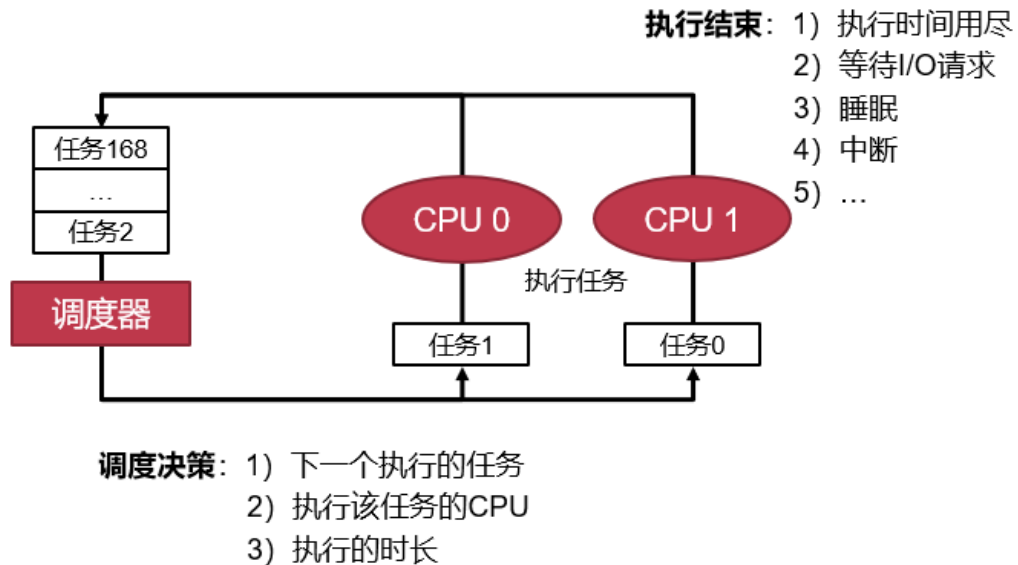


# Lecture8 CPU Sched

## 进程/线程调度

### 进程/线程调度



其中执行的时长可以在编译时被指定，如1ms/10ms，但也不是不变的，在动态运行的时候也会发生一些改变

## 什么是调度

协调对资源的使用请求



## 调度在不同场景下的目标

没有Uniform的调度，只有针对特定场景的优化

没有"One size fits all"

批处理系统



高吞吐量

交互式系统



低响应时间

网络服务器



可扩展性

移动设备



低能耗

实时系统



实时性

**一些共有的目标：**  
高资源利用率  
多任务公平性  
低调度开销

## 调度器的目标

1. **降低周转时间**：任务第一次进入系统到执行结束的时间
2. **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
3. **实时性**：在任务的截止时间内完成任务
4. **公平性**：每个任务都应该有机会执行，不能饿死
5. **开销低**：调度器是为了优化系统，而非制造性能瓶颈
6. **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/任务间有复杂的交互
- 调度目标的多样性

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

1. 调度开销&调度效果
2. 优先级&公平
3. 能耗&性能


## 经典调度

---


# First Come, First Served




大家排队  
先来后到!



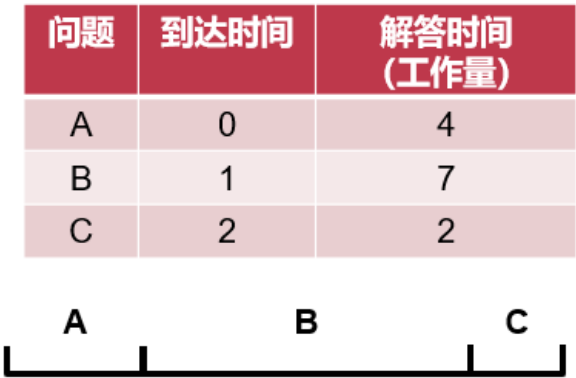
得嘞, 我第一



C,先来后到!




我的问题很简单  
却要等那么长时间...




先到先得：简单、直观  
问题：平均周转、响应时间过长

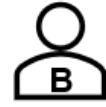
# Shortest Job First



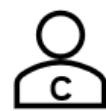
简单的问题先来



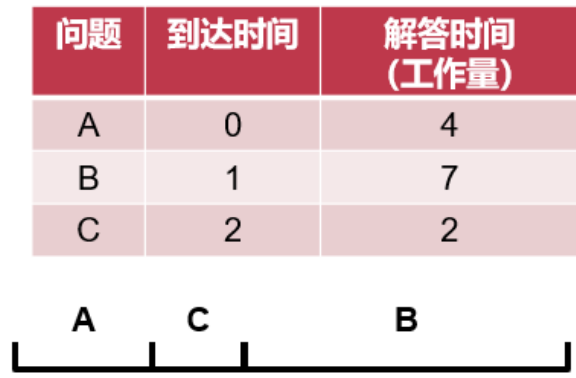
我最先到,  
我还是第一!



万一再来个短时间的  
D, 那我要等死了...



我可以先于B了☺



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)

- 1. 每次任务执行在一定时间后会被切换到下一任务，而非时间终止
- 2. 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...

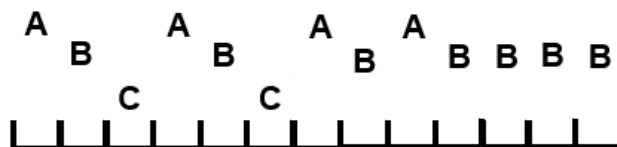


学霸的响应时间短  
了好多



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

Q：什么情况下RR的周转时间问题最明显？

A：每个任务的执行时间差不多相同

Q：时间片长短应该如何确定？

A1：过长->First Come, First Served

A2：过短->调度开销很大

## 优先级调度

### 调度优先级

优先级用于确保重要任务被优先调度

### 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制度（也可以使用其它调度策略）

### 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



**问题：**  
A被C占有的资源**阻塞**  
优先级较低的B先于A学习



优先级：A>B>C

2. 抢占C  
申请OS书失败  
等待

3. B优先级高于C  
可以向学霸学习

1. 申请OS书成功

## 解决方法：优先级继承



**解决方案：**  
A暂时将**优先级转移**给C  
让C尽快归还OS书



优先级：A>B>C

2. 抢占C  
转移优先级给C

4. 申请OS书成功  
继续学习

3. 归还OS书  
返回优先级

1. 申请OS书成功

## 目前的一些限制

- 周转时间过长

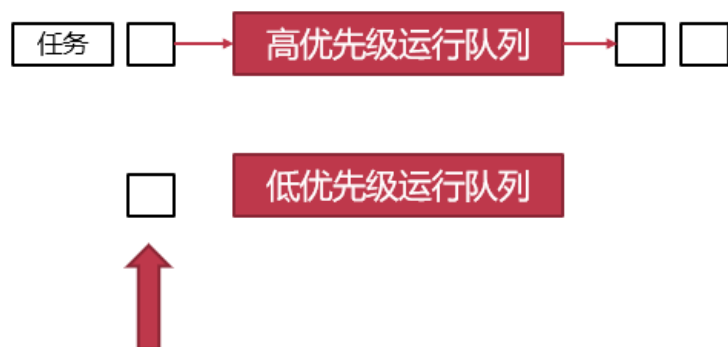
FCFS

- 依赖对于任务的先验知识

1. 预知任务执行时间->SJF
2. 预知任务是否为I/O密集型任务->MLQ

## 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

- 解决方案：优先级的动态调整

操作系统中的工作场景是动态变化的

- 静态优先级的问题
  1. 资源利用率低
  2. 优先级反转

## MLFQ的主要目标与思路

MLFQ: Multi-level Feedback Queue

- 一个无需先验知识的通用调度策略
  1. 周转时间低，响应时间低
  2. 调度开销低
- 通过动态分析任务运行历史，总结任务特征

e.g. 页替换策略、预取

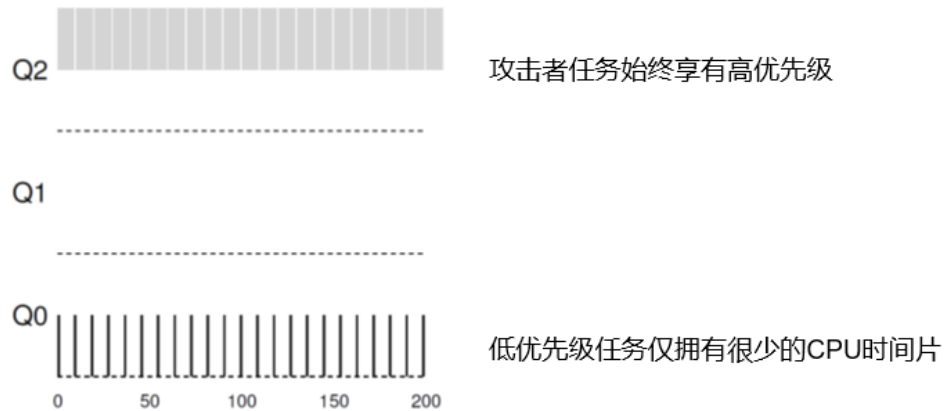
但是如果工作场景变化频繁，效果会很差

## MLFQ基本算法

- 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round-robin
- 规则3：任务被创建时，**假设该任务是短任务**，为它分配最高的优先级
- 规则4：一个任务时间片耗尽后（无论中间放弃了多少次CPU，它的时间片不会被重置），它的优先级会被降一级

解决了抢占CPU时间的攻击（如果重置的话攻击者几乎独占CPU!!!）

# 攻击示例



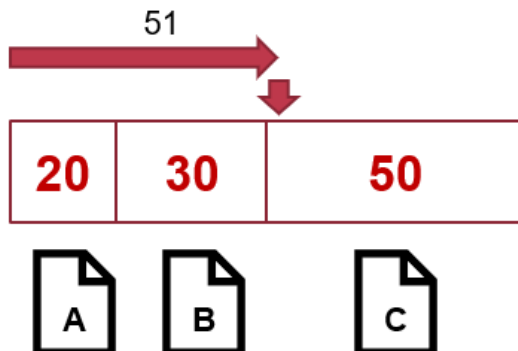
- 规则5: 在某个时间段S之后, 将系统中所有任务优先级提为最高
  - 效果1: 避免长任务被饿死 (最高级采用RR, 长任务一定会被调度到)
  - 效果2: 针对任务动态变化的场景 (MLFQ会定时重新审视每一个任务)

## 公平共享调度

### 彩票制度

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()
```

- Pros
  - 1. 简单
- Cons
  - 1. 不精确——伪随机不是真随机



## 步幅调度

### Stride Scheduling

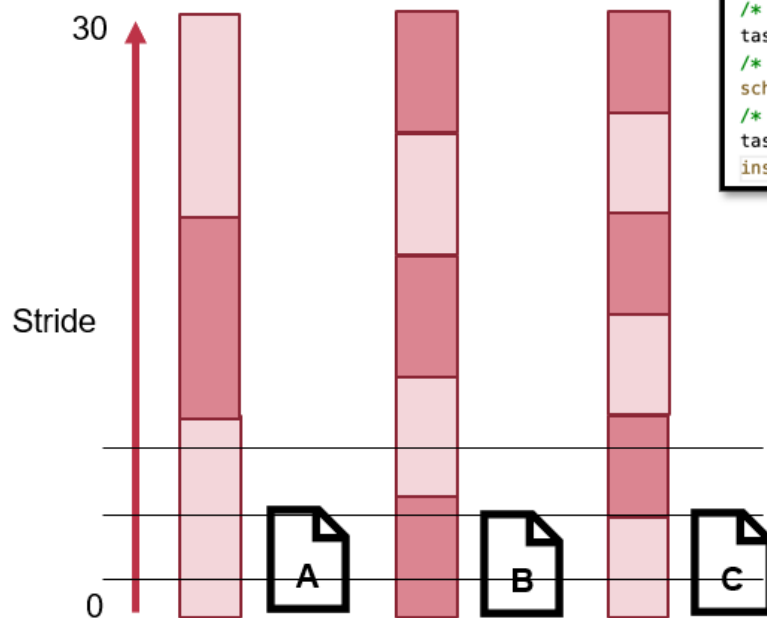
- 可以看做**确定性版本**的彩票调度
  - 可以沿用tickets的概念
- Stride——步幅，任务一次执行增加的**虚拟时间**
  - $stride = \frac{MaxStride}{ticket}$ 
    - MaxStride是一个足够大的整数
    - 本例中设为所有tickets的最小公倍数
- Pass——累计执行的虚拟时间

	Ticket	Stride
A1	30	10
A2	50	6
B1	60	5

MaxStride = 300

挑选pass最小的任务优先执行

## 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

### Summary

	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

## 多核调度策略

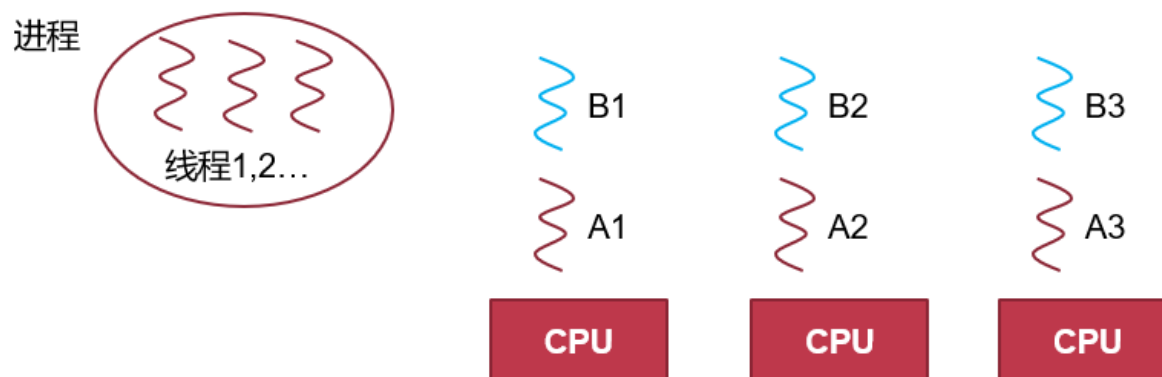
需要考虑的问题：

1. 一个进程的不同线程可以在不同CPU上同时运行
2. 同一个进程的线程间很可能有依赖关系

## 群组调度

Gang Scheduling

- 在多个CPU上同时执行一个进程的多个线程



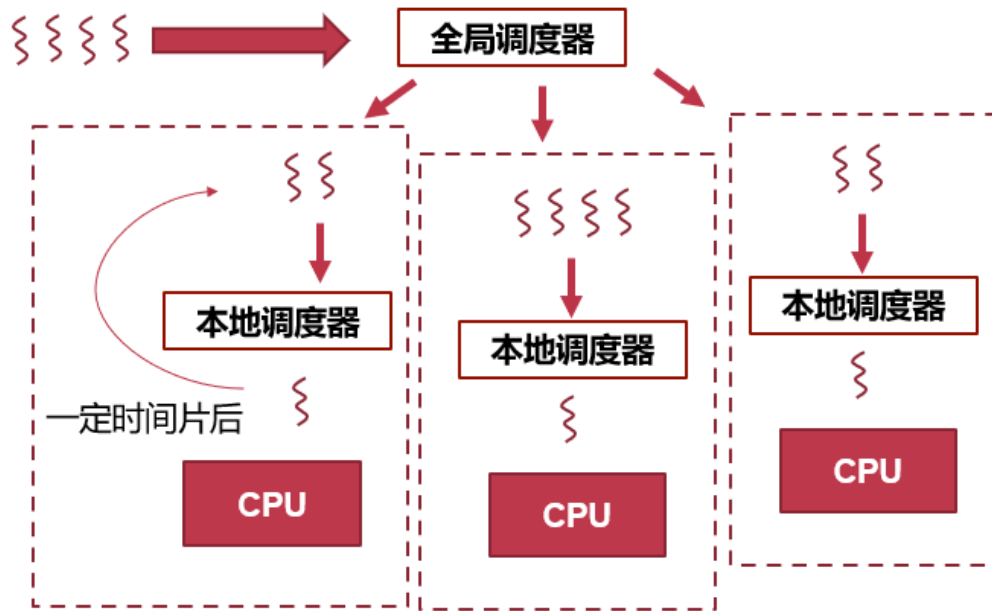
- 组内任务都是关联任务，需要尽可能同时执行

## 全局使用一个调度器的问题

1. 所有CPU竞争全局调度器
2. 同一个线程可能在不同CPU上切换

Locality差->切换的开销很大(Cache, TLB)

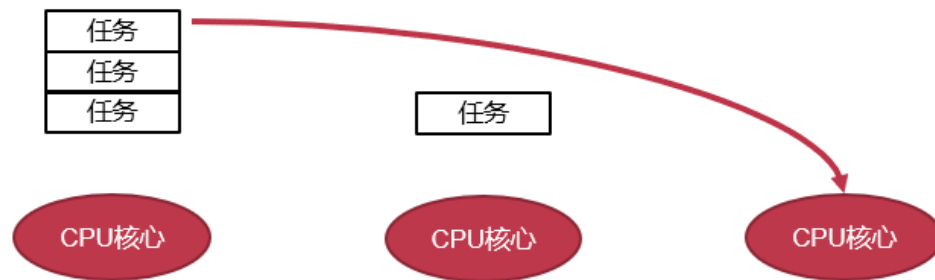
## Two-level Scheduling



66

## 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



## 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

72

## 调度小结

---

1. 调度指标
2. 经典调度
3. 优先级调度
4. 公平共享调度
5. 多核调度