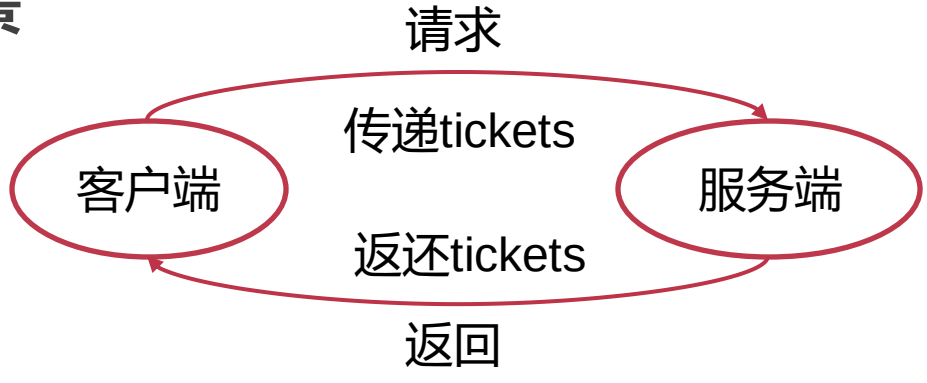



```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task.ticket
    if (R < sum) {
        break
    }
}
schedule()
```


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

彩票转让（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转让给服务端**
  - 确保服务端可以尽可能使用更多资源，迅速处理
- **同样适用于其他需要同步的场景**



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

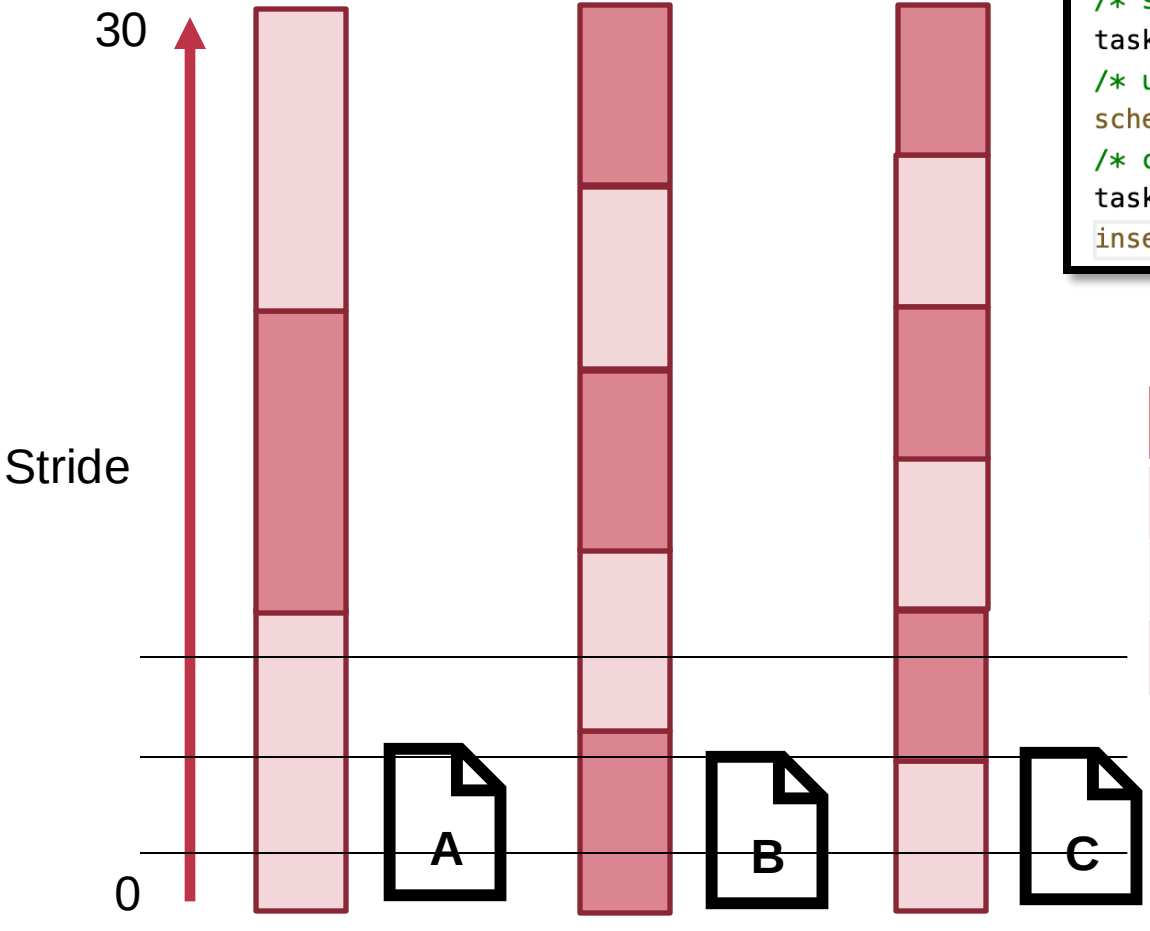
步幅调度（Stride Scheduling）

- **可以看做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**
  - $stride = \frac{MaxStride}{ticket}$
    - 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）




```
/* 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

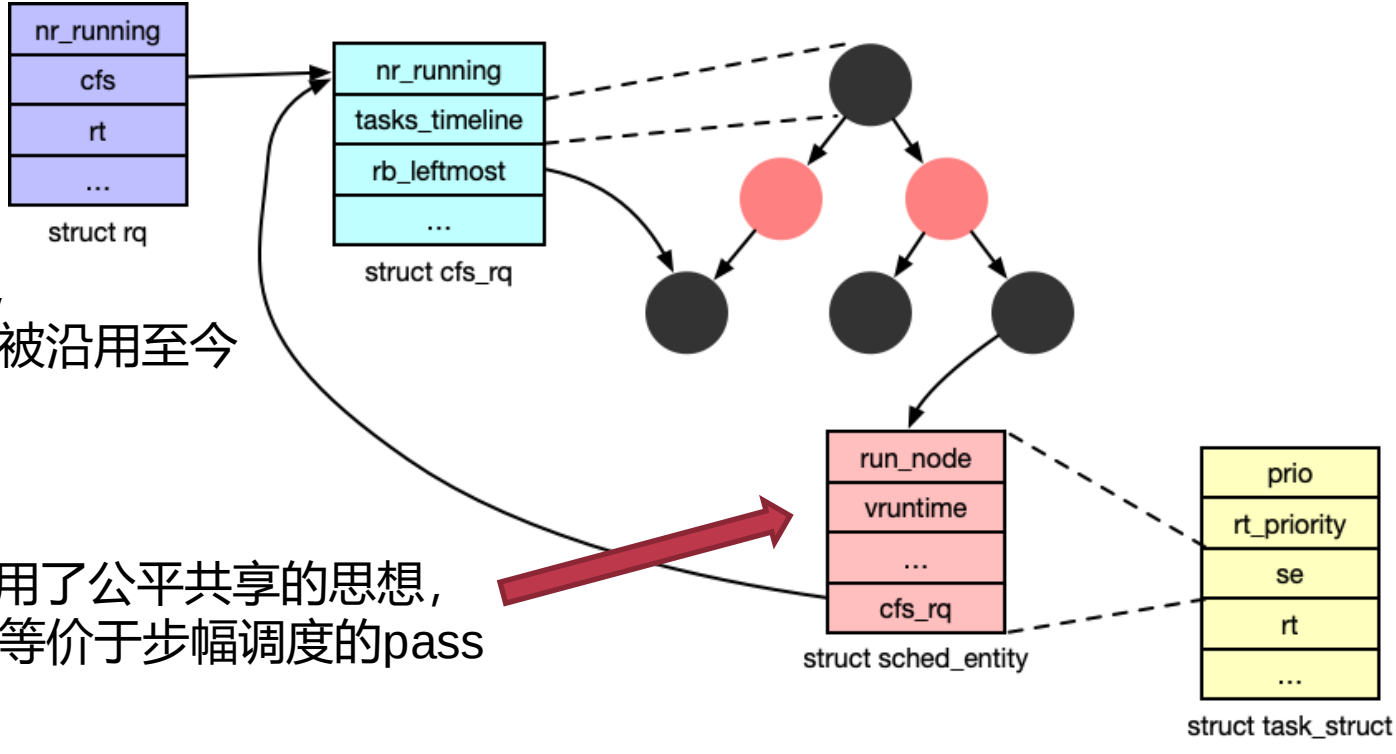
	Lottery Scheduling	Stride Scheduling
调度决策生成	随机	确定性计算
任务实际执行时间与预期的差距	大	小

预期——根据任务份额ticket计算的执行时间期望

Linux完全公平调度器（CFS，Complete Fair Scheduler）

CFS于2007年10月，
整合进Linux主线并被沿用至今

CFS同样采用了公平共享的思想，
其vruntime 等价于步幅调度的pass



Linux完全公平调度器（CFS，Complete Fair Scheduler）

- **CFS达成公平共享的方式**
  - 通过调整任务每次执行的时间

	达成公平共享的方式	调度次数之比	每次调度执行时间之比
步幅调度	控制调度次数	2：3	1：1
CFS	控制执行时间	1：1	2：3

例：步幅调度和CFS如何调度份额(ticket)之比是2：3的两个任务

思考：虚拟时间的更新

- 假设份额为ticket的任务某次执行了T单位时间，应该如何更新其虚拟时间?

已知，步幅调度更新虚拟时间的公式：

$$pass \mathrel{+}= \frac{T}{ticket}$$

问，CFS更新虚拟时间的公式是?

$$vruntime \mathrel{+}= \frac{T}{ticket}$$

	达成公平共享的方式	调度次数之比	每次调度执行时间之比
步幅调度	控制调度次数	2：3	1：1
CFS	控制执行时间	1：1	2：3

例：步幅调度和CFS如何调度份额(ticket)之比是2：3的两个任务

实际CFS的vruntime计算更加复杂，具体可参考： https://www.linuxjournal.com/node/10267

虚拟时间对调度行为的影响

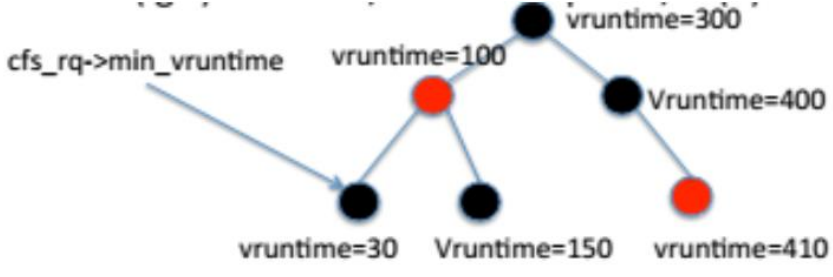
- **虚拟时间的意义**
  - 虚拟时间相对大小，对应了任务的优先级
  - 虚拟时间的绝对值，对应了任务预期占用CPU的时间长度
- **如果新创建、刚被唤醒的任务vruntime很小，可能遇到的问题是?**
  - 这些新创建、刚被唤醒的任务会立即长时间占用CPU
- **CFS当前的解决方案**
  - 维护所有任务虚拟时间的最小值min_vruntime
  - 当任务被创建、唤醒时，保证任务的vruntime不小于min_vruntime

CFS 进程饿死问题

- 采用CFS调度器，可能会出现进程饿死现象吗?

vruntime与进程管理

- **通过红黑树对进程进行管理**
  - 平衡二叉搜索树(Balanced binary search tree)
  - 以vruntime作为键值进行排序
  - O(lgN):插入、删除、更新
  - O(1):查找最小之值
  - min_vruntime 缓存最小vruntime
- **更新vruntime和min_vruntime的时间**
  - 每次新任务加入或删除时
  - 每个时钟周期，每次进程切换



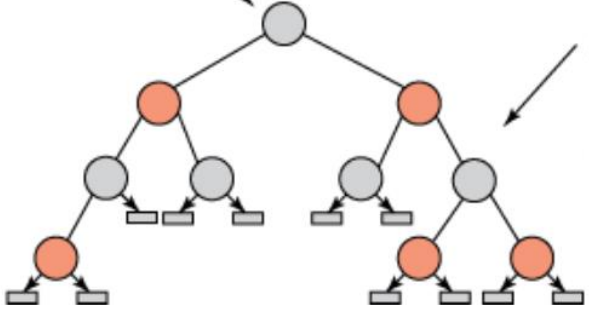
vruntime与进程管理结构

```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

不再需要专门的优先级队列，直接通过RB树对进程进行管理

```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
  ...
  struct rb_root tasks_timeline;
  ...
};
```



```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```

低runtime抢占当前进程代码

```
/*
 * wake_up_new_task - wake up a newly created task for the first
time.
*/
void wake_up_new_task(struct task_struct *p)
{
        ...
        rq = __task_rq_lock(p);
        activate_task(rq, p, 0);
        p->on_rq = 1;
        trace_sched_wakeup_new(p, true);
        check_preempt_curr(rq, p, WF_FORK);
        ...

        task_rq_unlock(rq, p, &flags);
}
```

新进程被唤醒时，调用check_preempt_curr函数检查是否可以抢占当前进程

check_preempt_curr函数

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void check_preempt_wakeup(struct rq *rq, struct task_struct
*p, int wake_flags)
{
        ...
        if (wakeup_preempt_entity(se, pse) == 1) {
                /*
                 * Bias pick_next to pick the sched entity that is
                 * triggering this preemption.
                 */
                if (!next_buddy_marked)
                        set_next_buddy(pse);
                goto preempt;
        }
        return;
preempt:
        resched_task(curr);
```

若可抢占，则重新调度

wakeup_preempt_entity函数

```
/*
 * Should 'se' preempt 'curr'.
 */
static int
wakeup_preempt_entity(struct sched_entity *curr, struct
sched_entity *se)
{
        s64 gran, vdiff = curr->vruntime - se->vruntime;

        if (vdiff <= 0)
                return -1;

        gran = wakeup_gran(curr, se);
        if (vdiff > gran)
                return 1;  //当前运行进程的vruntime大，可以抢占

        return 0;
}
```

CFS小结

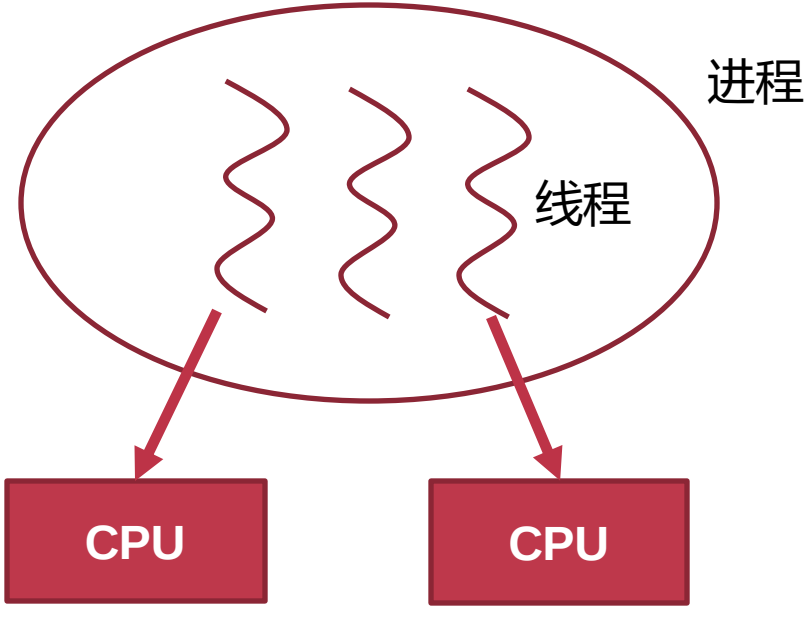
- **公平性**
  - 系统负载极重时，也能保证确定的响应时间和运行时间
  - 低优先级任务也能得到关注和执行
- **适应性**
  - 自动识别I/O密集和CPU密集
    - I/O进程被唤醒时自动优先得到调度
- **实现性**
  - CFS设计简洁，性能够用Olog(N)

Multicore Scheduling Policy

多核调度策略

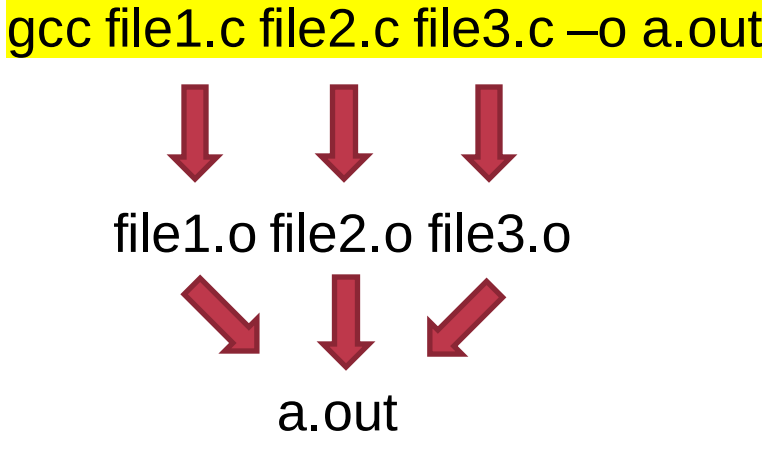
多核调度需要考虑的额外因素

- 一个进程的不同线程可以在不同CPU上同时运行



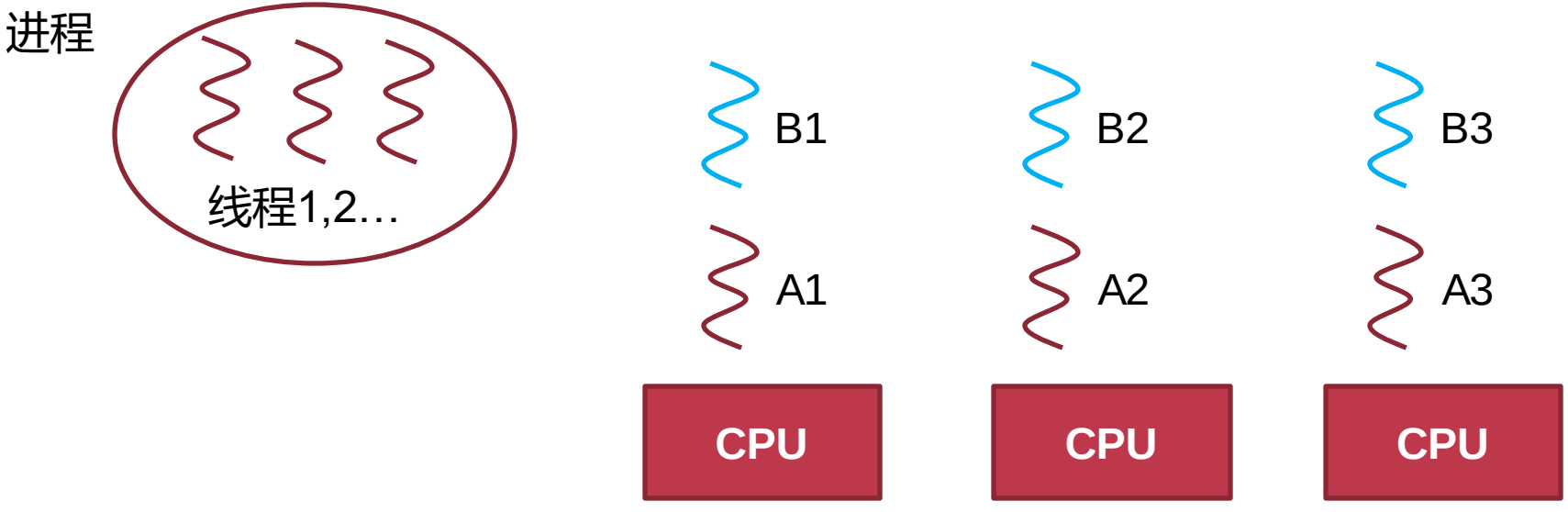
同一个进程的线程很可能有依赖关系

- **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



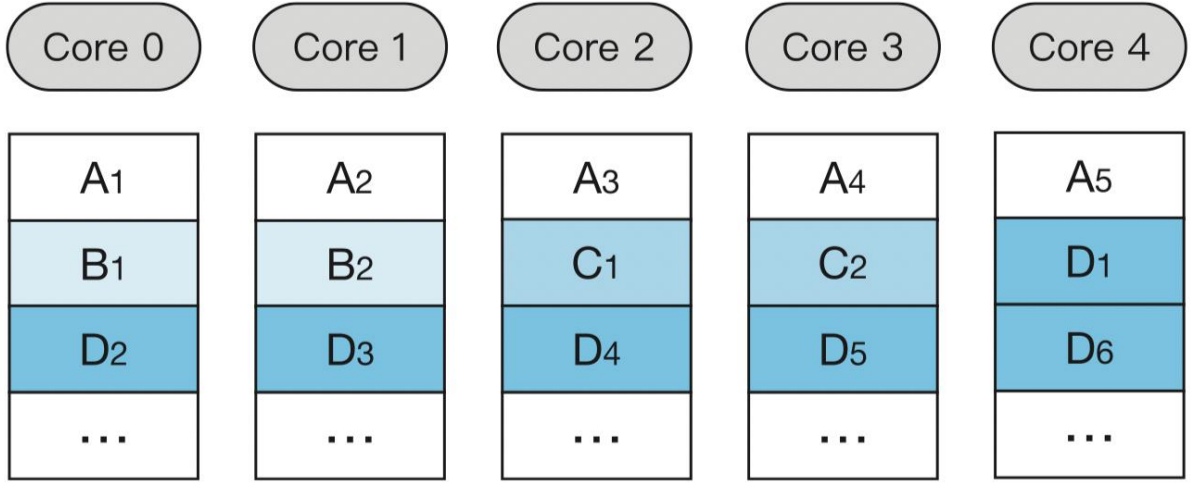
群组调度：Gang Scheduling

- 在多个CPU上同时执行一个进程的多个线程

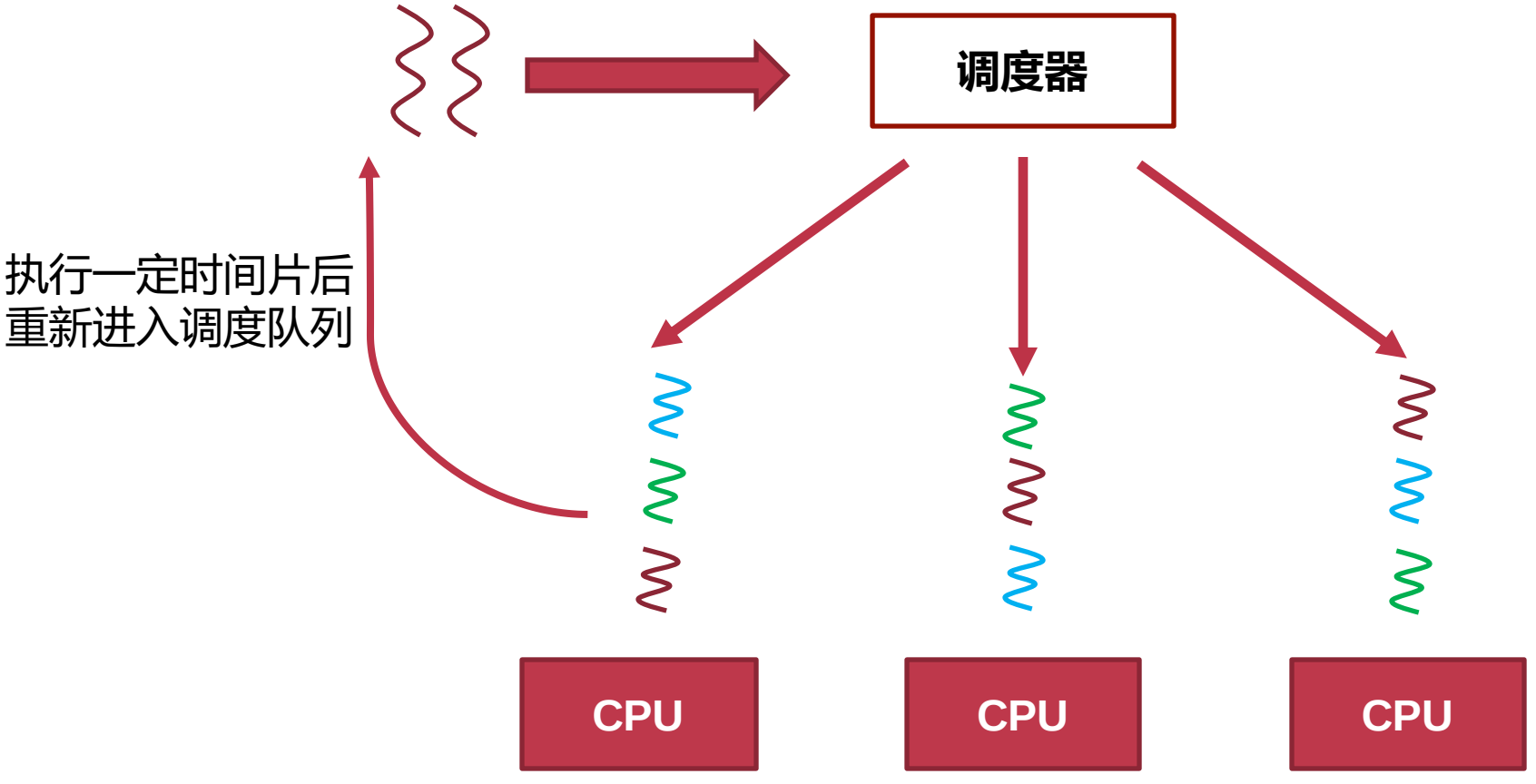


群组调度：Gang Scheduling

- **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

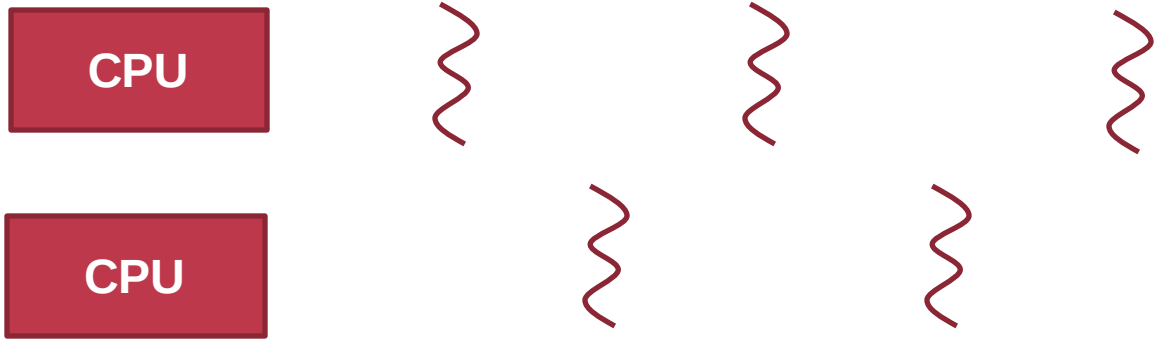


全局使用一个调度器的问题

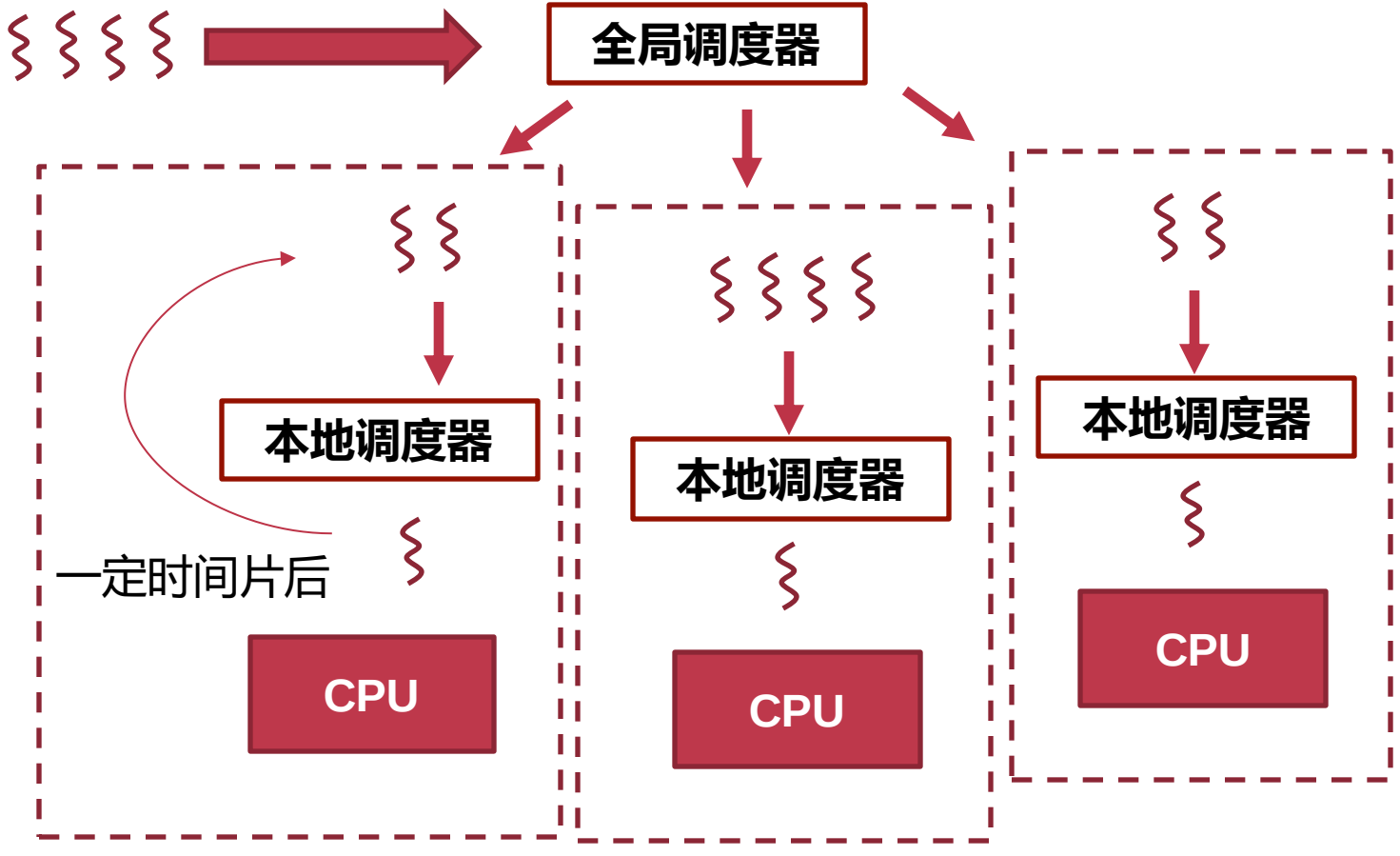


全局使用一个调度器的问题

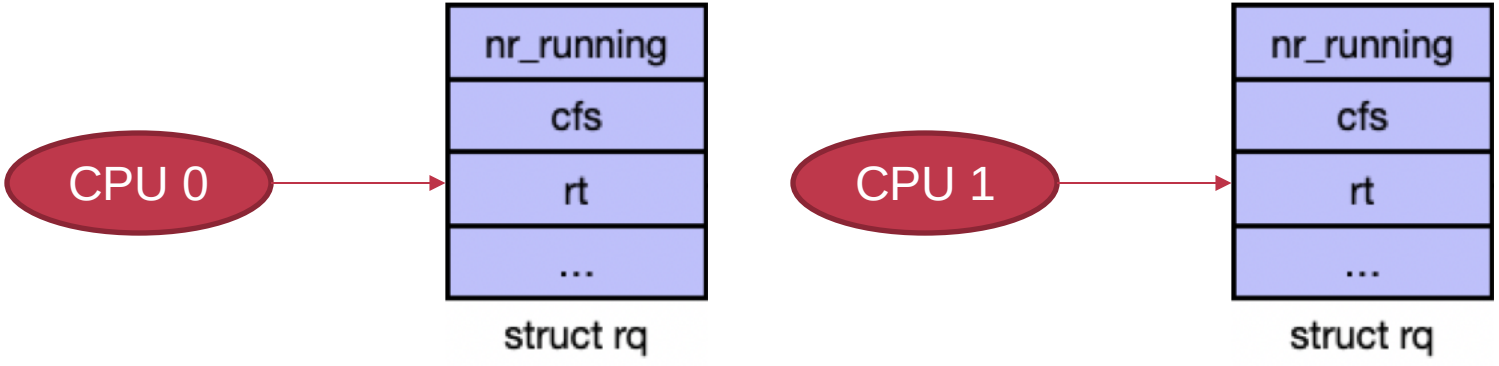
- **所有CPU竞争全局调度器**
- **同一个线程可能在不同CPU上切换**
  - 切换开销大：Cache、TLB、…
  - 缓存局部性差



Two-level Scheduling



Linux的多核调度

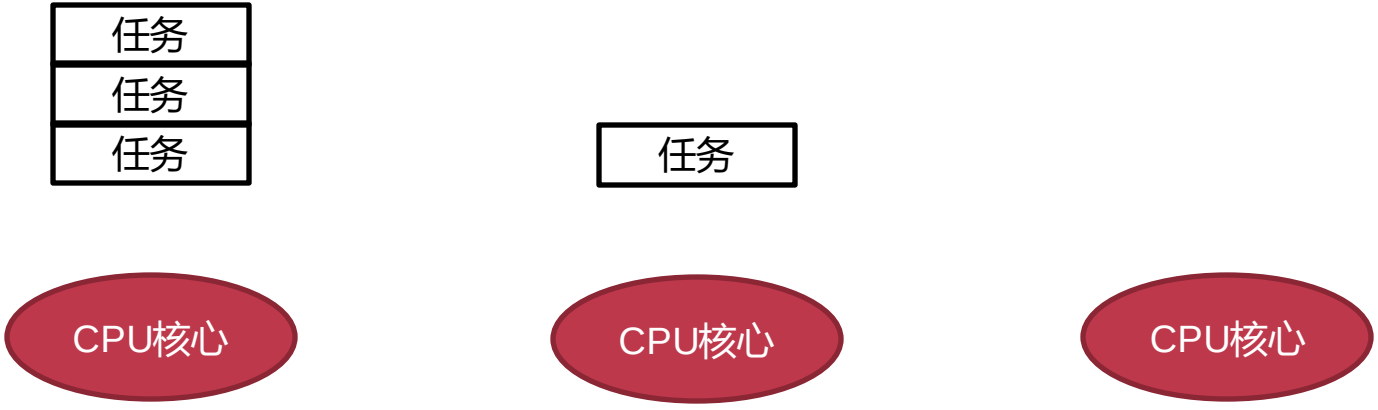


Linux使用Two-level Scheduling的架构

每个CPU有各自的本地调度器和runq

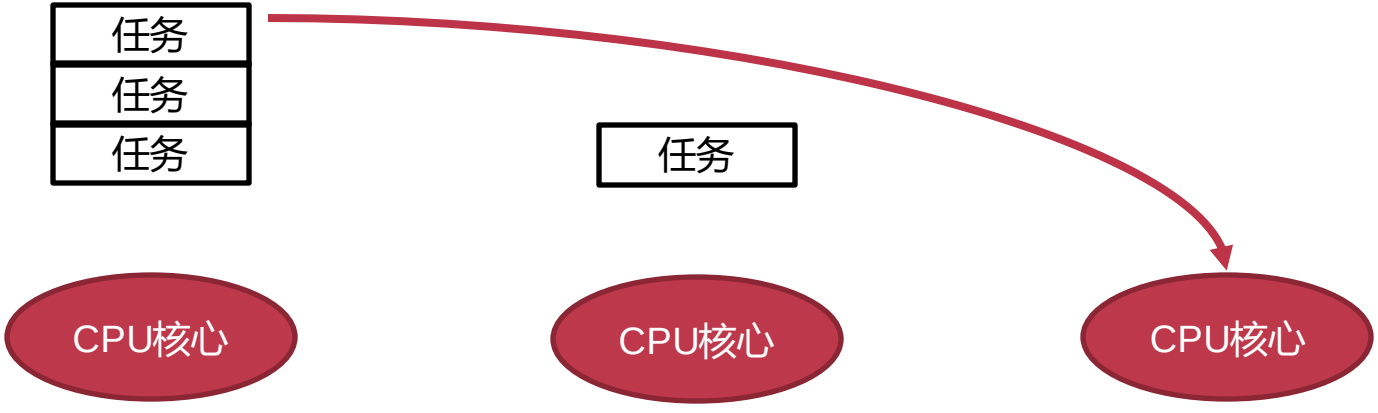
思考：指定线程在CPU上执行的问题

- 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



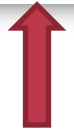
思考：如何定义任务的负载?

- **根据任务负载定义的不同，负载均衡的效果也不尽相同**
- **请探讨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
  - 每个任务单位时间内使用的CPU资源

亲和性（Affinity）

- **程序员如何控制自己程序的行为？**
  - 例如，程序员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

亲和性（Affinity）

- **命令行工具如何控制自己程序的行为?**
  - taskset
- **内核线程亲和性接口**
  - Kthread_bind

调度小结

- 调度指标
- 经典调度
- 优先级调度
- 公平共享调度
- 多核调度

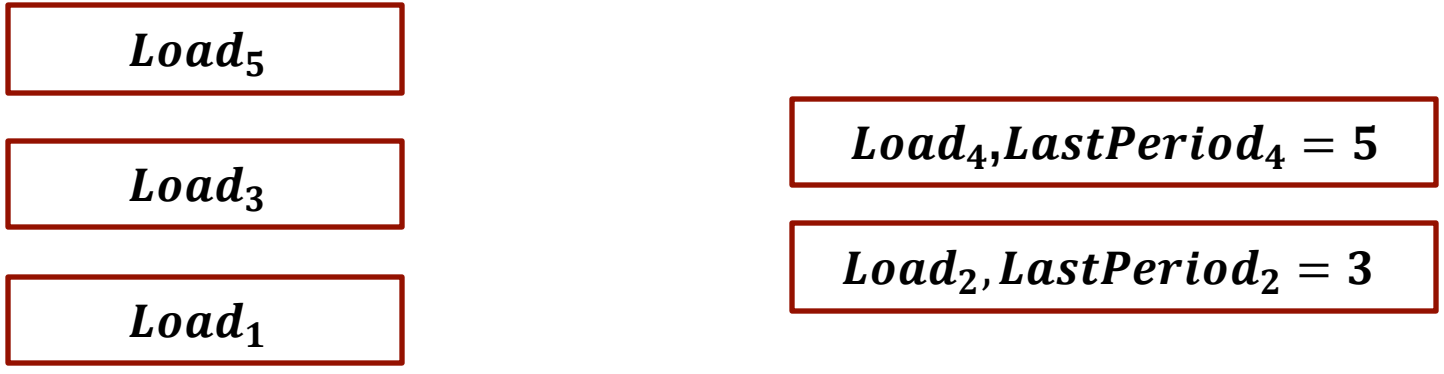
思考：被阻塞任务对负载的影响

- 一个任务等待I/O时会被阻塞

- 计算CPU总负载的时候是否考虑被阻塞的CPU?

PELT（per-entity load tracking）对阻塞任务的处理

$Period = 7$：当前周期



运行队列负载
$Load_{run}$

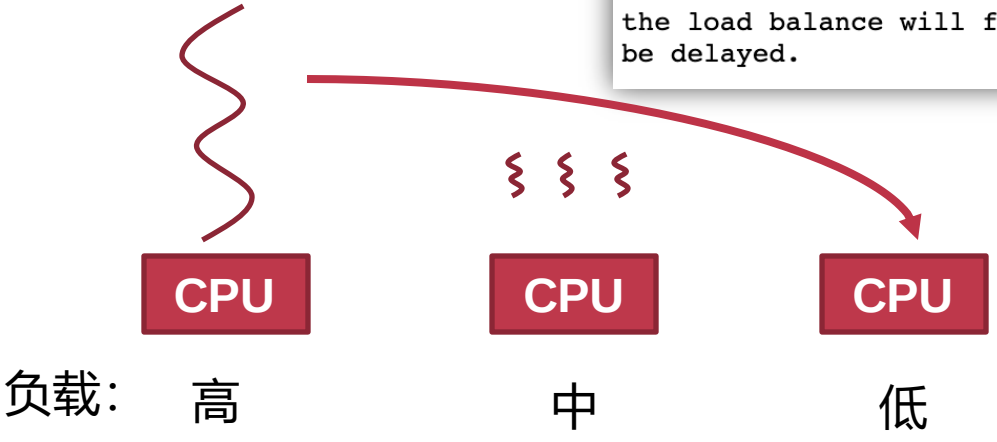
阻塞队列负载
$Load_{block}$

$$Load_{CPU} = Load_{run} + Load_{block}$$

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;
```

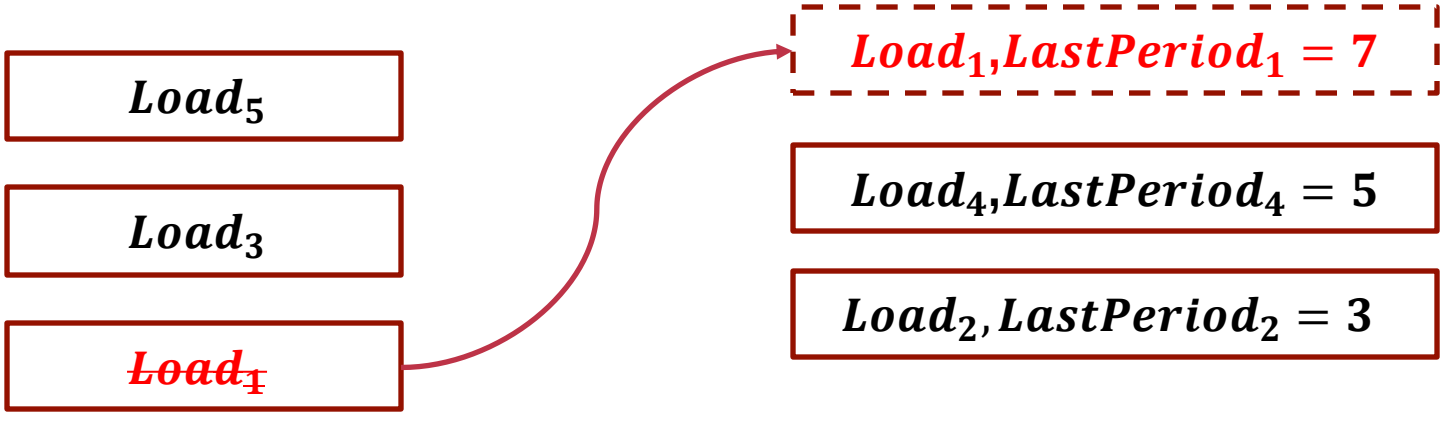
Below are the average results for 15 iterations on an arm64 octo core:
sysbench --test=cpu --num-threads=8 --max-requests=1000 run

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

例：PELT对阻塞任务的处理

$Period = 7$：当前周期

操作1：线程1进入阻塞队列



运行队列负载

$Load_{run} -= Load_1$

阻塞队列负载

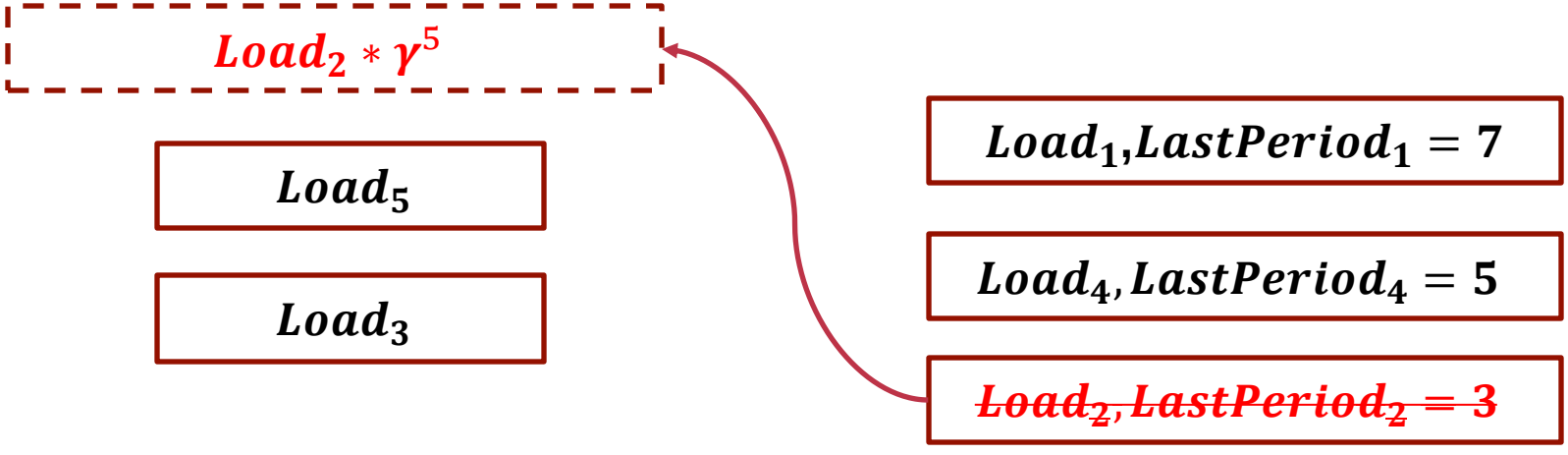
$Load_{block} += Load_1$

$$Load_{CPU} = Load_{run} + Load_{block}$$

例：PELT对阻塞任务的处理

$Period = 8$：当前周期

操作1：线程2返回运行队列



运行队列负载

$Load_{run} \mathrel{+}= Load_2 * \gamma^5$

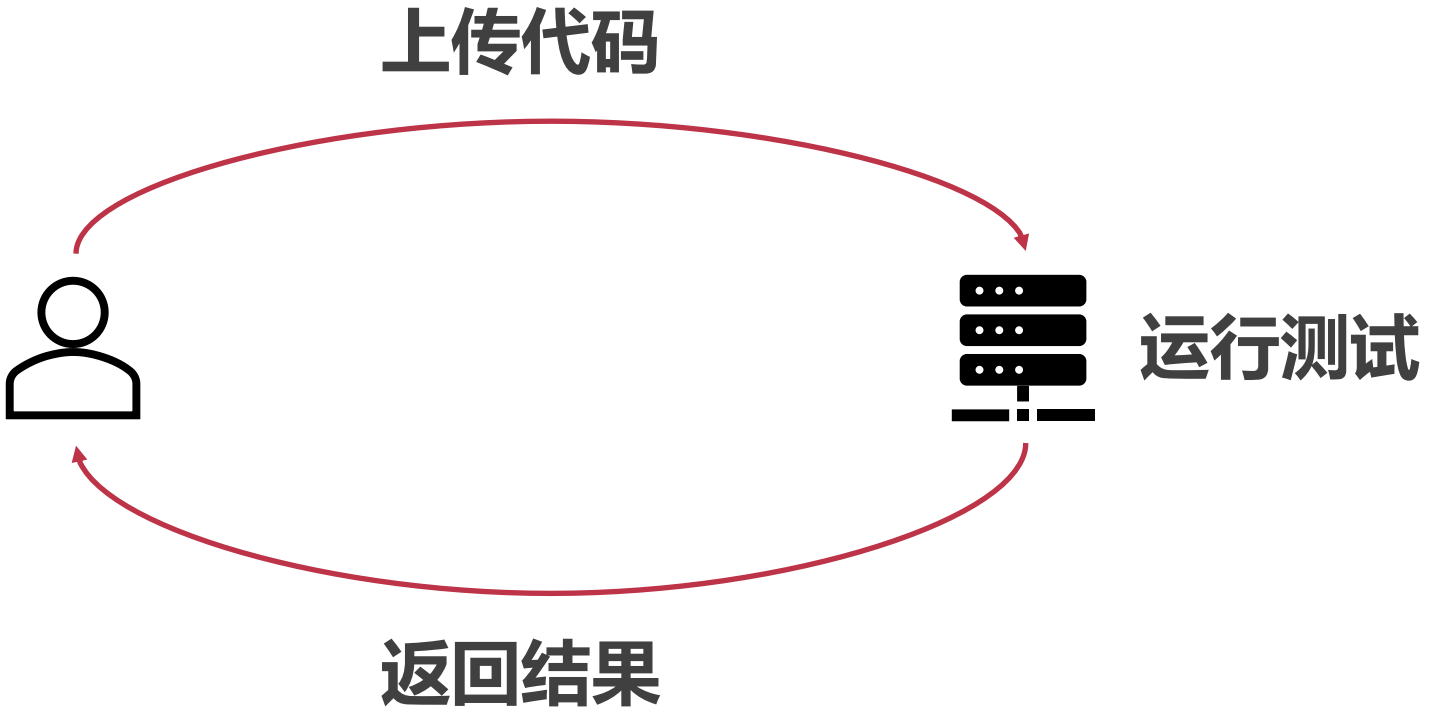
阻塞队列负载

$Load_{block} \mathrel{-}= Load_2 * \gamma^{Period - LastPeriod_2}$

$\mathrel{-}= Load_2 * \gamma^5$

$$Load_{CPU} = Load_{run} + Load_{block}$$

样例分析：在线编程网站



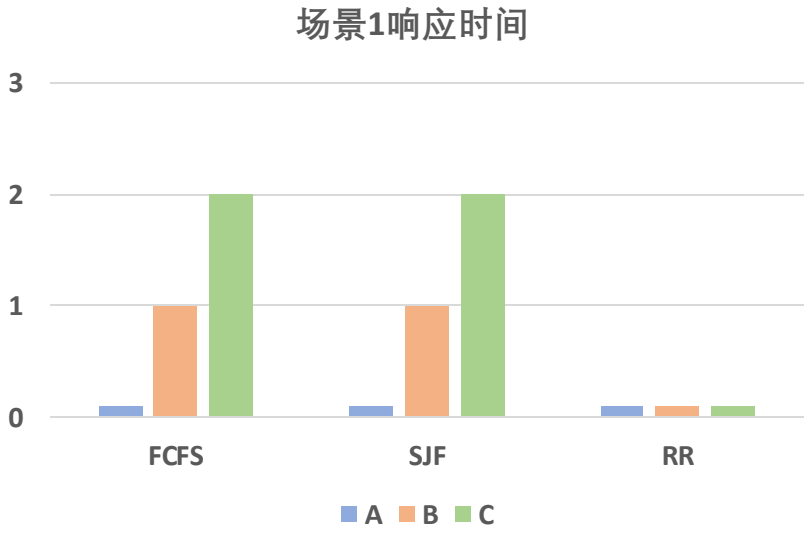
场景1：单一测试

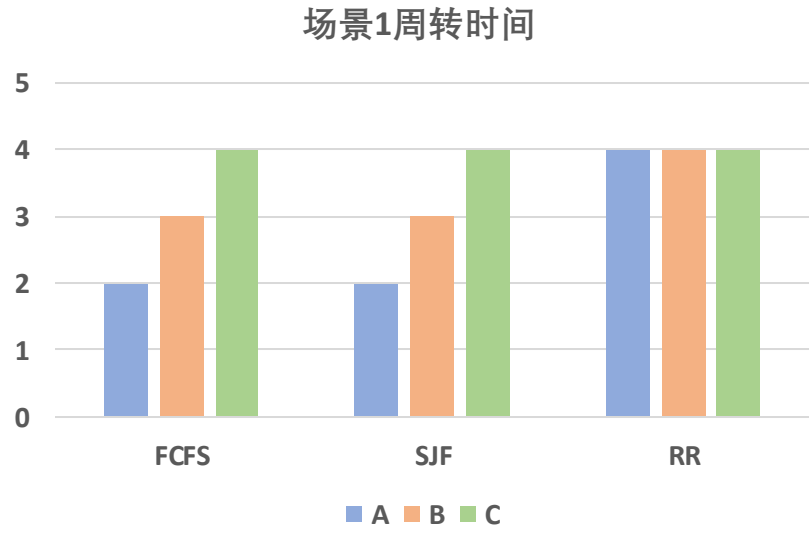
FCFS	到达时刻	工作时间
A	0	2
B	1	2
C	2	2

不同任务的工作时间相同

Time		0	1	2	3	4	5	6
FCFS	CPU							
SJF	CPU							
RR	CPU							

场景1：单一测试





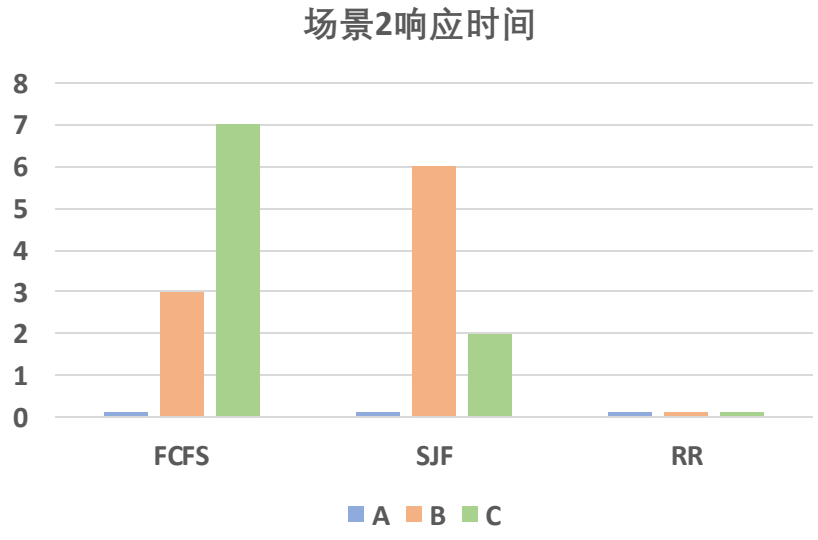
场景2：多样化测试

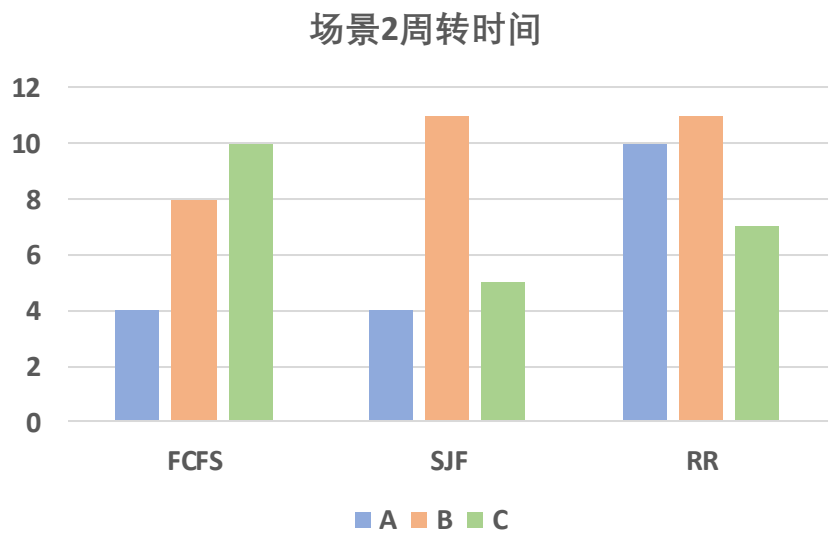
FCFS	到达时刻	工作时间
A	0	4
B	1	5
C	2	3

任务的工作时间呈现差异化

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
FCFS	CPU													
SJF	CPU													
RR	CPU													

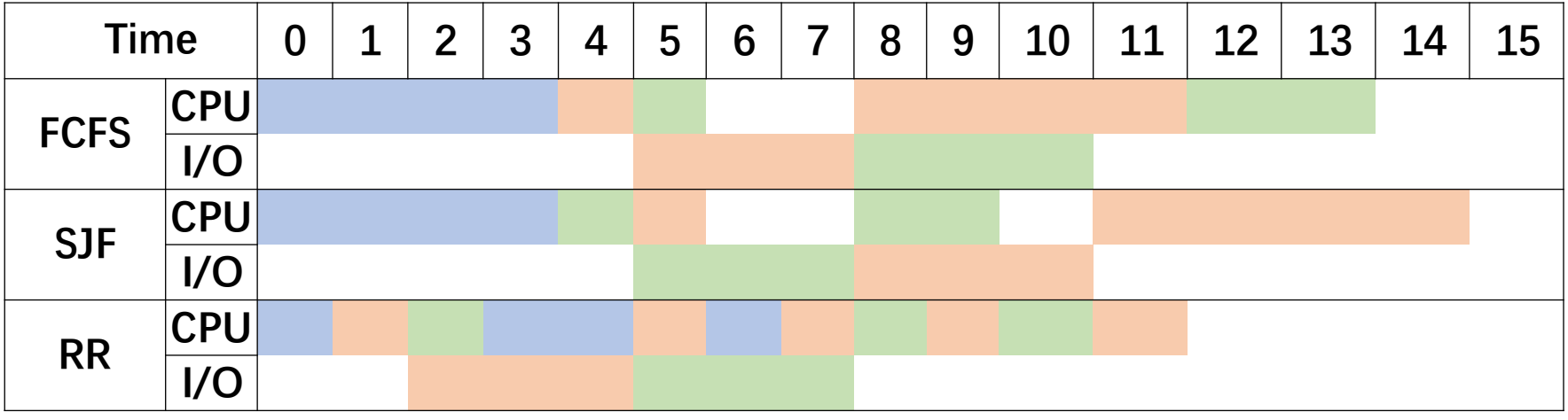
场景2：多样化测试


场景2响应时间
8
7
6
5
4
3
2
1
0
FCFS
SJF
RR
A
B
C


场景2周转时间
12
10
8
6
4
2
0
FCFS
SJF
RR
A
B
C

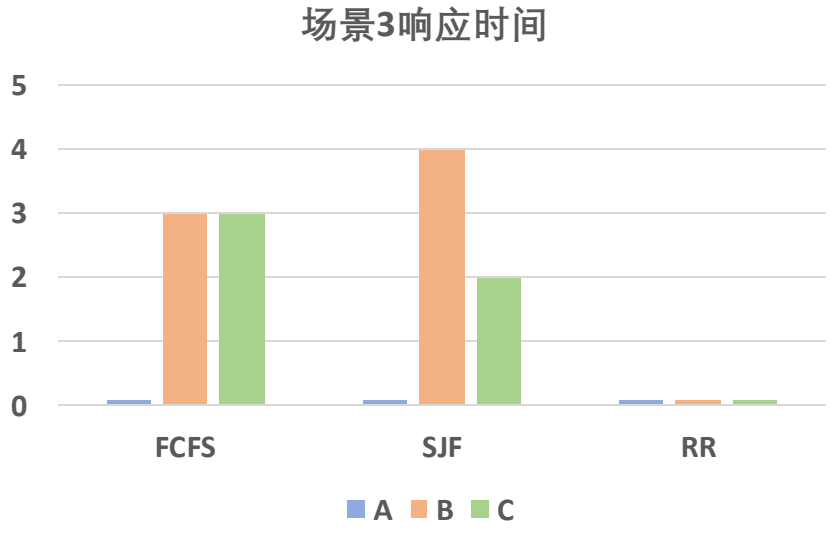
场景3：使用I/O读取测试数据集

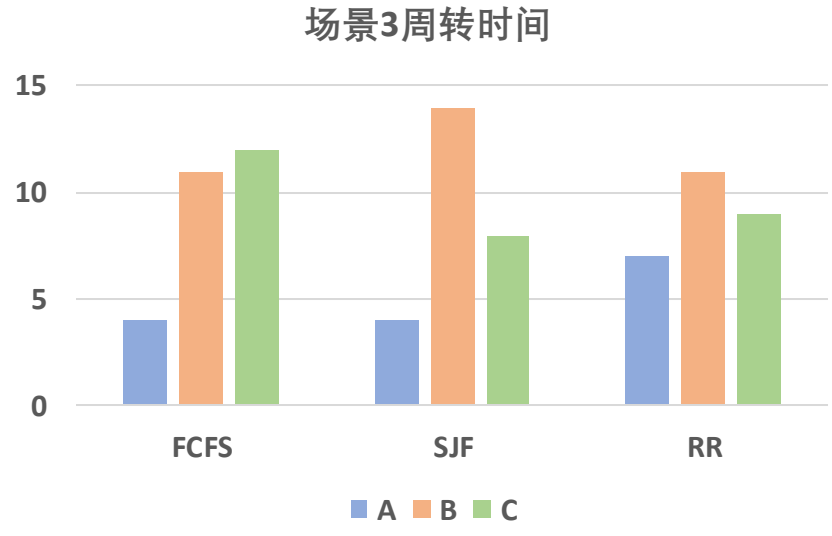
	到达时刻	I/O时间	工作时间
A	0	0	4
B	1	3	5
C	2	3	3



I/O任务先使用1单位工作时间发起I/O请求，在I/O返回后用剩余工作时间处理

场景3：使用I/O读取测试数据集

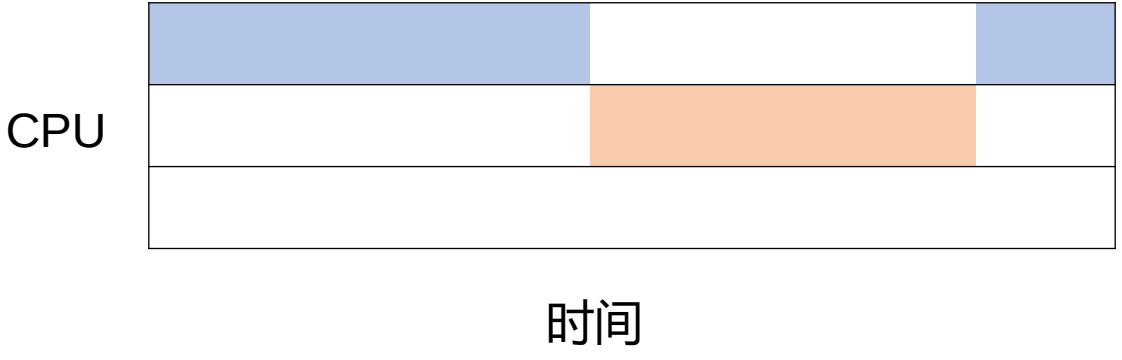




场景4：VIP优先功能

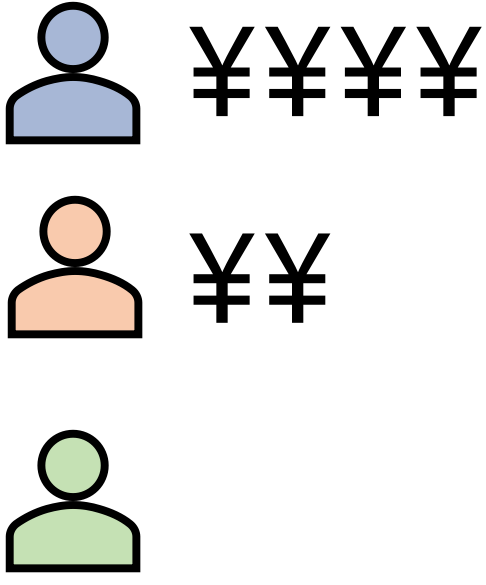
充值的VIP用户拥有更高的优先级:

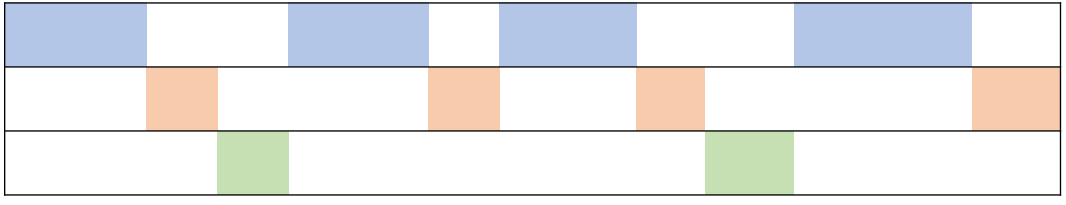
Multi-level Queues



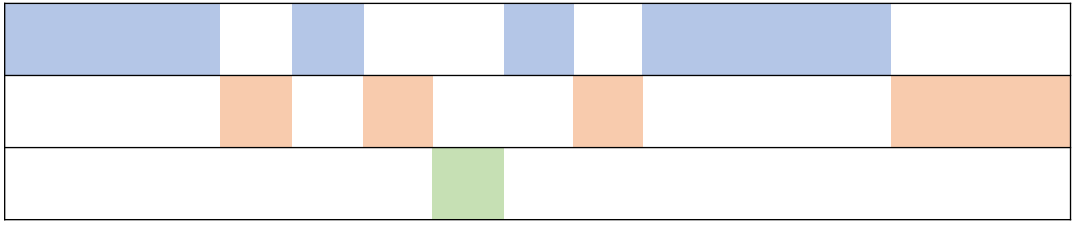
场景5：VIP优化——提高资源占比

充值的VIP用户占有更多的资源





？策略



？策略

更多功能:

- **视频浏览功能**
  - 实时调度
- **采用多核服务器**
  - 多核调度
  - 负载均衡
  - 能耗感知调度

Energy Aware Scheduling

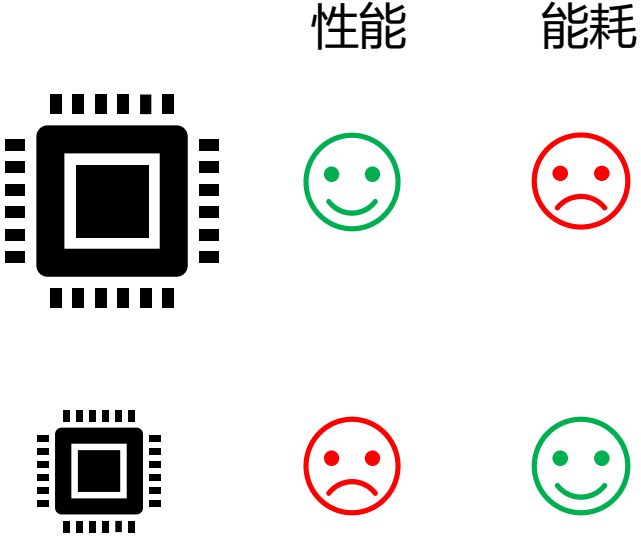
能耗感知调度

异构处理器架构

- **该类型架构上处理器的能力、能耗是不同的**

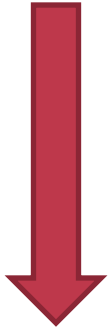
例：

  - 移动设备
  - ARM big.LITTLE架构



Linux Energy Aware Scheduling

- **目标**
  - 降低每条指令的平均能耗
  - 可接受的性能

$$\frac{Energy}{Instruction}$$ 

标准化CPU的处理能力

- **规定当前系统中，性能最高的CPU处理能力为1024**
  - 若CPU1需要CPU2的两倍耗时处理某任务
    - 则$Capacity_2 = 2 * Capacity_1$
- **负载追踪PELT的$CpuScaleFactor$与Capacity有关**
  - Load=1024的任务需要Capacity=1024的CPU一个单位时间处理

能耗模型（Energy Model）

- **记录CPU的处理能力与单位能耗的对应量化关系**
- **划分若干个性能域（Performance Domain，PD）**
  - 性能域内的CPU应是同质的
  - 性能域有不同Operation Performance Points（OPP）
    - 代表该模式下最高处理能力以及对应的单位能耗
    - 同一性能域的CPU必须处于同一操作性能点

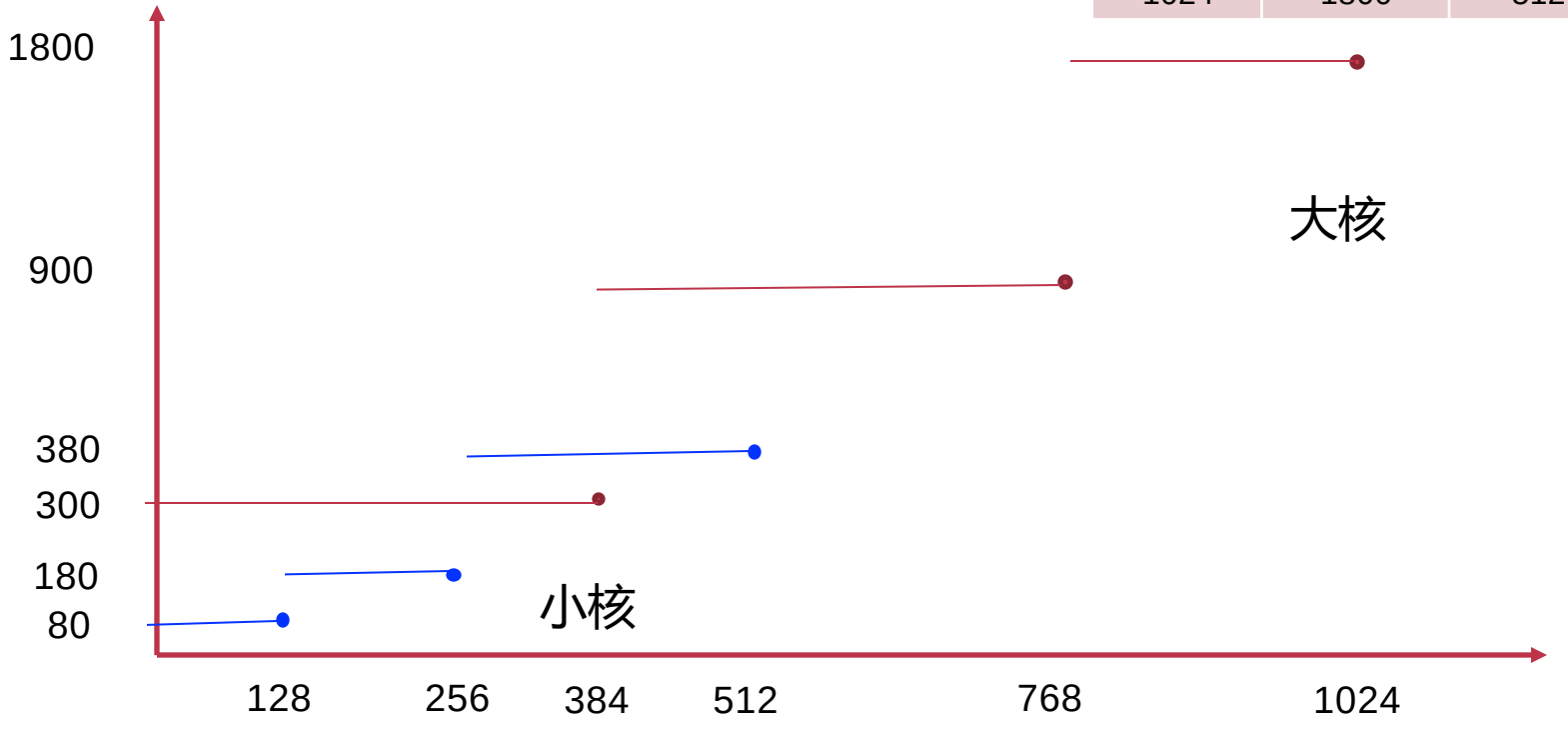
例：某大小核架构能耗模型（假设情况）			
大核（PD0）		小核（PD1）	
最高处理能力	单位能耗	最高处理能力	单位能耗
384	300	128	80
768	900	256	180
1024	1800	512	380

能耗模型

例：某大小核架构能耗模型（假设情况）			
大核（PD0）		小核（PD1）	
最高处理能力	单位能耗	最高处理能力	单位能耗
384	300	128	80
768	900	256	180
1024	1800	512	380

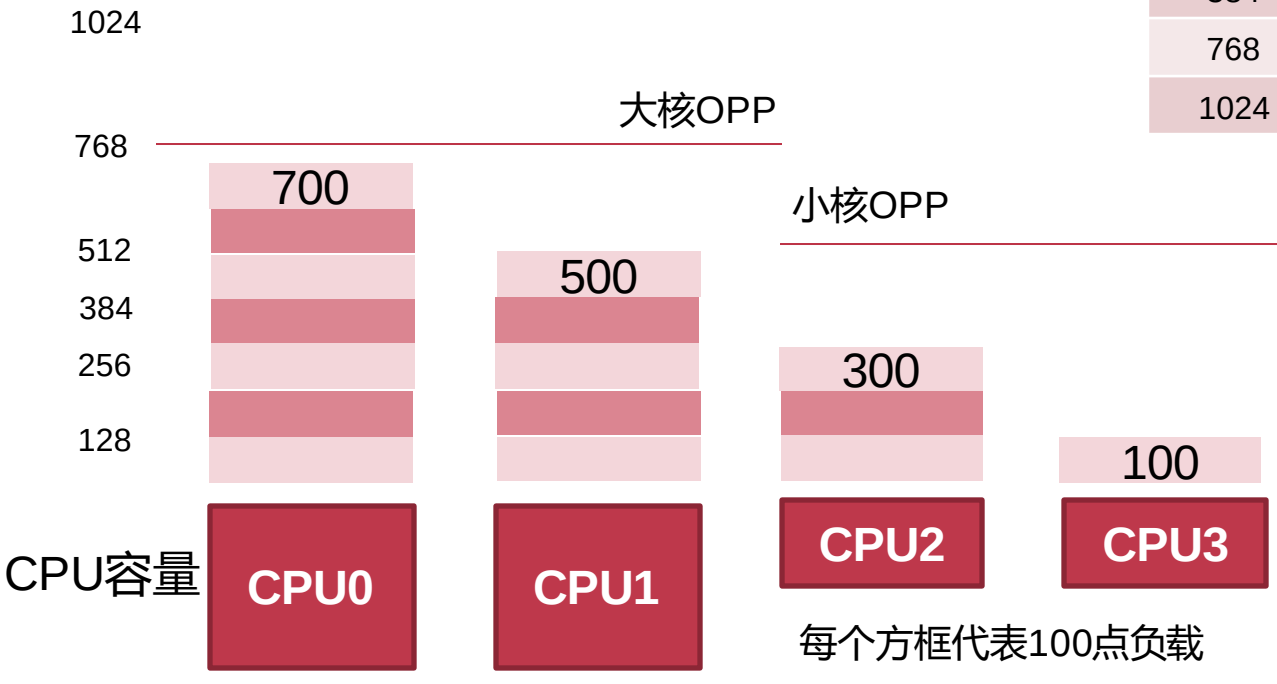
单位能耗

蓝线、红线的交叉点意味着什么？



例：EAS（Energy Aware Scheduler）调度

例：某大小核架构能耗模型（假设情况）			
大核（PD0）		小核（PD1）	
最高处理能力	单位能耗	最高处理能力	单位能耗
384	300	128	80
768	900	256	180
1024	1800	512	380



$$Energy_0 = \frac{700}{768} * 900 = 820$$

$$Energy_1 = \frac{500}{768} * 900 = 586$$

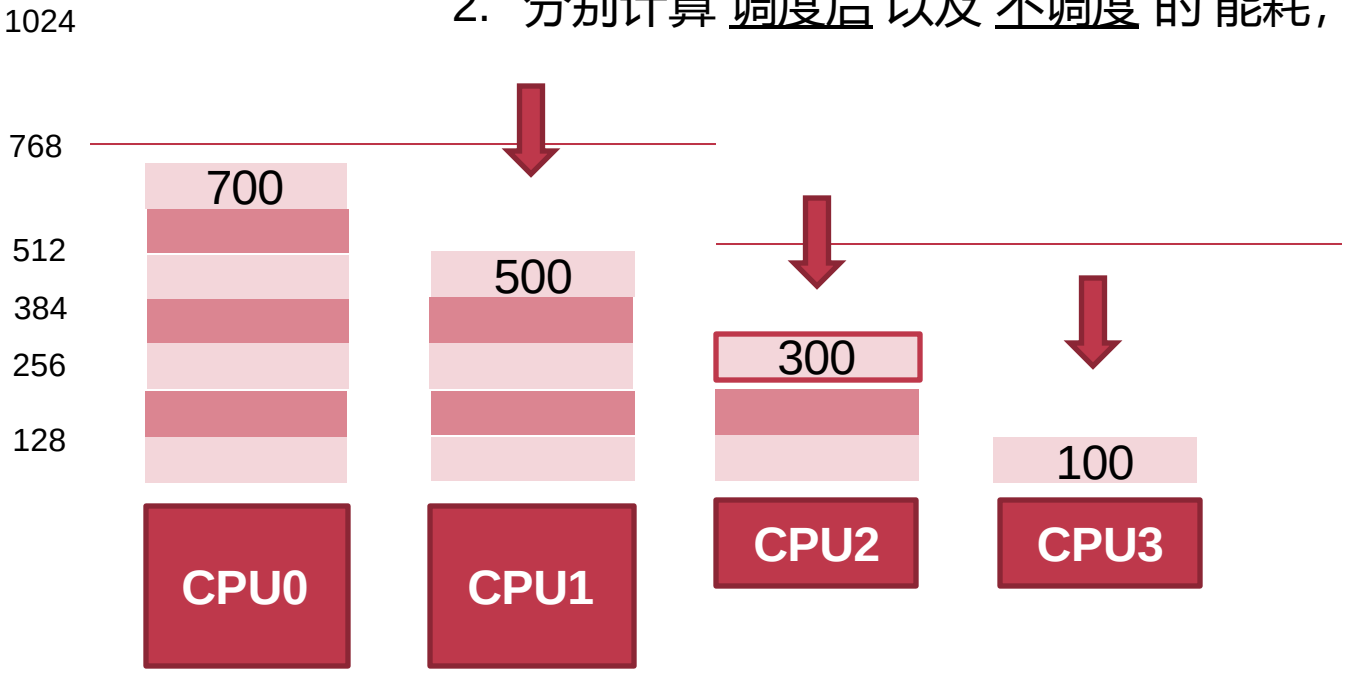
$$Energy_2 = \frac{300}{512} * 380 = 223$$

$$Energy_3 = \frac{100}{512} * 380 = 74$$

例：EAS调度

从CPU2调度100点负载的任务，选择目标CPU：

1. 从每个性能域选取当前剩余处理能力最多的CPU
2. 分别计算 调度后 以及 不调度 的 能耗，选择能耗最低的方案

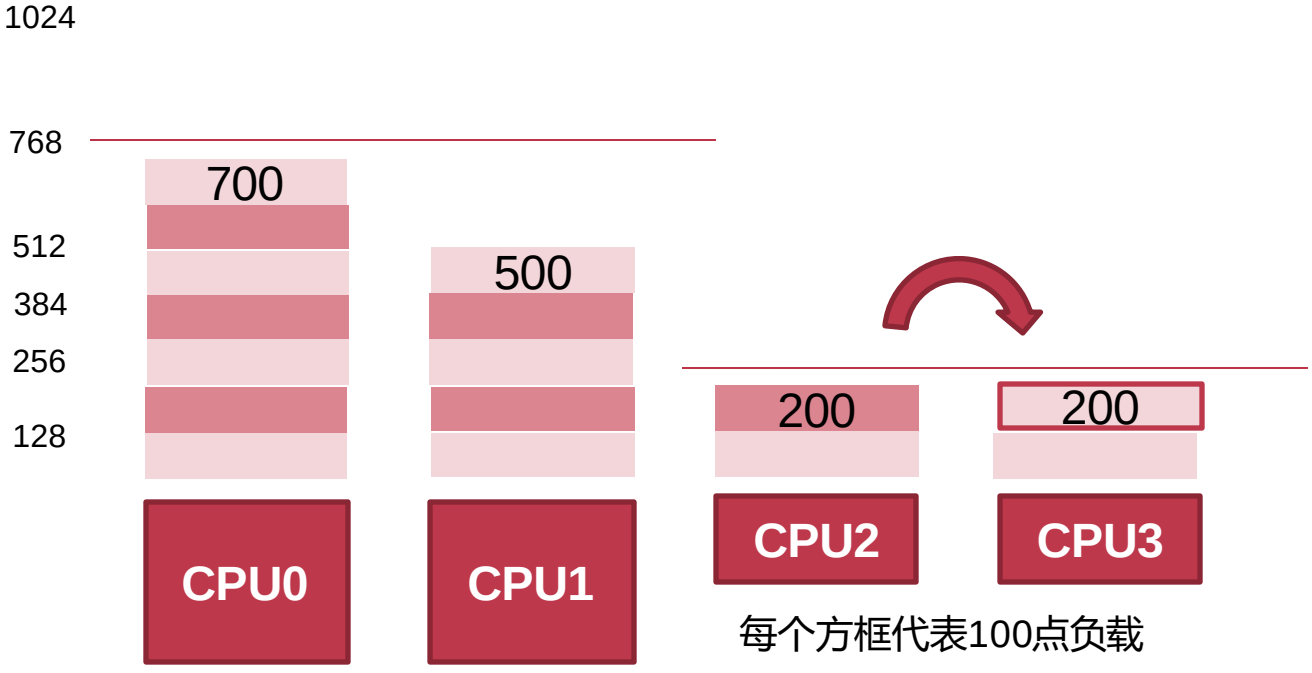


例：EAS调度

例：某大小核架构能耗模型（假设情况）			
大核（PD0）		小核（PD1）	
最高处理能力	单位能耗	最高处理能力	单位能耗
384	300	128	80
768	900	256	180
1024	1800	512	380

处理能力

最终选择从CPU2调度至CPU3

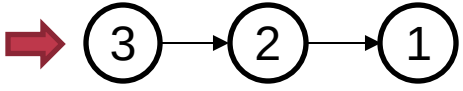


思考：EAS适用场景

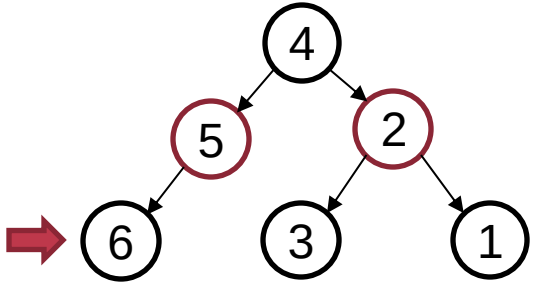
- 哪些使用场景是适合EAS的?

调度的 策略 V.S. 机制 (1)

- **策略**
  - 保存所有任务
  - 能够O(1)地获取最大优先级的任务
- **机制**
  - 队列



  - 红黑树



Linux调度策略的切换

- Linux调度策略
  - SCHED_FIFO
  - SCHED_RR
  - SCHED_OTHER
- 机制
  - chrt

```
➜  ~ sudo chrt --fifo -p 30 30152
➜  ~ chrt -p 30152
pid 30152's current scheduling policy: SCHED_FIFO
pid 30152's current scheduling priority: 30
```

DESCRIPTION top

chrt sets or retrieves the real-time scheduling attributes of an existing *pid*, or runs *command* with the given attributes.

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

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

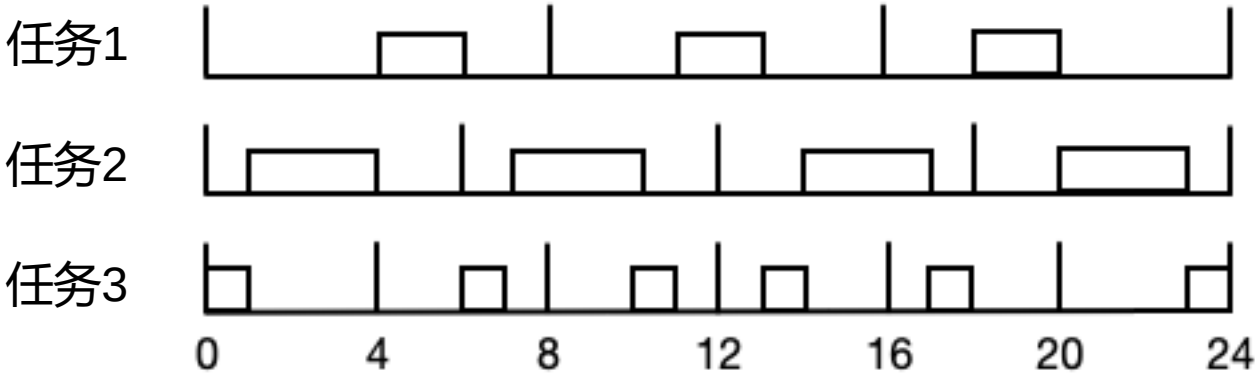
反映了CPU利用率

- **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

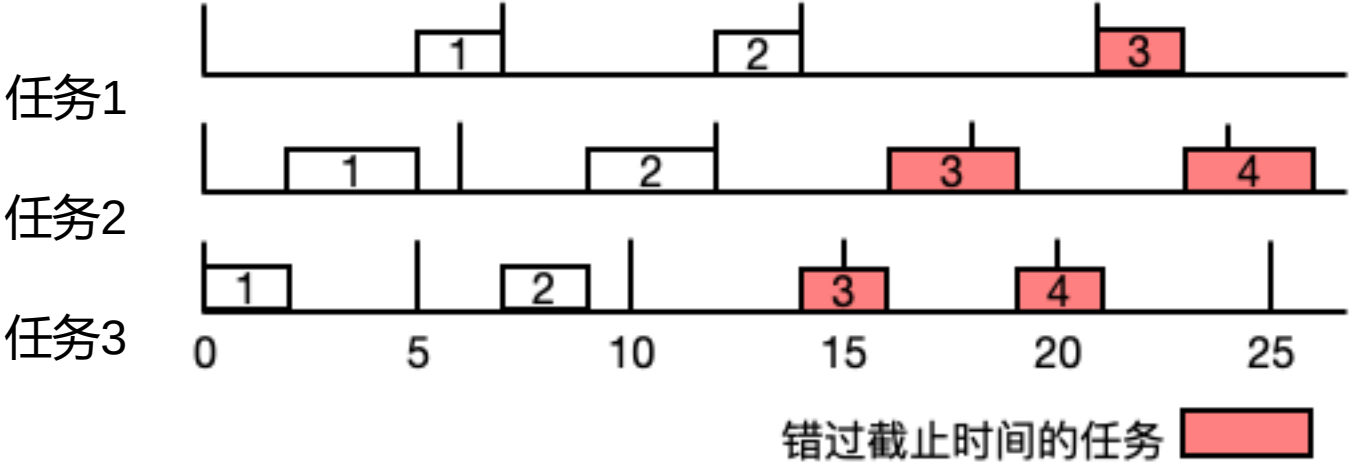


$$\sum_{i=1}^{n} \frac{C_i}{P_i} = 1$$

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

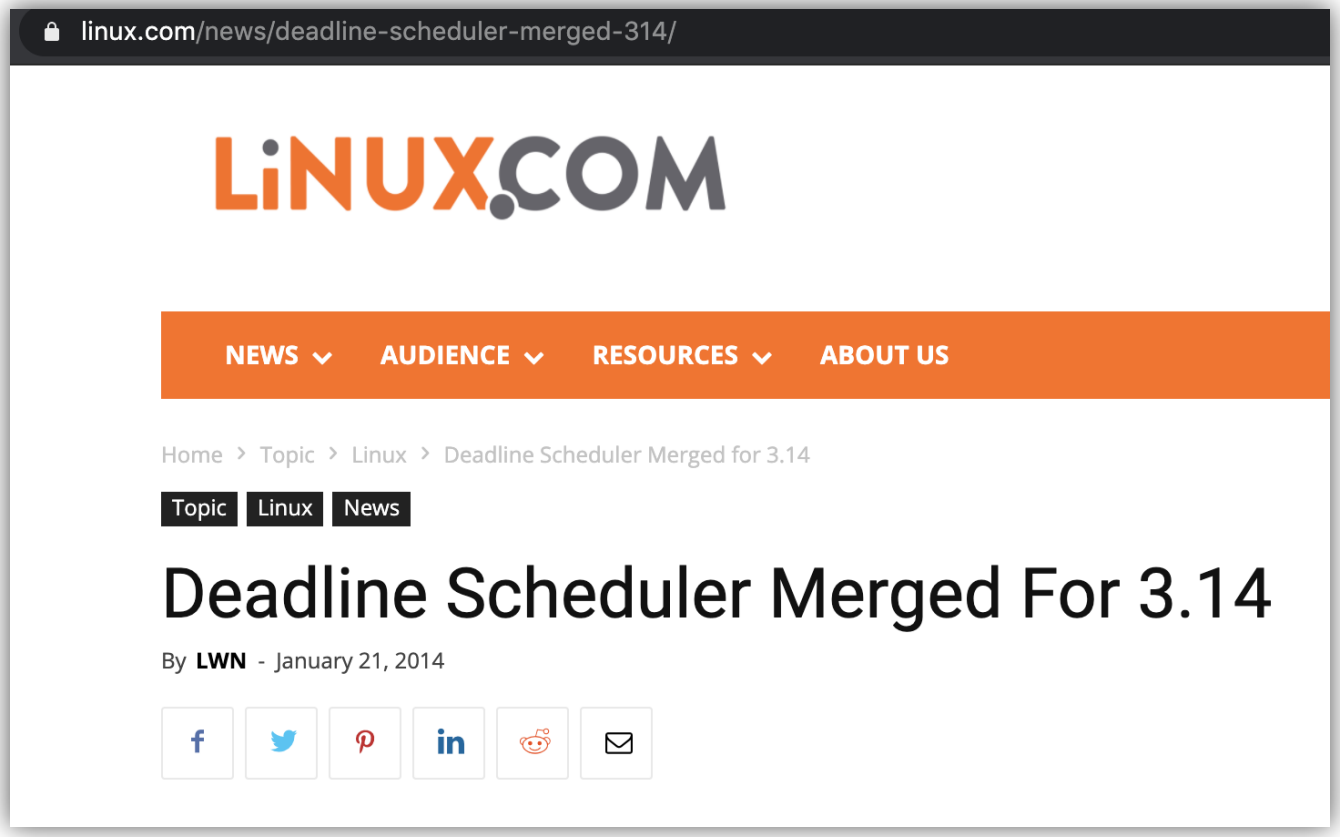
	执行时间	任务周期/截止时间
任务1	2	7
任务2	3	6
任务3	2	5



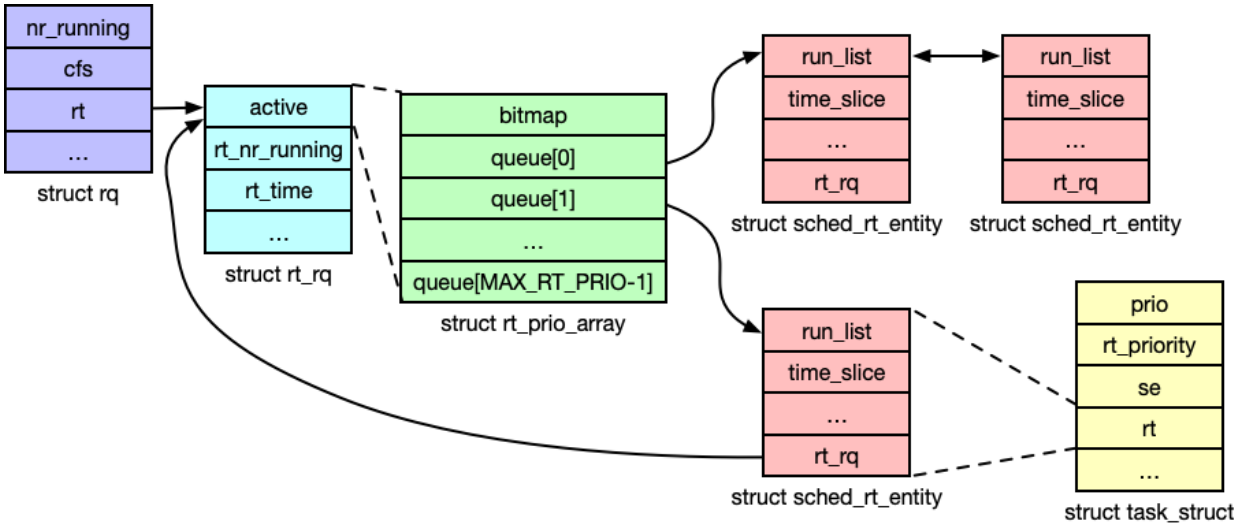
$$\sum_{i=1}^{n} \frac{C_i}{P_i} > 1$$

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

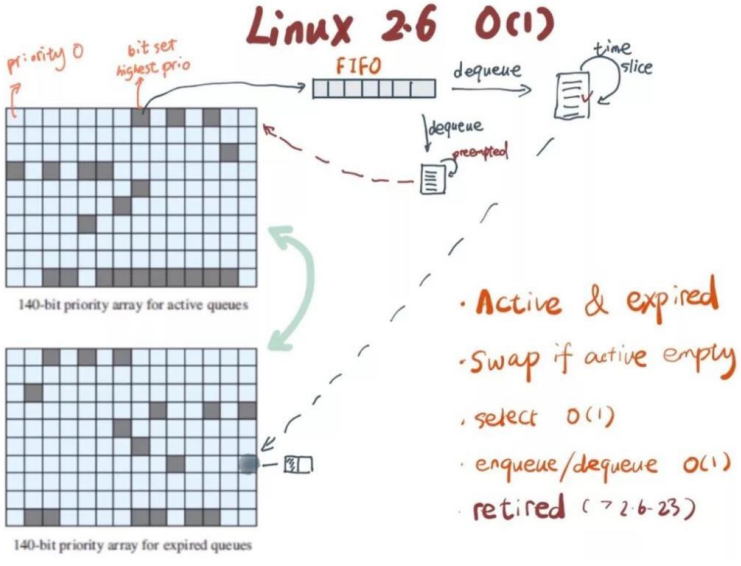


Linux Real-Time（软实时）Scheduler



- **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

O(1)调度器



- **2.6版本的O(1)调度器由Ingo Molnar设计实现**
- **设计动机：**
  - 为唤醒、上下文切换、定时器中断开销建立O(1)的调度器
- **进程有140种优先级，可用长度为140的array去记录优先级**
  - access是O(1)