

CPU Scheduling (OSTEP: Sec 7-10)

**[https://pages.cs.wisc.edu/~remzi/
OSTEP/](https://pages.cs.wisc.edu/~remzi/OSTEP/)**

Outline

- **CPU 调度基本策略**
- Multi-Level Feedback Queue
- Proportional Share
- 多核 CPU 调度

回顾

- Exception: Interrupt, Trap, Fault, Abort
- Process: States, Fork, Execve, Waitpid,
- Signal: 分类, 接收 / 发送, handling...

这些主要讨论外部事件或者程序主动触发的控制流切换

CPU 调度

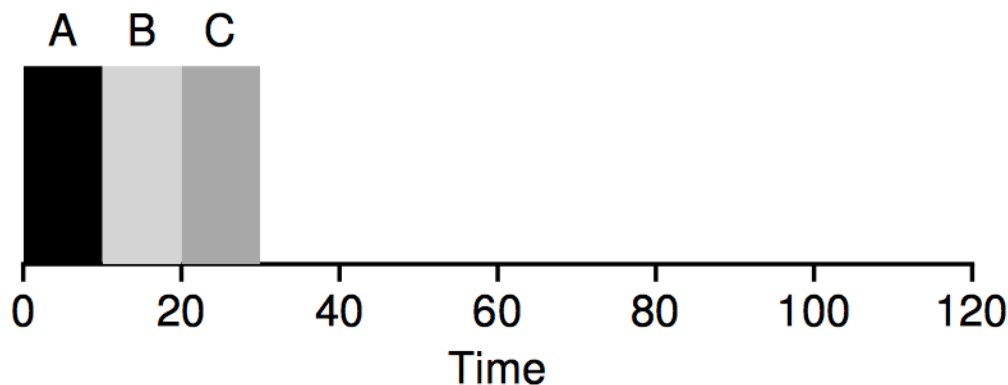
- OS kernel 如何实现多进程 / 多任务共享 CPU ?
- 调度：进程切换的决策
 - 优化目标 1：更小的 turnaround time（周转时间），任务从到达整体完成的延迟

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- 多任务：尽可能小的平均 turnaround time

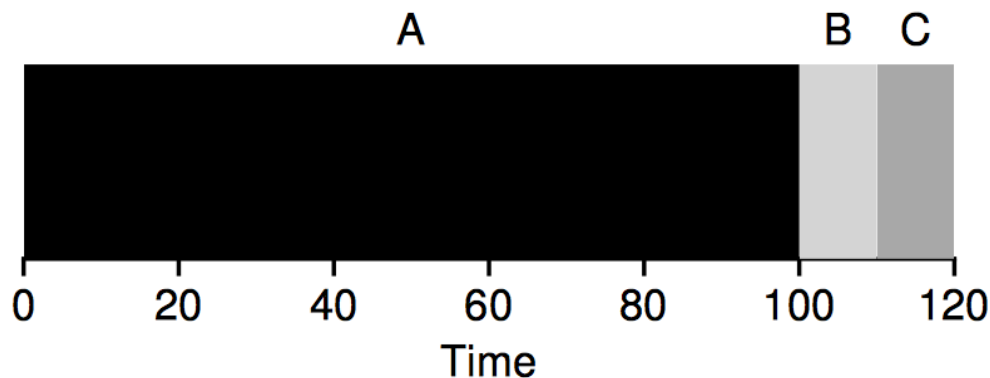
First In, First Out (FIFO)

- 也叫 First Come, First Served (FCFS)
- 例如 A, B, C 几乎同时到达, $T_{\text{arrival}} = 0$
- 平均 turnaround time = $(10+20+30)/3=20$
- 优点：简单



First In, First Out (FIFO)

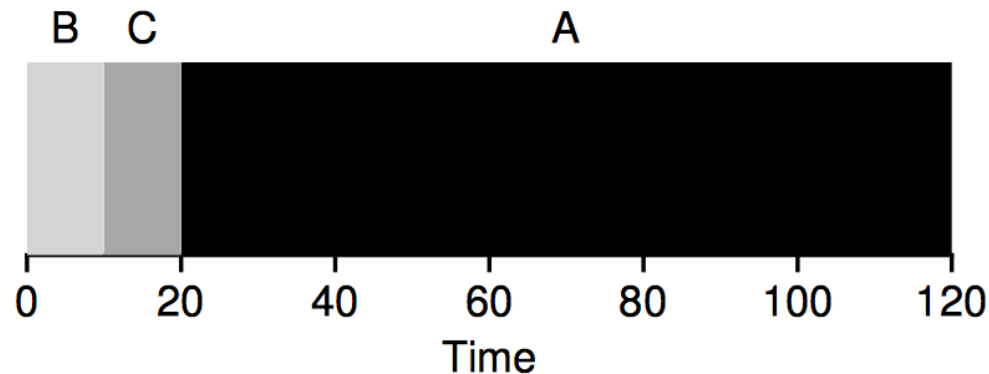
- 缺点
 - 当任务长度差距较大时，如下图
 - 平均 turnaround time = $(100+110+120)/3 = 110$



如何解决这个问题？

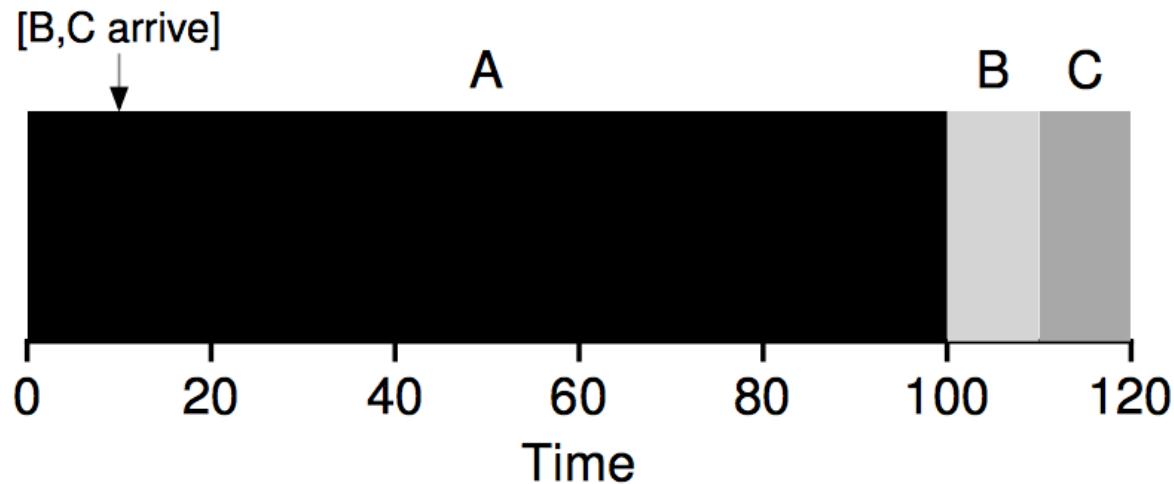
Shortest Job First (SJF)

- 有助于减小平均 turnaround time ， 例如上面的情况
- 平均 turnaround time = $(10+20+120)/3 = 50$



Shortest Job First (SJF)

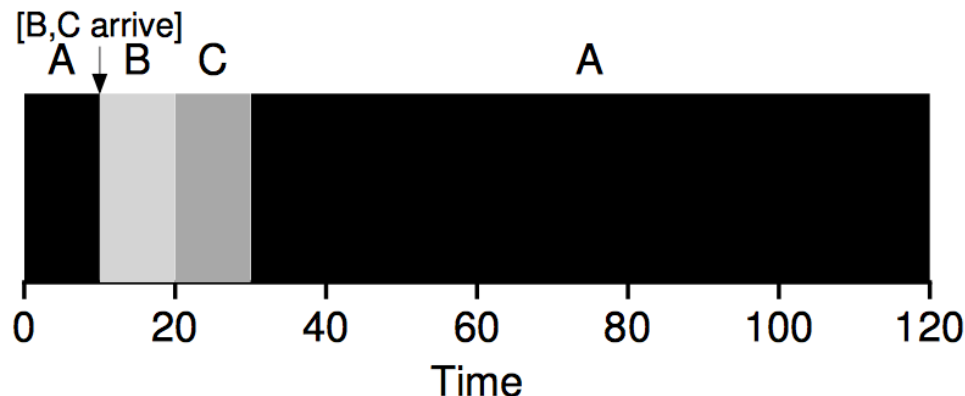
- 对于任务同时到达， SJF 是最优调度算法了
- 但如果长任务 A 先到达，则 SJF 没有办法
 - 例如 A 在 0 时刻到达， B 和 C 在 10 时刻到达
 - 平均 Turnaround time = $(100+100+110)/3 = 103.3$



这个问题的关键在哪里？如何解决？

Shortest Time-to-Completion First (STCF)

- 上述问题改进的关键：
 - 不要把一个任务执行到底才切换
 - 允许任务抢占（时间片 / 中断作为切换点）
- 优化算法： STCF
 - 在 SJF 上增加抢占功能，剩余时间最短的任务优先
 - **Preemptive Shortest Job First (PSJF)**
 - 平均 turnaround time = $(120+10+20)/3 = 50$



平均 Turnaround Time 的问题

- 平均 Turnaround Time 反映整体性能
 - 但有些 Job 被平均（自己牺牲，换来整体平均更好，例如长任务）
- 不能做到公平（Fairness）
 - 用户感受可能不好
 - 极端情况出现“饥饿问题”（job 不断到达，某些 Job 一直不被调度）
- 如果所有 job 属于一个用户，还可以接受
 - 如果 jobs 属于不同用户（尤其是付费用户），则无法接受牺牲
 - 效率和公平往往不能兼得

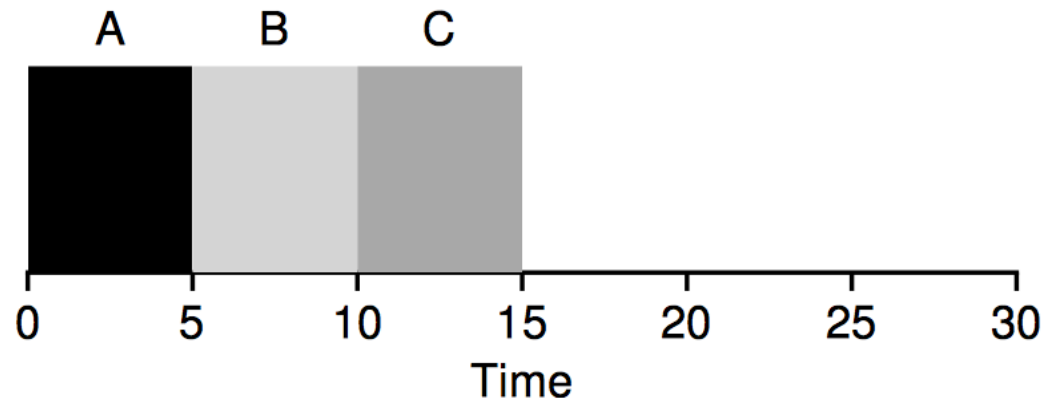
CPU 调度

- **优化目标 2：Response time**

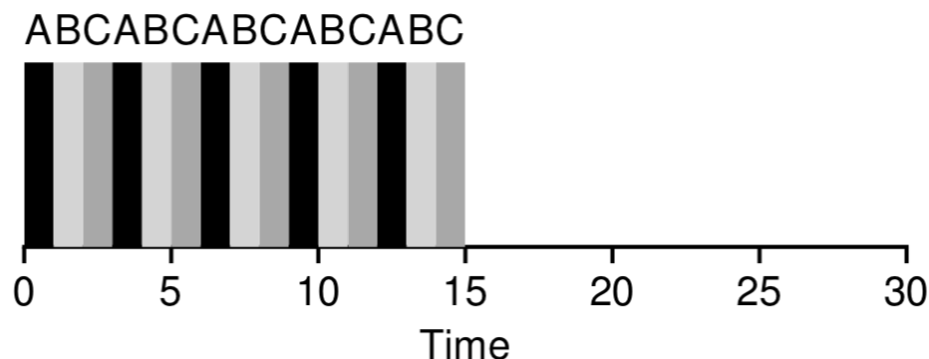
- 任务的**第一次被调度时间**与到达时间之差

$$T_{response} = T_{firstrun} - T_{arrival}$$

- 之前的调度算法（如 SJF）会导致 response time 很差（B 和 C）



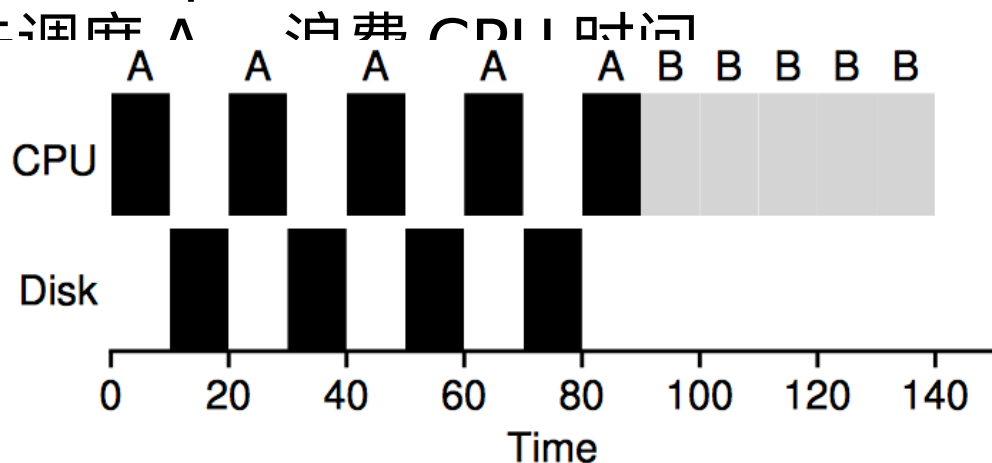
Round Robin



- 基于时间片轮转
 - 降低 response time
 - 增加 context switch 开销
 - 增大时间片能减小开销的比例，但也会增大 response time
 - 所以需要在二者之间 tradeoff
 - 偏重公平 (response time) : RR
 - 不看重公平 (turnaround time) : SJF, STCF
- Systems 领域多数情况都是在 tradeoff , 某方面最优一般都伴随其他方面的副作用
- e.g., CAP, RUM

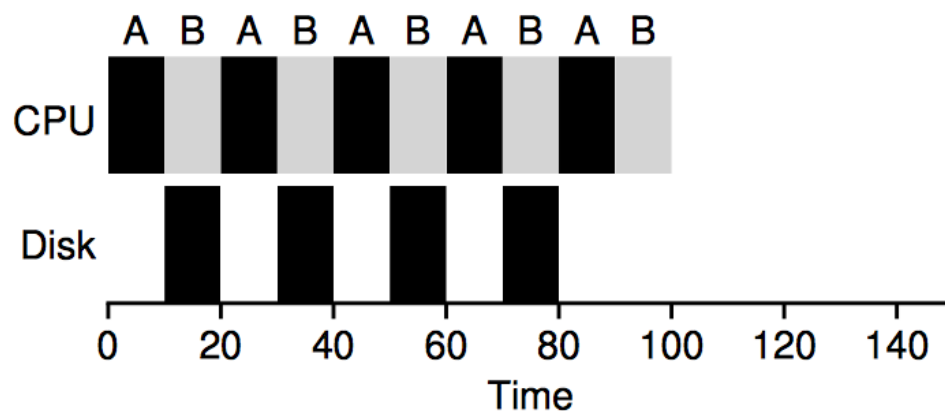
考虑 I/O

- I/O 操作远比 CPU 周期要慢
- 进程执行 I/O 操作时，要 block 住，直到 I/O 完成
- STCF 的问题
 - A 和 B 都是 50ms 的任务
 - 每次 interrupt，A 剩余的 CPU 时间都小于 B，那么会优先调度 A，浪费 CPU 时间



考虑 I/O

- 解决方案
 - I/O 时将任务移出就绪队列
 - 每次 A 开始 I/O 时，调度器只剩 B 一个选项，调度 B
 - 当 I/O 完成时（interrupt），A 回到就绪队列，调度 A



课堂练习

例 1：假定要在一台处理器上执行如下图所示的作业，它们在 0 时刻以 1, 2, 3, 4, 5 的顺序到达。给出采用下列调度算法时的调度顺序、平均周转时间(turnaround time)和平均响应时间(response time)

(1) FCFS

(2) RR(时间片为 1, 不考虑优先级)

(3) 非抢占式 SJF(shortest job first)

作业	执行时间	优先级
1	10	3
2	1	1
3	2	2
4	3	4
5	5	2

练习答案

(1) FCFS: (2 分)

P1	P2	P3	P4	P5	
0	10	11	13	16	21

$$\text{平均响应时间} = (0 + 10 + 11 + 13 + 16) / 5 = 10$$

$$\text{平均周转时间} = (10 + 11 + 13 + 16 + 21) / 5 = 14.2$$

(2) RR(TQ=1)

P1	P2	P3	P4	P5	P1	P3	P4	P5	P1	P4	P5	P1	P5	P1	P5	P1	P1	P1	P1	P1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
20	21																			

$$\text{平均响应时间} = (0 + 1 + 2 + 3 + 4) / 5 = 2$$

$$\text{平均周转时间} = (21 + 2 + 7 + 11 + 16) / 5 = 11.4$$

练习答案

(3) SJF

P2	P3	P4	P5	P1	
0	1	3	6	11	21

$$\text{平均响应时间} = (11 + 0 + 1 + 3 + 6) / 5 = 4$$

$$\text{平均周转时间} = (21 + 1 + 3 + 6 + 11) / 5 = 8.4$$

课堂练习

- 有 5 个批处理作业 A ~ E ， 它们几乎同时到达一个计算中心。估计它们的运行时间分别是 10、6、2、4 和 8 分钟。其优先极（外部设定）分别是 3、5、2、1 和 4，其中 5 为最高级。对于下列每种调度算法，计算平均周转时间，可忽略进程切换的开销。
 - 轮转法
 - 优先级调度
 - 先来先服务
 - 最短作业优先

Outline

- CPU 调度基本策略
- **Multi-Level Feedback Queue**
- Proportional Share
- 多核 CPU 调度

以上调度算法的共同问题

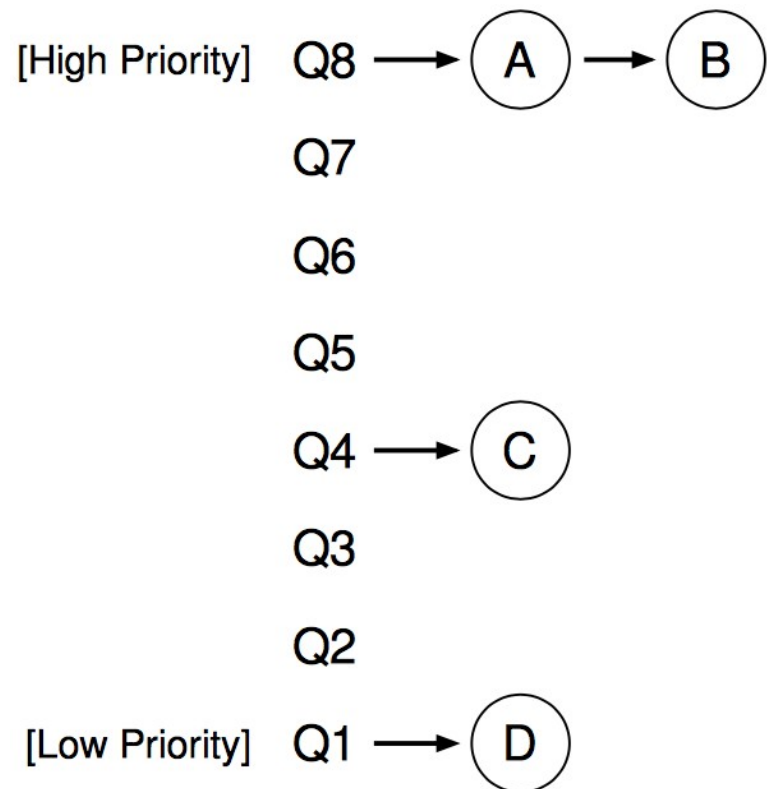
- 假设预先知道每个任务的执行时间
- 但实际上我们一般是没法预知的

Multi-Level Feedback Queue (MLFQ)

- 提出者
 - Corbato , 1962 , Compatible Time-Sharing System (CTSS) 的一部分
 - Corbato 后来参与 Multics 操作系统的工作
 - 后来获得 ACM 图灵奖
- MLFQ 目标
 - 不知道任务长度的前提下, 减小 response time
 - 但又不像 RR 那样伤害 turnaround time

Multi-Level Feedback Queue (MLFQ)

- 分为多个优先级队列
- 规则
 - **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
 - **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.



Multi-Level Feedback Queue (MLFQ)

- **Workload 分析**

- 一种是 interactive 任务，经常与 I/O 交互（包括磁盘、网络、键盘鼠标等），让出 CPU，对 response time 要求高
- 一种是 longer-running CPU-bound 任务，对 response time 要求不高，但对 turnaround time 要求高

- 在 System 研究中，看似矛盾的两方面往往不会同时出现，这需要对 workload 的深入洞察

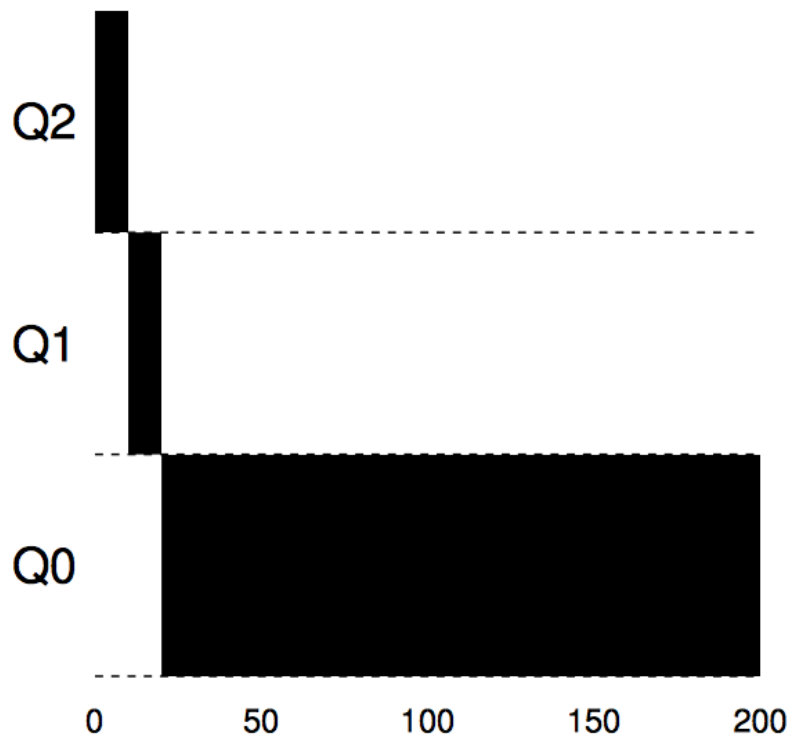
- 例如上述分类中，两类 job 并不都是 turnaround time 和 response time 都需要，而是每类偏重一个方面，这样就容易优化了

Multi-Level Feedback Queue (MLFQ)

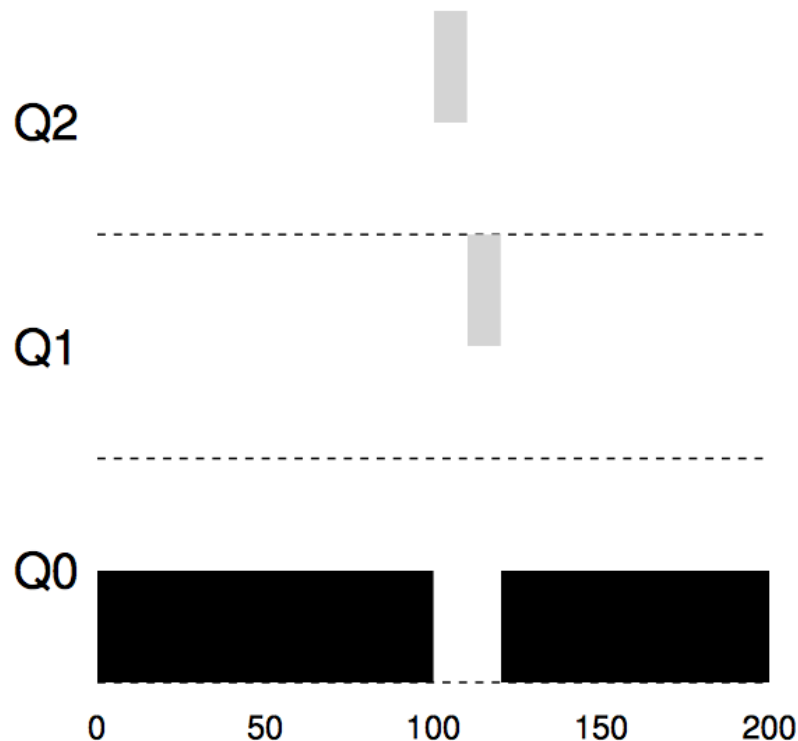
- Attempt #1: How To Change Priority
 - **Rule 3:** When a job enters the system, it is placed at the highest priority (the **topmost** queue).
 - **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue). 【针对 CPU-bound 任务】
 - **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level. 【针对 interactive 任务】

Multi-Level Feedback Queue (MLFQ)

- 例 1. 长任务



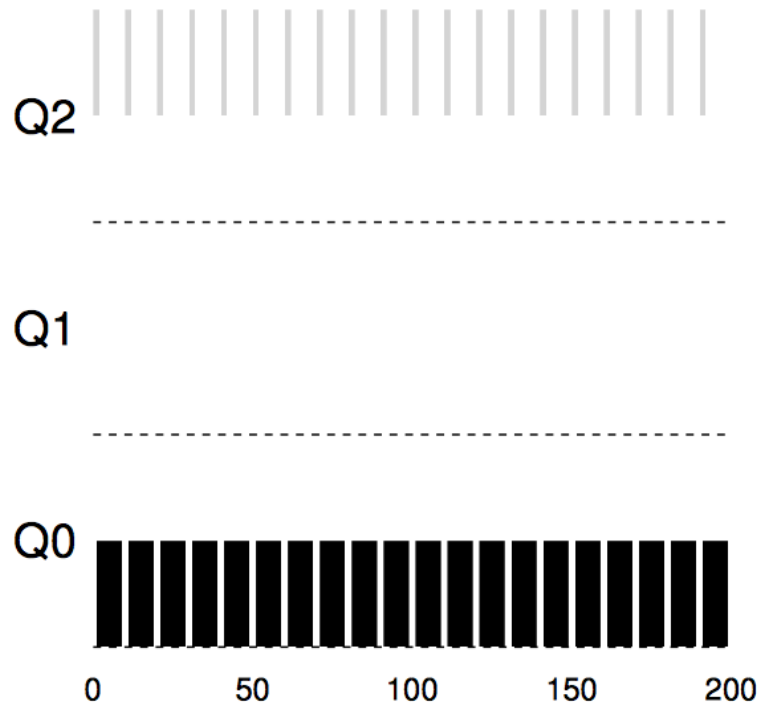
- 例 2. 又来了交互式任务



效果上近似于 SJF，有利于周转时间
同时交互式任务的响应时间很短

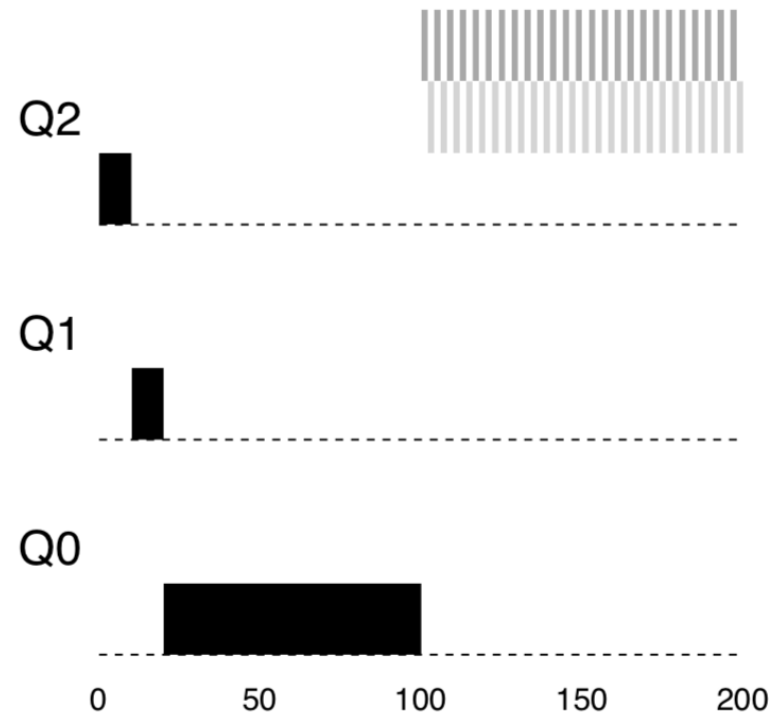
Multi-Level Feedback Queue (MLFQ)

- Interactive (I/O bound) 任务
 - 按照 Rule 4b, 由于 I/O 阻塞, 会提前释放 CPU , 所以一直留在高优先级



Multi-Level Feedback Queue (MLFQ)

- MLFQ 问题总结
 - 1. CPU-bound 型任务的饥饿问题
 - 如果交互式任务数量较多，它们会交替使用 CPU
 - 低优先级的 CPU-bound 型任务可能饥饿（调度算法一定要避免的灾难）
 - * 系统工作，首先要考虑避免灾难，然后才是优化性能



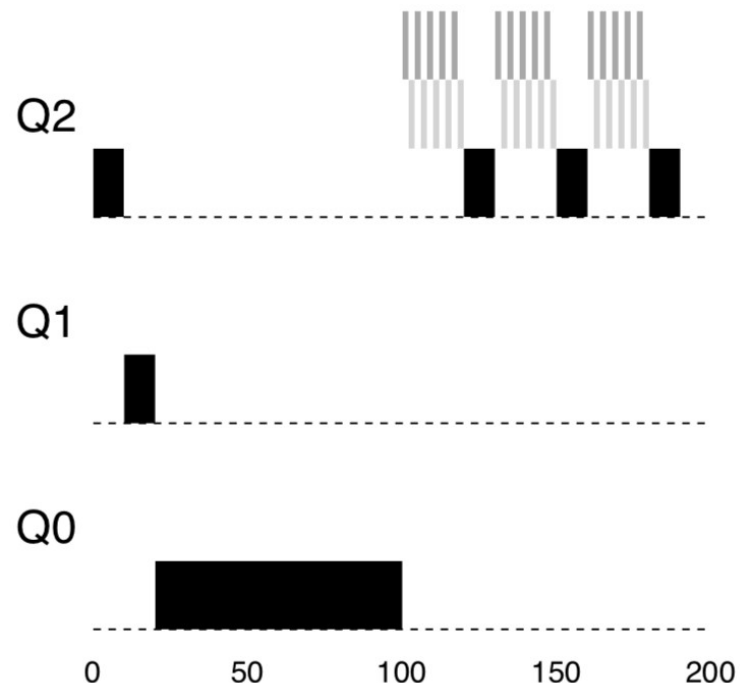
Multi-Level Feedback Queue (MLFQ)

- MLFQ 问题总结

- 1. 饥饿问题
- 2. 用户会针对性修改程序，在时间片结束前进行一次没有意义的 I/O （访问一个不需要的文件），这样可以一直停留在最高优先级
 - Client 和 Server 之间的欺诈、对抗
- 3. 程序 workload 改变模式（计算 -> 交互），没有机会上升优先级

Multi-Level Feedback Queue (MLFQ)

- Attempt #2: The Priority Boost
 - **Rule 5:** After some time period S , move all the jobs in the system to the topmost queue.
 - 解决了两个问题
 - 饥饿
 - Workload 模式改变 (重新做人的机会)



Multi-Level Feedback Queue (MLFQ)

- Attempt #2: The Priority Boost
 - Rule 5 中的 time period S 应该如何设置？
 - 如果过大，没有起到效果，长任务还是可能饥饿；
 - 如果过小，交互式任务所占比重可能太少
 - 需要 black magic 才能做到（与系统中两类任务的比例动态相关）
 - 这类问题在 Systems 领域叫做 voo-doo constants
 - 西非的伏都教，信奉巫术
 - Ousterhout's Law: 在设计算法时尽量避免引入 voo-doo constants
 - John Ousterhout，1988 年和 Fred Douglass 提出第一个日志文件系统
 - 这也是简单的方法往往更流行的原因之一（如 LRU）

Multi-Level Feedback Queue (MLFQ)

- Attempt #2: The Priority Boost
 - 仍然没有解决问题 2（用户欺诈，伪装自己，在让出时间片前做一次无意义的 I/O，以停留在高优先级）



**Job A，欺诈，占用
99% 的 CPU**

Q1



A horizontal bar chart representing queue Q1. It is empty, with no visible bars. Dashed lines indicate the boundaries of the queue area.

Q0



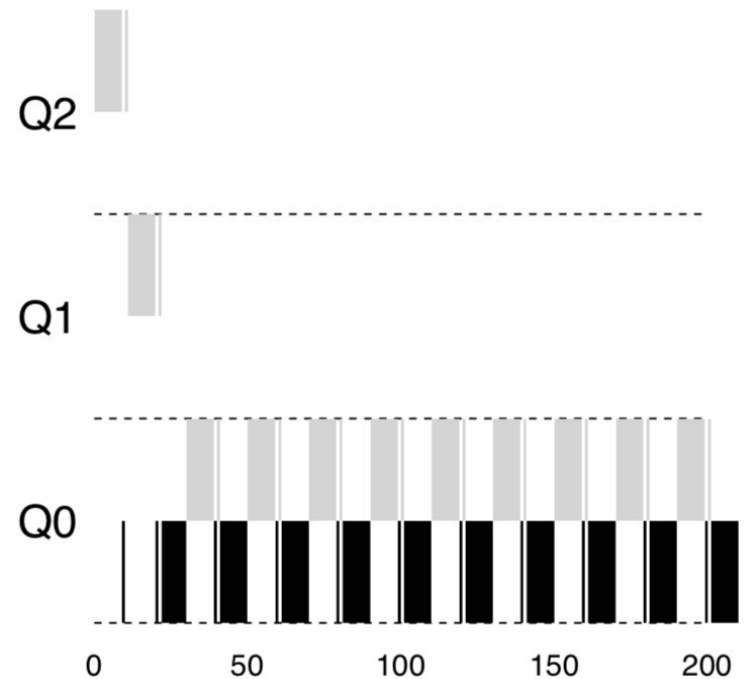
A horizontal bar chart representing queue Q0. It contains 20 vertical bars of equal height, all filled with a light gray color. The bars are closely spaced and extend from the left edge of the chart area.

**Job B，老实，占用 1%
的 CPU**

0 50 100 150 200

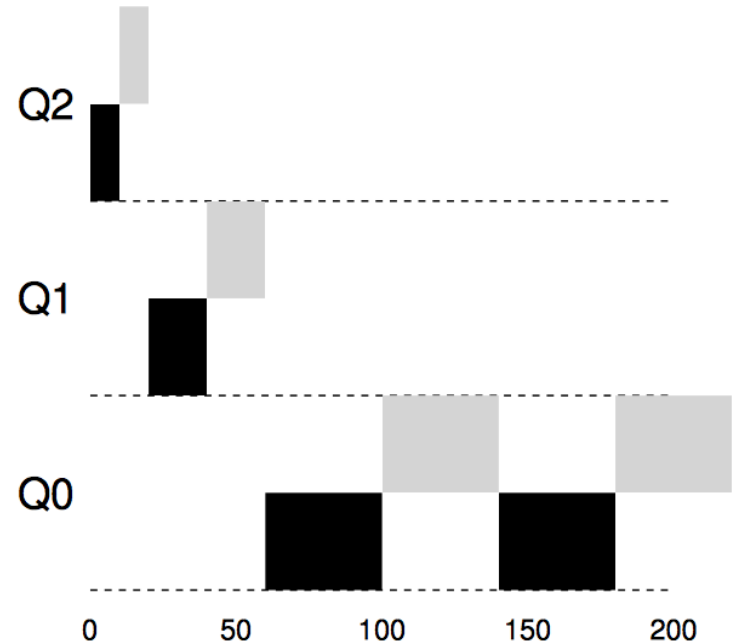
Multi-Level Feedback Queue (MLFQ)

- Attempt #3: Better Accounting
 - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
 - 占用 CPU 总数达到一定量，就下降一个优先级
 - 交互式 Job 占用 CPU 少，所以应该优先极高，能够达到设计目标
 - 比提前让出时间片更合理，不容易被骗



Multi-Level Feedback Queue (MLFQ)

- MLFQ 参数调优
 - Queue 数量、每个 Q 的 time slice 大小、所有任务回到最高级的周期
 - 例如，上级的 Q 用小的 time slice，下级的 Q 用大的 time slice
 - Interactive job: 减小 response time
 - CPU-bound job: 减小 context switch 的开销



Multi-Level Feedback Queue (MLFQ)

- MLFQ 参数调优
 - Solaris MLFQ 实现— Time-Sharing scheduling class (TS)
 - 一系列参数可以调整
 - 例如队列数量默认是 60 个
 - 时间片从 20ms 到几百 ms ，甚至 1s
 - FreeBSD scheduler 采用公式计算任务的优先极

Multi-Level Feedback Queue (MLFQ)

- **MLFQ Summary:**

- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the

复杂系统设计

- Not pursuing the best,
- But avoiding disaster

Outline

- CPU 调度基本策略
- Multi-Level Feedback Queue
- **Proportional Share**
- 多核 CPU 调度

Proportional Share

- Proportional share scheduler 也叫 Fair-share scheduler
- 不追求优化 turnaround time 或 response time ， 而是保障每个任务得到一定比例的 CPU 时间
- 一个典型算法是 lottery scheduling (Waldspurger and Weihl, 1994)

Lottery Scheduling

- 每个任务持有一定的 tickets ，生成随机数，落在哪个任务的范围内，就调度哪个任务
- Tickets 比例决定优先级
 - 例如 A 有 75 个 ticket ， B 有 25 个 ticket

Here is an example output of a lottery scheduler's winning tickets:

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49

Here is the resulting schedule:

A B A A B A A A A A A B A B A A A A A

Lottery Scheduling

- 为什么不使用确定性的调度方法？
 - 如 stride scheduling (1995)
 - 每个任务调度一次后，走一定步长 (stride)，每次选择累计步长 (pass) 最小的任务调度
 - $\text{stride} = C / \text{tickets}$, C 是一个比较大的常数，如 10000
 - 例如 A, B, C 的 tickets 分别为 100, 50, 250 (2:1:5)，则 stride 分别为 100, 200, 40

结果 A 调度 2 次，B 1 次，C 5 次，与预期一致

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Lottery Scheduling

- 为什么不使用确定性的调度方法？
 - 因为 stride scheduling 需要维护 global state (A, B, C 的当前步长)
 - 而 lottery scheduling 不需要维护 global state，对于每个 job 都通过生成随机数调度

Lottery Scheduling

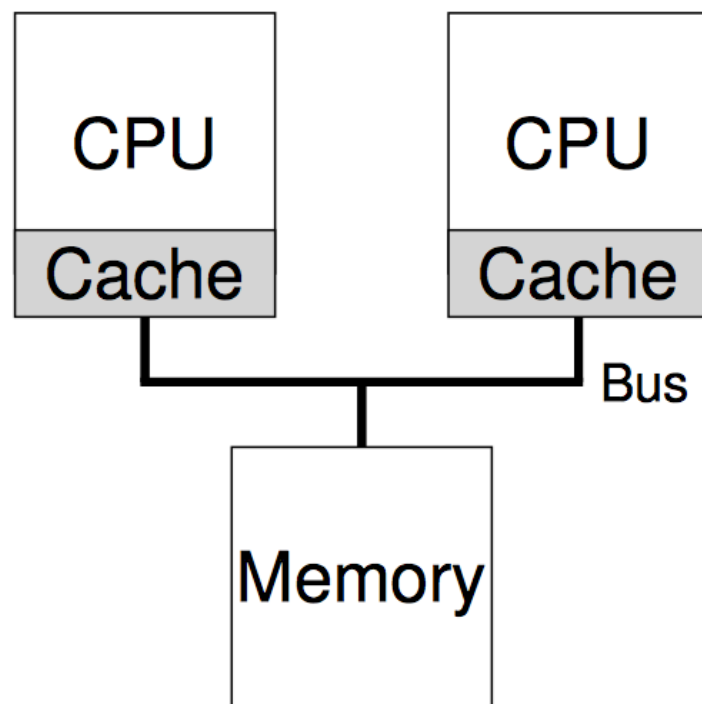
- 随机算法的好处
 - 避免一些特殊情况的问题（例如 LRU 在 cyclic-sequential workloads 下命中率低的问题）
 - 简单，轻量级
 - 性能好，快速
- 深入理解 RANDOM 算法
 - 冷热数据均等几率淘汰
 - 但热数据重新进入概率高很多
 - 看似公平随机，实际更有利于热数据（达到设计目的）

Outline

- CPU 调度基本策略
- Multi-Level Feedback Queue
- Proportional Share
- 多核 **CPU 调度**

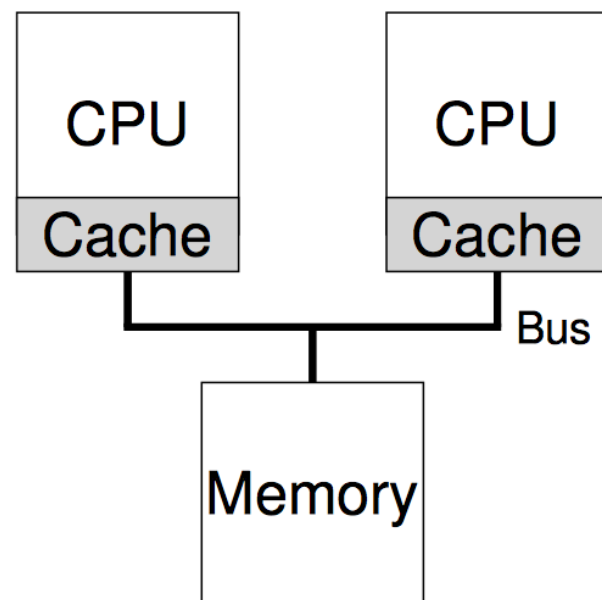
多核 CPU 调度

- 多核 CPU 的新问题：
cache coherence (缓存一致性)
 - 核 1、核 2 都 cache 同一份数据
 - 核 1 修改数据
 - 核 2 读数据会不一致
 - 解决：
 - **bus snooping** (总线监听)，在 bus 监控 cache 行为，做相应处理
 - 增加性能开销



多核 CPU 调度

- bus snooping (总线监听)
 - MESI 方案 (针对 cacheline)
 - **M (modified)**: cl 经过修改, 是系统唯一正确版本, 并且与 memory 不一致 (dirty)
 - **E (exclusive)**: cl 仅在该 CPU cache 中, 并且与 memory 一致
 - **S (share)**: cl 在多个 CPU 的 cache 中, 并且都与 memory 一致
 - **I (invalidate)**: 该 CPU cache 中的 cl 失效



M 状态的 cl 写回到 memory , 变成 E ; E 状态的 cl 被其他 CPU 读取, 变成 S ;

E 状态的 cl 被修改, 变成 M; S 状态下的 cl 被修改, 变成 M , 其他 CPU cache 中的该 cl 变成 I ; CPU 看到 I 状态的 cl , 进行缓存同步

多核 CPU 调度

这世上不存在 lock-free (无锁) , 只有 optimistic lock (乐观锁) 和 pessimistic Lock (悲观锁)

- 多线程共享数据的问题
 - 加锁, 但会带来性能开销
 - 线程可能被调度到的核越多, 冲突越大, 性能就越差

```
1  typedef struct __Node_t {
2      int          value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;          // remember old head ...
8      int value   = head->value;    // ... and its value
9      head        = head->next;     // advance head to next pointer
10     free(tmp);                  // free old head
11     return value;               // return value at head
12 }
```

不加锁的话, 两个线程同时拿到 head , 都做删除, 会 **double free**

多核 CPU 调度

- Cache Affinity (密切关系, 亲近)
 - 一个进程连续调度到同一个 CPU 核上, cache 和 TLB 都有效, 性能会比较好
 - 如果调度到不同的核上, 则 cache 资源浪费

多核 CPU 调度

- **Single-Queue Multiprocessor Scheduling or SQMS**
 - 沿用单 **CPU** 的调度策略，只是增大队列
 - 哪个核闲置，就从队列中调度一个任务来执行
 - 优点： **Load balance**
 - 缺点： **LOCK** 开销大； **cache affinity** 差，性能差

CPU 0	A	E	D	C	B	... (repeat) ...
CPU 1	B	A	E	D	C	... (repeat) ...
CPU 2	C	B	A	E	D	... (repeat) ...
CPU 3	D	C	B	A	E	... (repeat) ...

多核 CPU 调度

- **Single-Queue Multiprocessor Scheduling or SQMS**

- 改进:

- 尽量调度到一个 CPU 核上
 - 少数任务切换

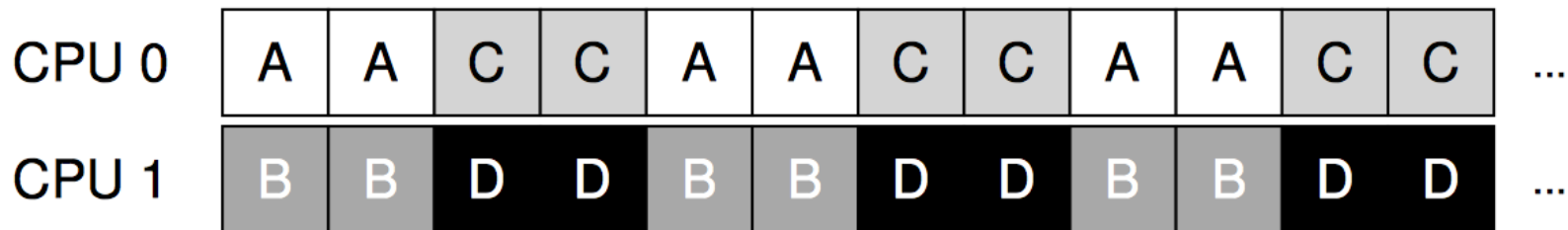
- 扩展性不好

CPU 0	A	E	A	A	A	... (repeat) ...
CPU 1	B	B	E	B	B	... (repeat) ...
CPU 2	C	C	C	E	C	... (repeat) ...
CPU 3	D	D	D	D	E	... (repeat) ...

多核 CPU 调度

- **Multi-Queue Multiprocessor Scheduling (or MQMS)**

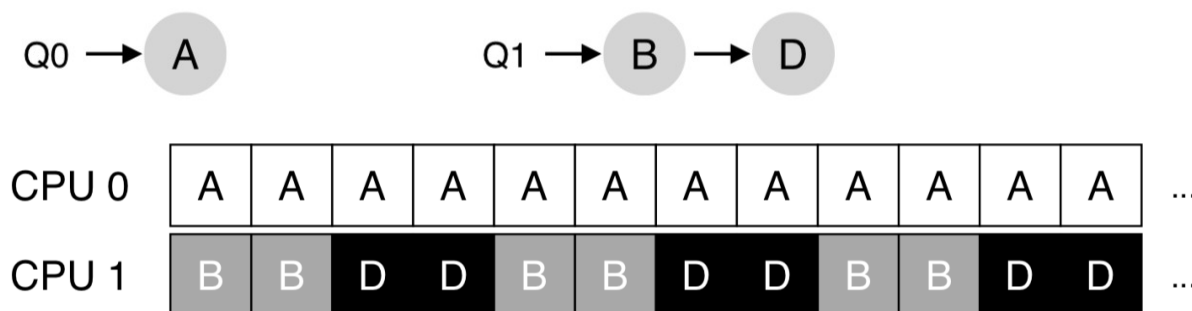
- 每个 CPU 核一个独立的队列
- 优点：性能好（Lock 冲突少，cache affinity 好）
- 缺点：load balance



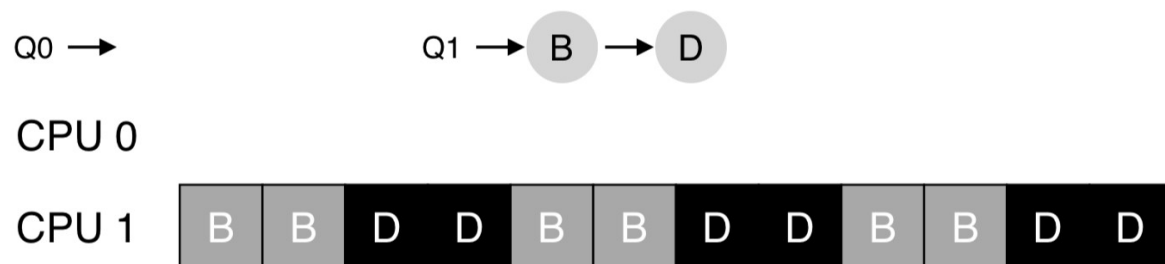
多核 CPU 调度

- **Multi-Queue Multiprocessor Scheduling (or MQMS)**

- 主要缺点: load imbalance



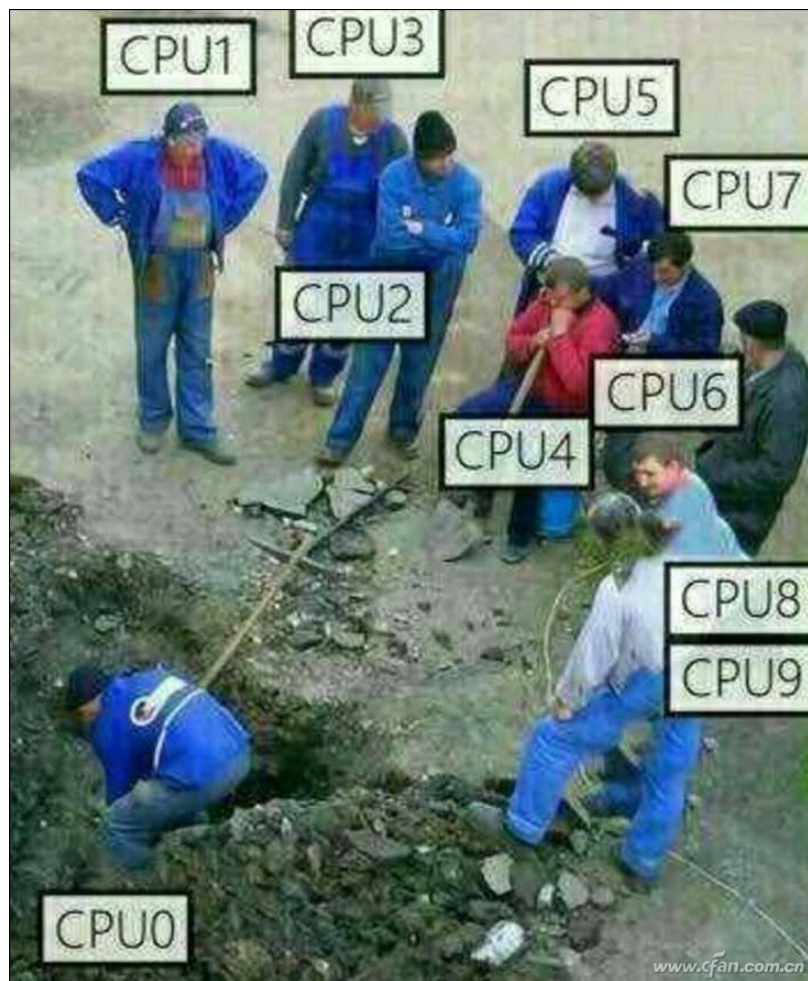
Job C 执行完了，此后 A 的 CPU 份额是 B,D 的两倍



Job A 也执行完了，CPU 0 完全闲置

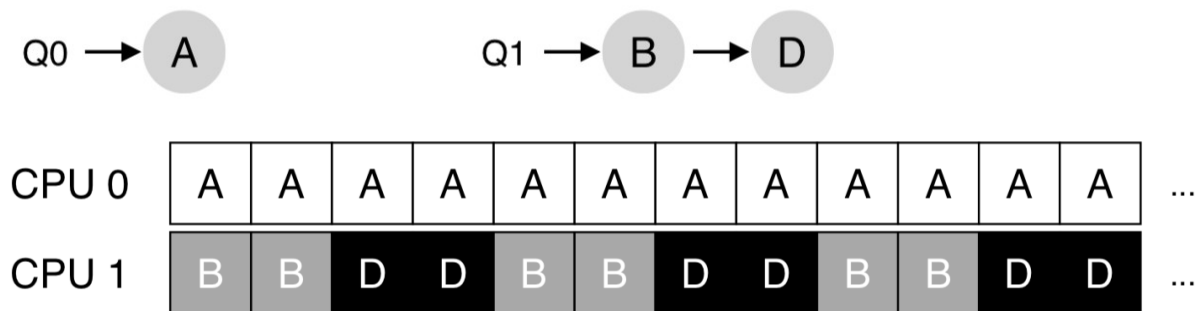
多核 CPU 调度

— 一核有难，九核围观

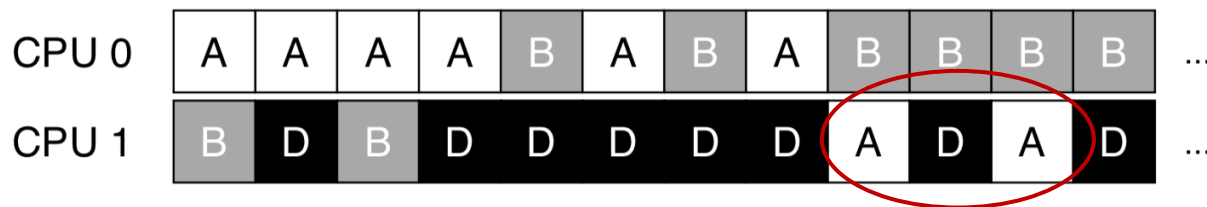


多核 CPU 调度

- 解决 load imbalance 的办法： job migration
 - 但这种情况下，单独迁移一个 job 并不解决问题

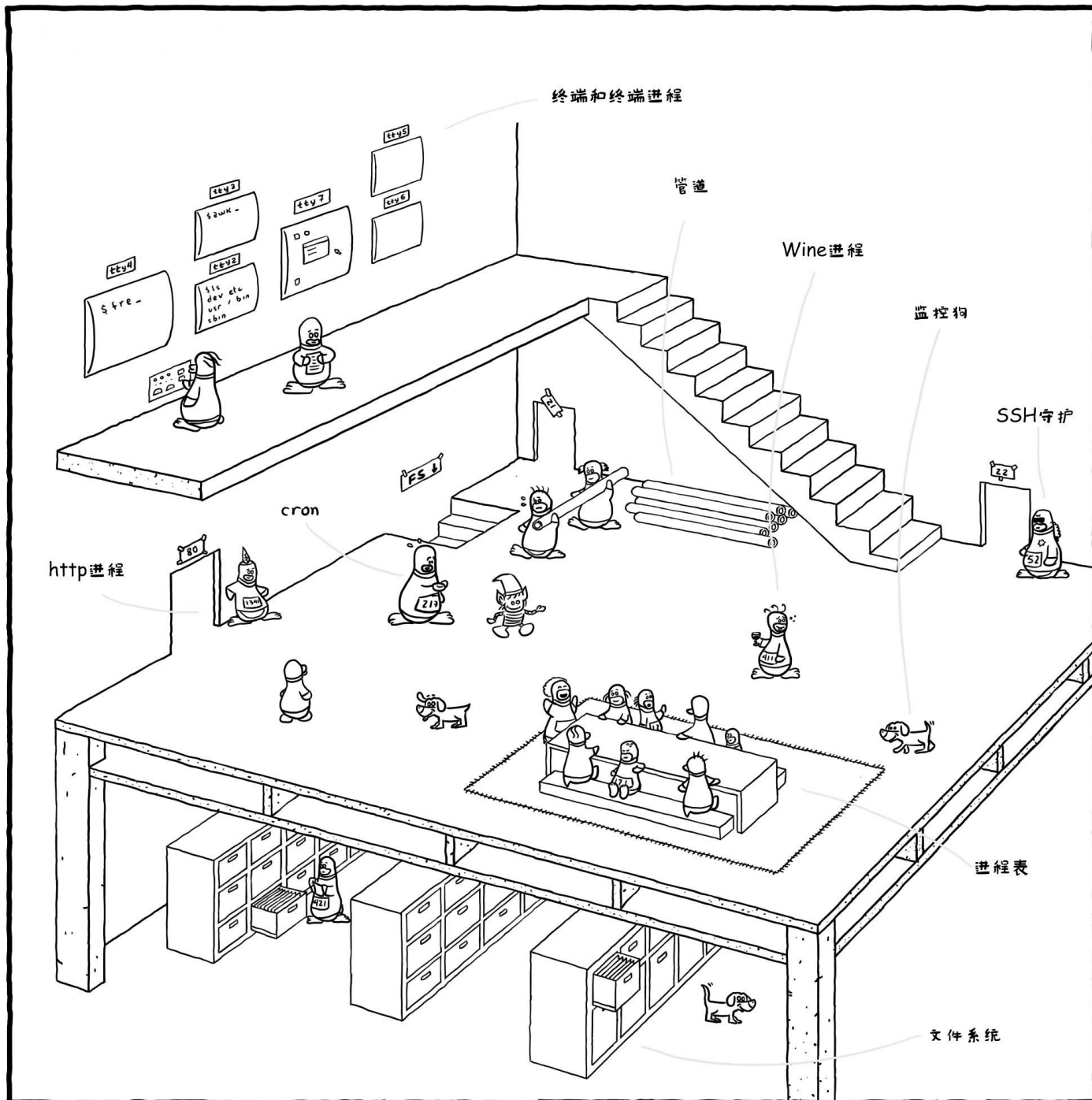


- 更灵活的 job migration： Work stealing
 - 不饱和队列从其他队列拿过来一两个 job；但频率太高也会有问题



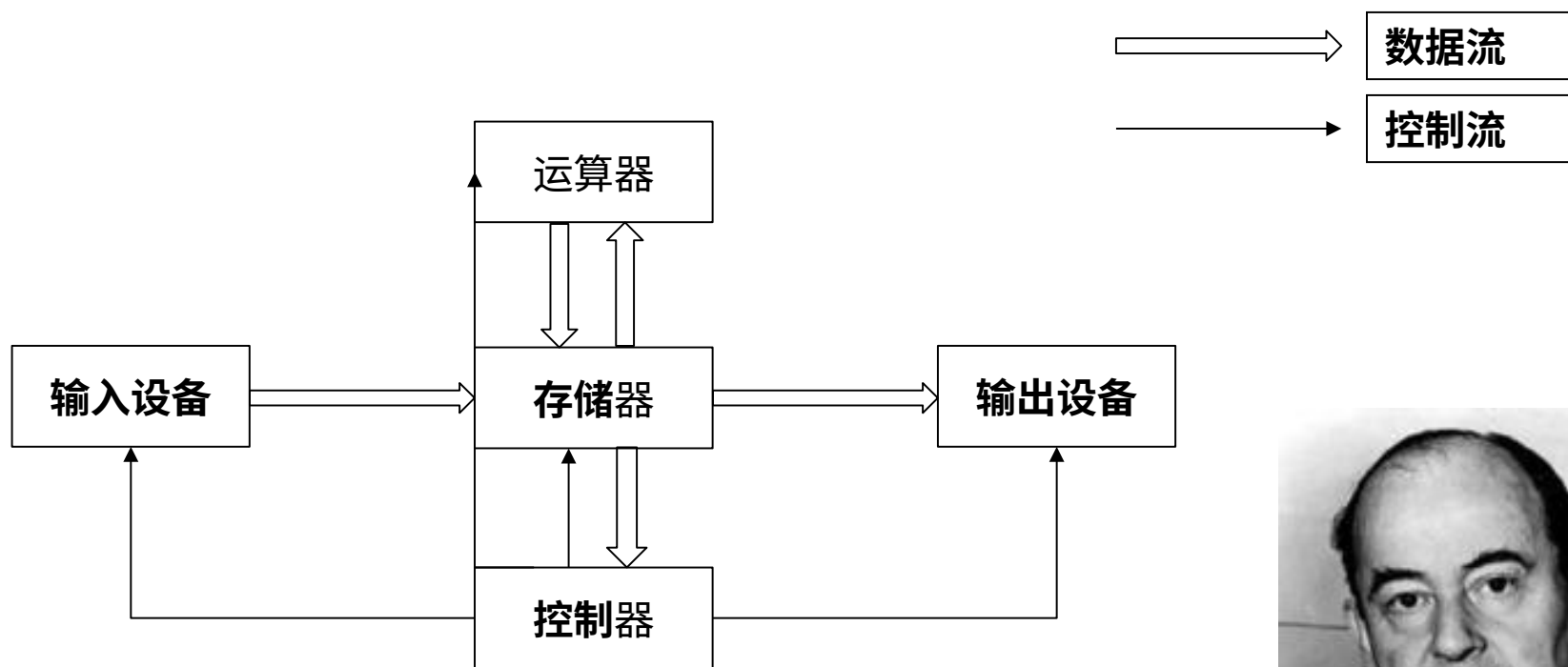
多核 CPU 调度

- Linux 的多核 CPU 调度策略
 - 没有一种确定的方案完全优于其他
 - CFS （ Completely Fair Scheduler ）： 多队列
 - 类似于 stride scheduler
 - BFS （ Brain-Fuck Scheduler ）： 单队列
 - O(1) Scheduler： 多队列
 - 类似于 MLFQ



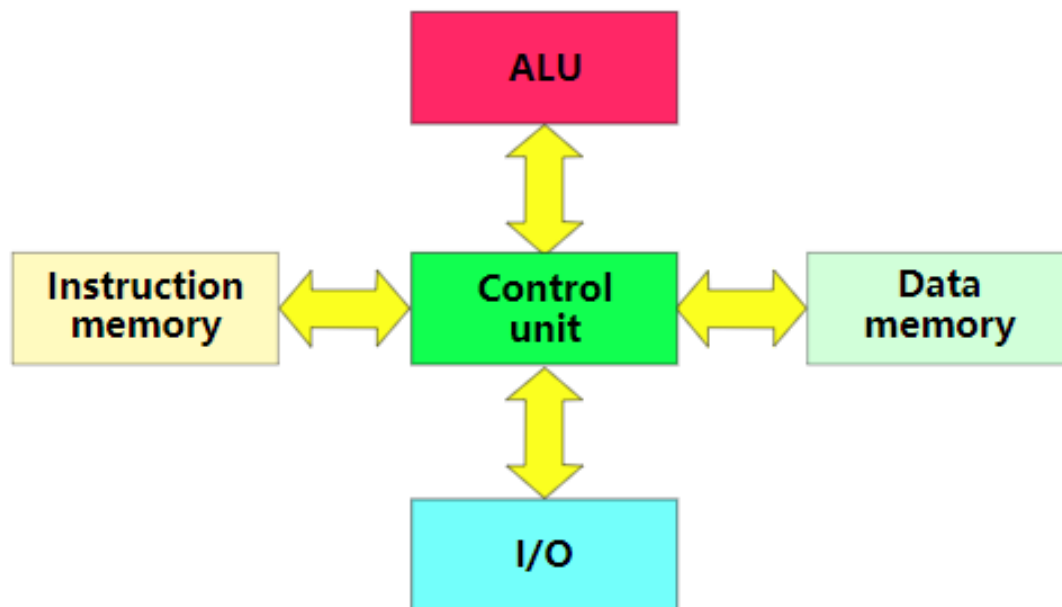
国产 CPU

冯诺伊曼结构



计算机之父——约翰·冯·诺依曼

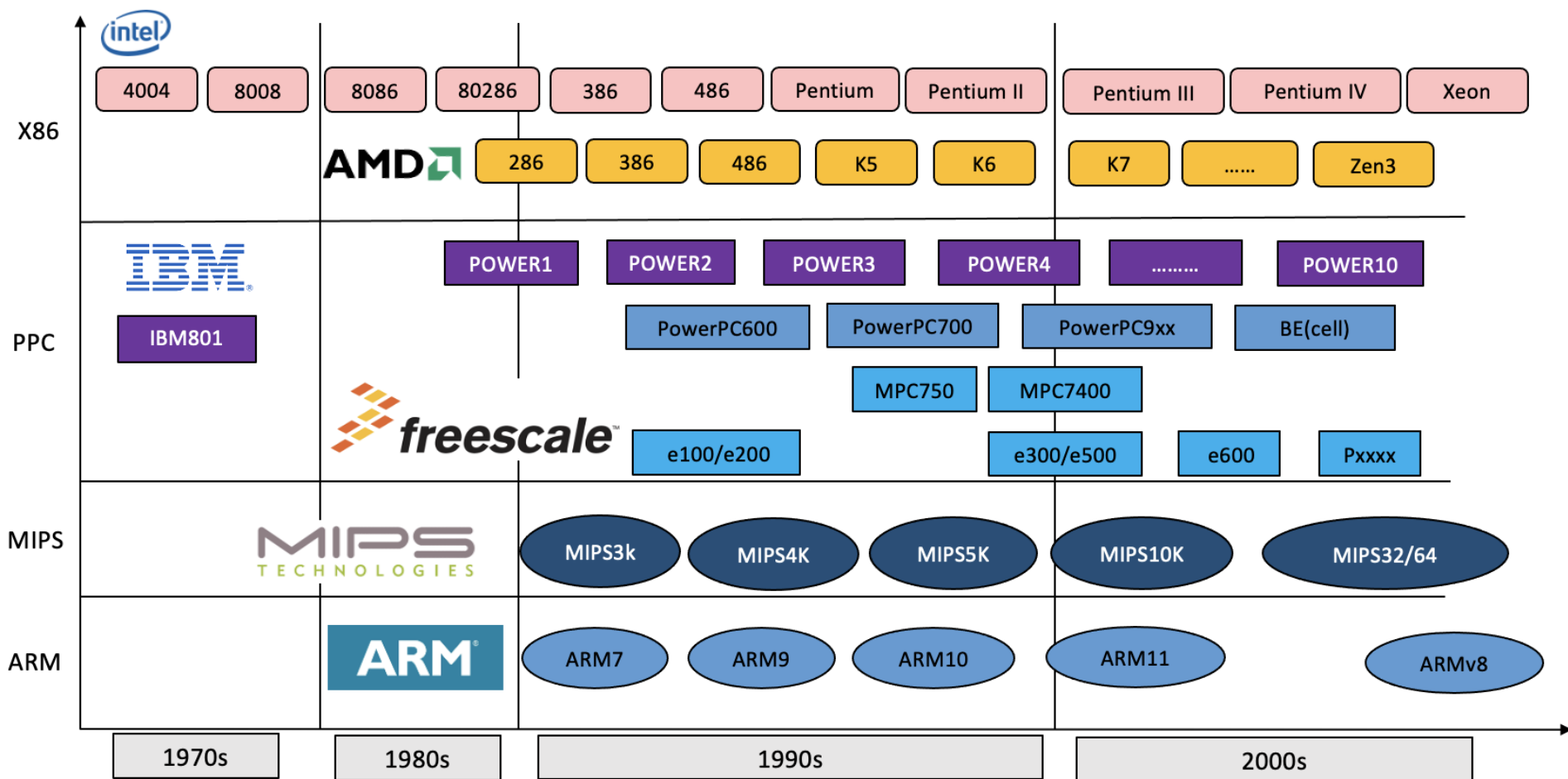
哈佛结构



哈佛结构是一种将程序指令存储和数据存储分开的存储器结构，它的主要特点是将程序和数据存储在不同的存储空间中
即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问，目的是为了减轻程序运行时的访存瓶颈。

哈佛架构的中央处理器典型代表 **ARM9/10** 及后续 **ARMv8** 的处理器，例如：华为鲲鹏 920 处理器。

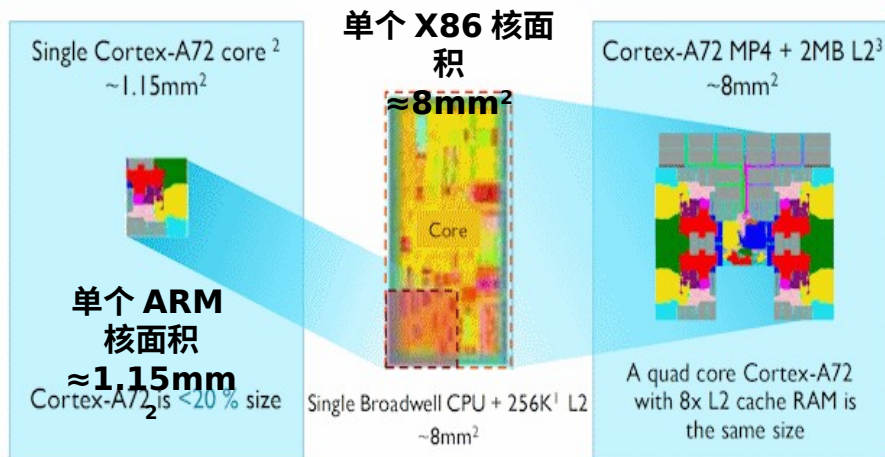
主流 CPU 发展路径



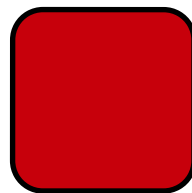
ARM 提供更多计算核心

- 工艺、主频遇到瓶颈后，开始通过增加核数的方式来提升性能；
- 芯片的物理尺寸有限制，不能无限制的增加；
- ARM 的众核横向扩展空间优势明显。

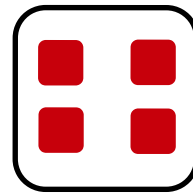
Cortex-A72: Ideal for dense compute environments



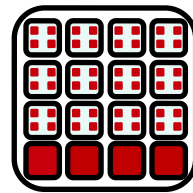
单核



多核



众核



ARM 公司授权体系

- ARM 目前在全球拥有大约 1000 个授权合作商、320 家伙伴，但是购买架构授权的厂家不超过 20 家，中国有华为、飞腾获得了架构授权。

架构 / 指令集授权

- 按照所授权的架构和指令集（如 **ARMv8**）自行编写代码、设计芯片。

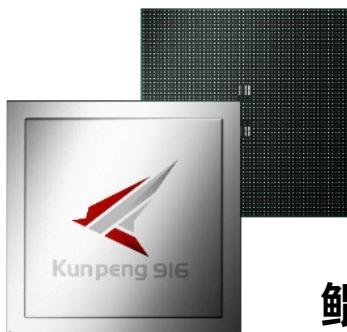
处理器授权

- 提供 **RTL** 代码，处理器的核数、缓存可以自己配置。自主设计主频、工艺、代工厂等。

处理器优化包 / 物理 IP 包授权 (POP)

- 只能按照 **ARM** 设计好的处理器类型、在指定的代工厂进行生产。

鲲鹏处理器



鲲鹏 916

支持多路互联的 ARM 处理器

- 32 核，2.4 GHz 主频
- SPECint 性能匹配业界中端，功耗低至 75 W
- 支持 4 通道 DDR4 控制器
- 支持 PCI-e 3.0 和 SAS/SATA 3.0
- 集成板载 GE/10 GE 网络
- 支持 2 路互联



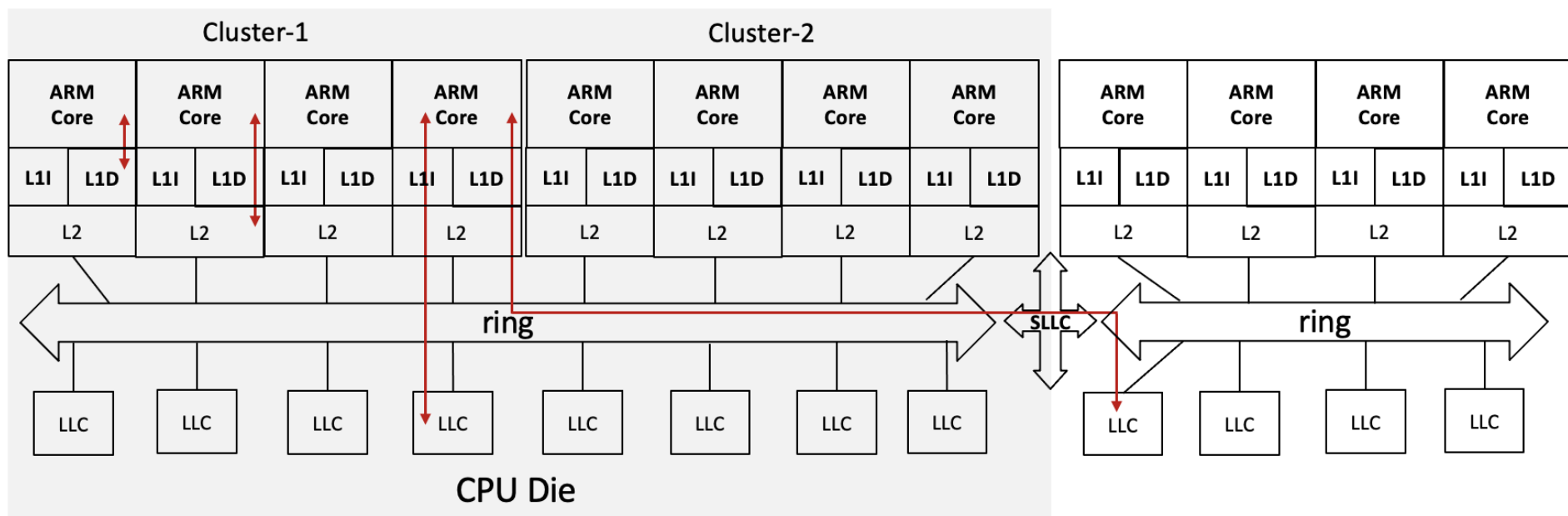
鲲鹏 920

7nm 制程，数据中心 ARM 处

- 计算核数提升 1 倍，~~最理~~ 64 核
- SPECint 性能提升超过 2 倍
- 内存通道数提升 1 倍，支持 8 通道 DDR4 控制器
- 支持 PCI-e 4.0 和 CCIX
- 集成板载 100 GE 网络和加密、压缩加速引擎
- 支持 2 路或 4 路互联

鲲鹏 920 系列芯片架构——乐高架构

- TaiShan Core 独享 L1 Cache 和 L2 Cache，4 个 Core 和 1 个 L3 Cache tag 组成一个 Cluster，6~8 个 Cluster 组成一个 CPU Die，合封后的两个 CPU Die 共



OpenEuler 的 CPU 调度策略

- CFS(Completely Fair Scheduler) 调度
 - 每个进程一个队列
 - 结合时间片和优先级，引入虚拟运行时间
 - 按照当前系统负载和普通进程的优先级给进程分配 CPU 使用时间的比例，确保相对公平

homework3
