第 12 讲：多处理器调度

第一节：对称多处理与多核架构

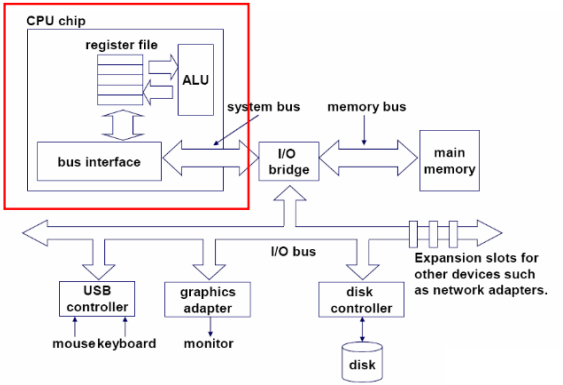
向勇、陈渝

清华大学计算机系

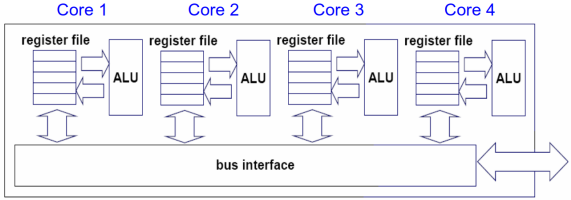
xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

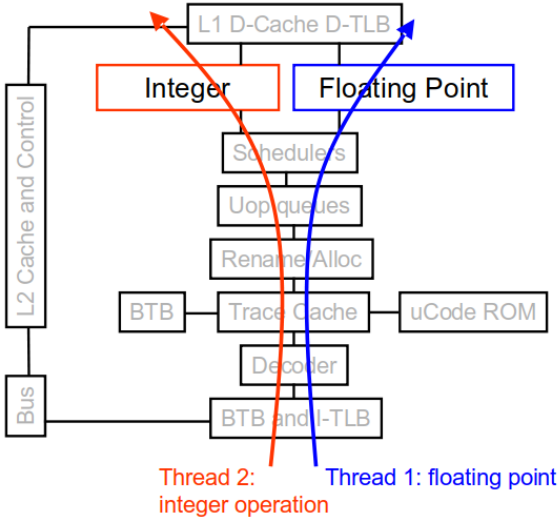
单核处理器


CPU chip
register file
ALU
system bus
memory bus
bus interface
I/O bridge
main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
graphics adapter
disk controller
mouse
keyboard
monitor
disk

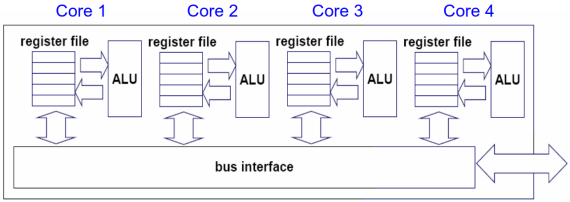
多核处理器


Core 1
Core 2
Core 3
Core 4
register file
ALU
bus interface

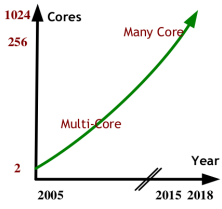
Large SMT(Simultaneous multithreading 或 hyper-thread，超线程)

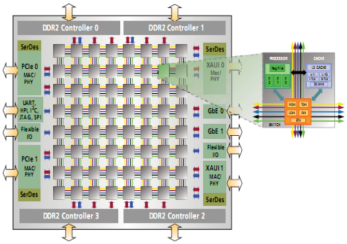


多核处理器

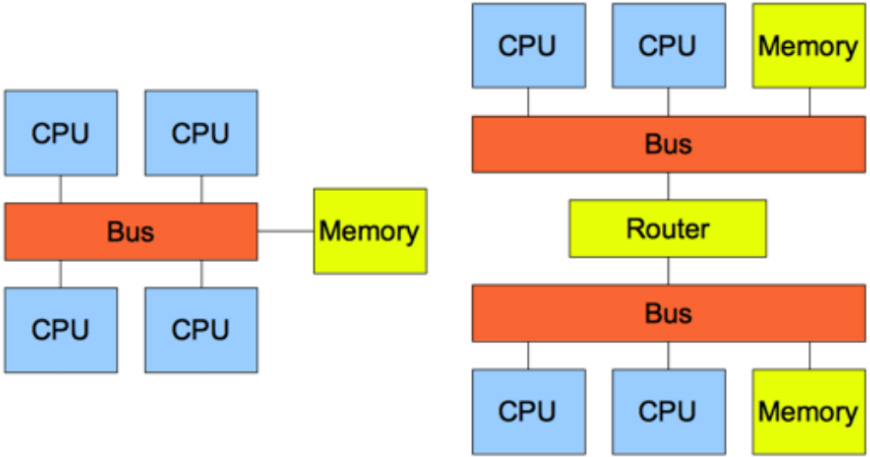

Core 1
Core 2
Core 3
Core 4
register file
ALU
bus interface

众核处理器

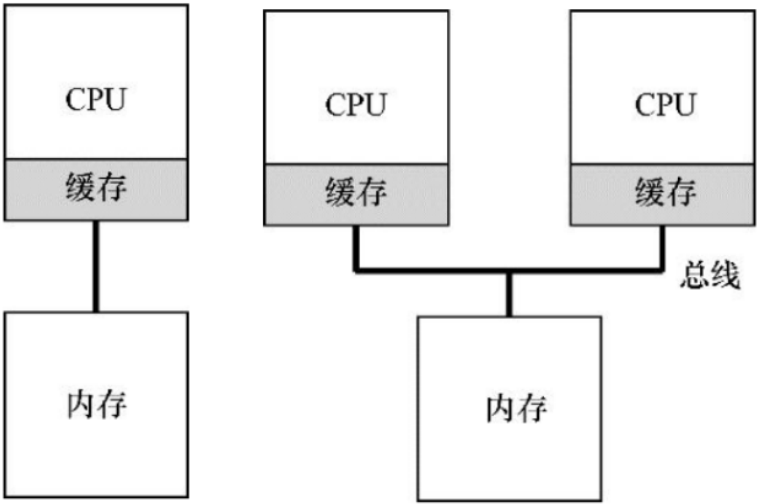

1024
256
2
Cores
Many Core
Multi-Core
Year
2005
2015 2018


DDR2 Controller 0
DDR2 Controller 1
DDR2 Controller 3
DDR2 Controller 2
SerDes
PCIe 0 MAC PHY
UART, MII, I²C, JTAG, SPI
Flexible IO
PCIe 1 MAC PHY
XAUI 0 Mac PHY
GbE 0
GbE 1
XAUI 1 MAC PHY

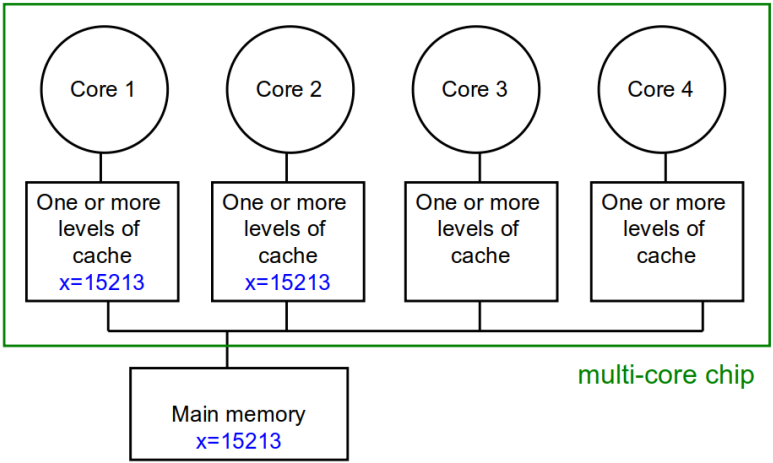
Large SMP（对称多处理系统）与 NUMA (非一致内存访问系统)



Cache 一致性 (Cache Coherence)



Cache 一致性问题


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=15213
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory
x=15213

第 12 讲：多处理器调度

第二节：多处理器调度概述

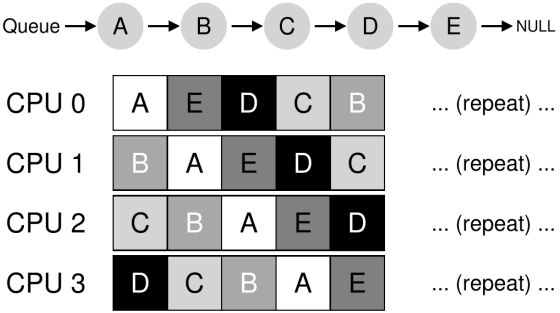
向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

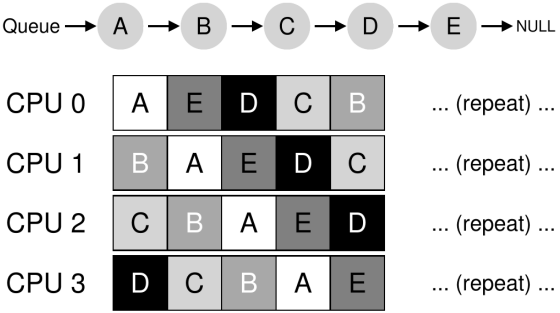
2020 年 5 月 5 日

单队列调度



最基本的方式是简单地复用单处理器调度的基本架构，将所有需要调度的进程放入一个单独的队列中，我们称之为单队列多处理器调度 (Single Queue Multiprocessor Scheduling，SQMS)。

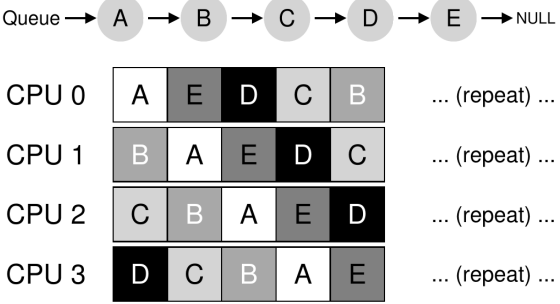
单队列调度



单队列多处理器调度 (SQMS)

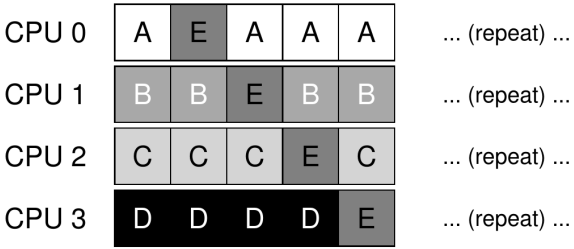
- 缺乏可扩展性 (scalability)
- 缓存亲和性 (cache affinity) 弱

单队列调度

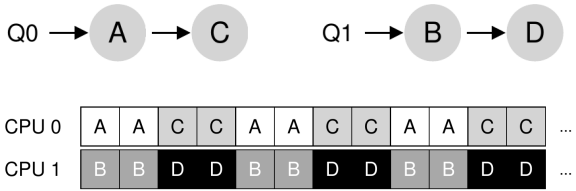


单队列多处理器调度 (SQMS)

尽可能让进程在同一个 CPU 上运行。保持一些进程的亲和度的同时，可能需要牺牲其他进程的亲和度来实现负载均衡。



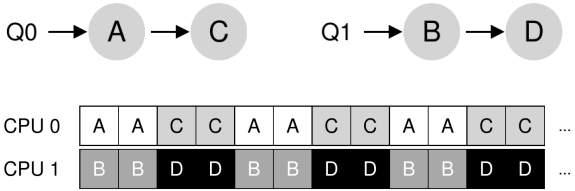
多队列调度



多队列多处理器调度 (Multi-Queue Multiprocessor Scheduling，MQMS)

- 在 MQMS 中，基本调度框架包含多个调度队列，每个队列可以使用不同的调度规则，比如轮转或其他任何可能的算法。
- 当一个进程进入系统后，系统会依照一些启发性规则 (如随机或选择较空的队列) 将其放入某个调度队列。这样一来，每个 CPU 调度之间相互独立，就避免了单队列的方式中由于数据共享及同步带来的问题。

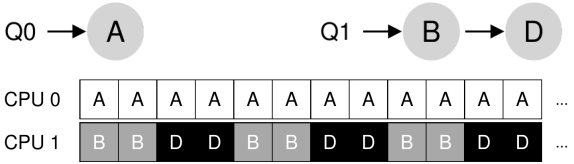
多队列调度



多队列多处理器调度 (MQMS)

- 根据不同队列的调度策略，每个 CPU 从两个进程中选择，决定谁将运行。例如，轮转调度
- 具有可扩展性。队列的数量会随着 CPU 的增加而增加，因此锁和缓存争用的开销不是大问题。
- 具有良好的缓存亲和度。所有进程都保持在固定的 CPU 上，因而可以很好地利用缓存数据。

多队列调度



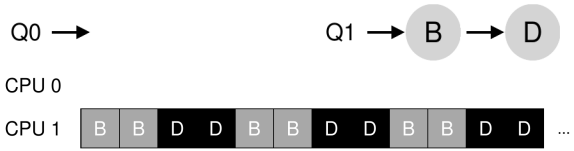
多队列多处理器调度 (MQMS)

负载不均 (load imbalance)

假定和上面设定一样 (4 个进程，2 个 CPU)，但假设一个进程 (如 C) 这时执行完毕。如果对系统中每个队列都执行轮转调度策略：

- A 获得了 B 和 D 两倍的 CPU 时间

多队列调度



多队列多处理器调度 (MQMS)

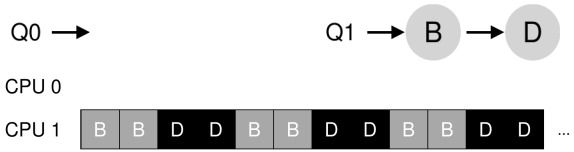
负载不均 (load imbalance)

假定和上面设定一样 (4 个进程，2 个 CPU)，但假设 A 和 C 都执行完毕，系统中只有 B 和 D。如果对系统中每个队列都执行轮转调度策略：

- CPU1 很忙
- CPU0 空闲

怎样才能克服潜伏的负载不均问题?

多队列调度



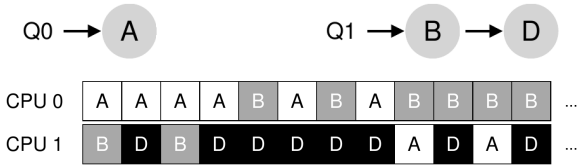
关键问题：如何应对负载不均

多队列多处理器调度程序应该如何处理负载不均问题，从而更好地实现预期的调度目标?

最明显的答案是让进程移动，这种技术我们称为迁移 (migration)。通过进程的跨 CPU 迁移，可以真正实现负载均衡。

- 情况：有一个 CPU 空闲，另一个 CPU 有一些进程。
- 迁移：将 B 或 D 迁移到 CPU0。

多队列调度



关键问题：如何应对负载不均

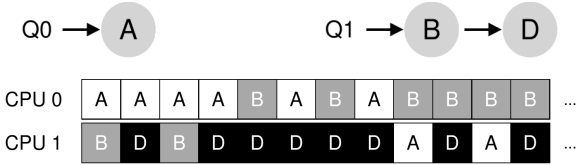
多队列多处理器调度程序应该如何处理负载不均问题，从而更好地实现预期的调度目标?

最明显的答案是让进程移动，这种技术我们称为迁移 (migration)。通过进程的跨 CPU 迁移，可以真正实现负载均衡。

- 情况：A 独自留在 CPU 0 上，B 和 D 在 CPU 1 上交替运行
- 迁移：不断地迁移和切换一个或多个进程

系统如何决定发起这样的迁移?

多队列调度



- 进程量较少的 (源) 队列不定期地“偷看”其他 (目标) 队列是不是比自己的进程多
- 如果目标队列比源队列 (显著地) 更满，就从目标队列“窃取”一个或多个进程，实现负载均衡。

关键问题：系统如何决定发起这样的迁移?

一个基本的方法是采用一种技术，名为工作窃取 (work stealing)

- 如果太频繁地检查其他队列，就会带来较高的开销，可扩展性不好
- 如果检查间隔太长，又可能会带来严重的负载不均

第 12 讲：多处理器调度

第三节：O(1) 调度

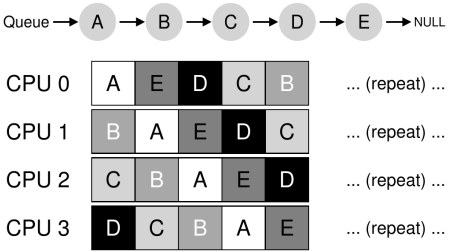
向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

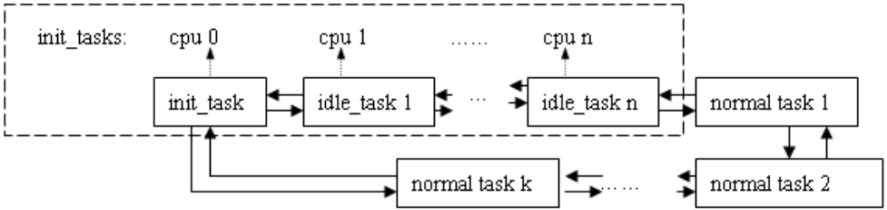
SMP 和 Linux 内核



- 在 Linux 2.0 的早期，SMP 支持由一个“大锁”组成，这个“大锁”对操作系统内部的访问进行串行化
- 在 2.2 前的内核中，SMP 实现在用户级，Linux 内核本身并不能因为有多个处理器而得到加速

SMP 和 Linux 内核

- 在 2.4 内核后，SMP 实现在核心级，使用多处理器可以加快内核的处理速度。一开始的调度器是复杂度为 O(n) 的始调度算法
  - 2.4 scheduler 维护两个 queue：runqueue 和 expired queue
  - 两个 queue 都永远保持有序
  - 一个 process 用完时间片，就会被插入 expired queue
  - 当 runqueue 为空时，只需要把 runqueue 和 expired queue 交换一下即可

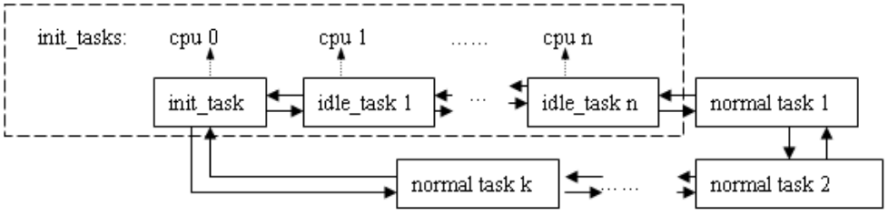


图：进程管理相关的数据结构示意图

（注：新进程总是添加到 init_task 的左端，即 prev 端，如图）

SMP 和 Linux 内核

- 在 2.4 内核后，SMP 实现在核心级，使用多处理器可以加快内核的处理速度。一开始的调度器是复杂度为 O(n) 的始调度算法
  - 全局共享的就绪队列
  - 寻找下一个可执行的 process，这个操作一般都是 O(1)
  - 每次进程用完时间片，执行插入操作是，实际上会遍历所有任务，复杂度为 O(n)

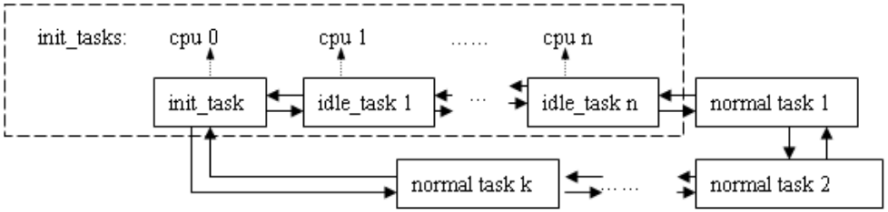


图：进程管理相关的数据结构示意图

（注：新进程总是添加到 init_task 的左端，即 prev 端，如图）

SMP 和 Linux 内核

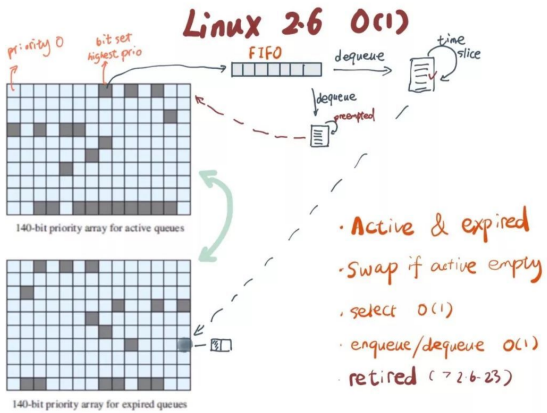
- 在 2.4 内核后，SMP 实现在核心级，使用多处理器可以加快内核的处理速度。一开始的调度器是复杂度为 O(n) 的始调度算法
  - 现代操作系统都能运行成千上万个进程
  - O(n) 算法意味着每次调度时，对于当前执行完的 process，需要把所有在 expired queue 中的 process 过一遍，找到合适的位置插入
  - 这不仅仅会带来性能上的巨大损失，还使得系统的调度时间非常不确定——根据系统的负载，可能有数倍甚至数百倍的差异



图： 进程管理相关的数据结构示意图

（注：新进程总是添加到 init_task 的左端，即 prev 端，如图）

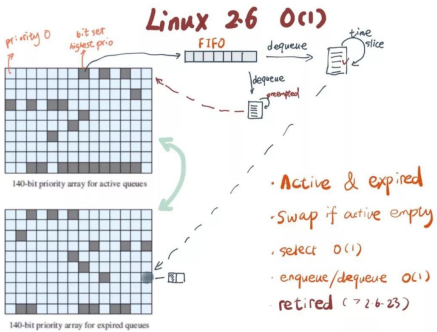
O(1) 调度器



O(1) 调度器

2.6 版本的调度器是由 Ingo Molnar 设计并实现的。Ingo 从 1995 年开始就一直参与 Linux 内核的开发。他编写这个新调度器的动机是为唤醒、上下文切换和定时器中断开销建立一个完全 O(1) 的调度器

O(1) 调度器

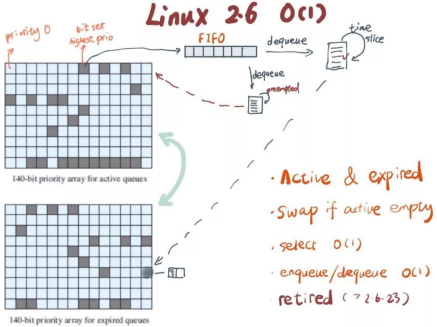


满足 O(1) 的数据结构?

回顾一下数据结构的四种基本操作和时间复杂度

- access：随机访问
  - array: 平均情况和最坏情况均能达到 O(1)
  - linked list 是 O(N)
  - tree 一般是 O(log N)

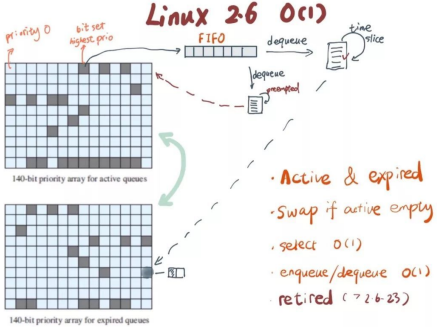
O(1) 调度器



满足 O(1) 的数据结构?
回顾一下数据结构的四种基本操作和时间复杂度

- search：搜索
  - hash table 时间复杂度是 O(1)，但它最坏情况下是 O(N)
  - 大部分 tree（b-tree / red-black tree）平均和最坏情况都是 O(log N)

O(1) 调度器

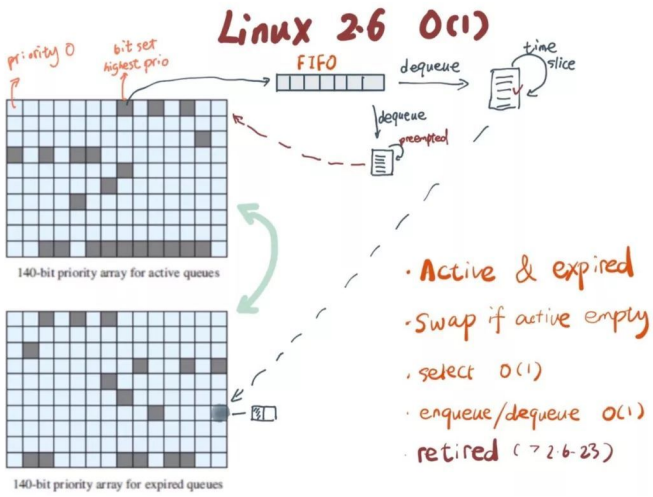


满足 O(1) 的数据结构?

回顾一下数据结构的四种基本操作和时间复杂度

- insert/deletion：插入和删除
  - hash table 时间复杂度是 O(1)，但它最坏情况下是 O(N)
  - linked list，stack，queue 在平均和最坏情况下都是 O(1)

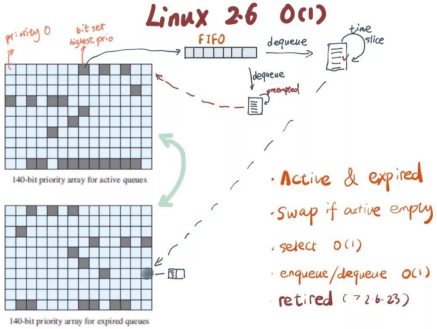
O(1) 调度器



O(1) 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)
- 每个优先级下面用一个 FIFO queue 管理这个优先级下的 process。新来的插到队尾，先进先出，insert / deletion 都是 O(1)

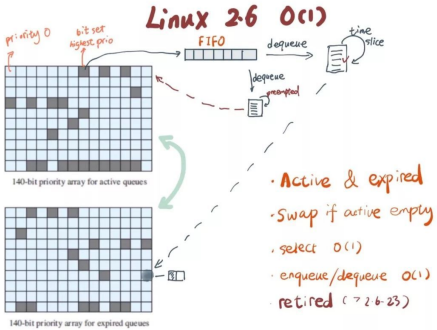
O(1) 调度器



O(1) 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)
  - bitarray，它为每种优先级分配一个 bit，如果这个优先级队列下面有 process，那么就对相应的 bit 染色，置为 1，否则置为 0。
  - 问题就简化成寻找一个 bitarray 里面最高位是 1 的 bit（left-most bit），这基本上是一条 CPU 指令的事。

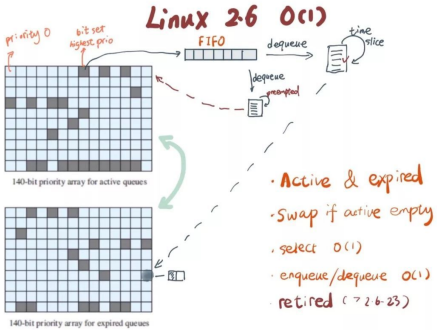
O(1) 调度器



O(1) 调度器

- 在 active bitarray(APA) 中寻找 left-most bit 的位置 x。
- 在 APA 中找到对应队列 APA[x]。
- 从 APA[x] 中 dequeue 一个 process。
- 对于当前执行完的 process，重新计算其 priority，然后 enqueue 到 expired priority array(EPA) 相应的队里 EPA[priority]。
- 如果 priority 在 expired bitarray 里对应的 bit 为 0，将其置 1。
- 如果 active bitarray 全为零，将 active bitarray 和 expired bitarray 交换一下。

O(1) 调度器



O(1) 调度器：多核/SMP 支持

- 在一定时间间隔后，进行 load balance 分析
- rq> cpu_load : represents load on the CPU
- 在每个时钟中断后进行计算
- current_load = rq>nr_running * SCHED_LOAD_SCALE;
- Pulling 进程而不是 pushing 进程

第 12 讲：多处理器调度

第四节：CFS 调度

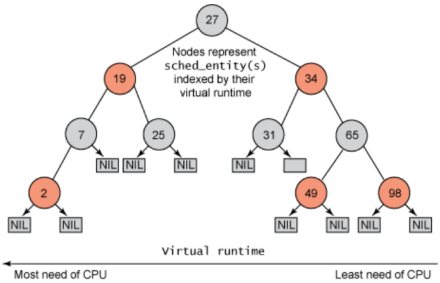
向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

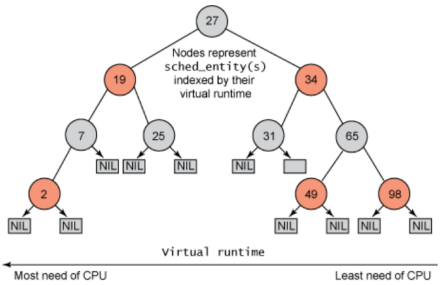
2020 年 5 月 5 日

CFS 调度



- 以前的 Linux 调度算法根据进程的优先级进行调度，即通过一系列运行指标确定进程的优先级，然后根据进程的优先级确定调度哪个进程
- CFS 则转换了一种思路，它不计算优先级，而是通过计算进程消耗的 CPU 时间（标准化以后的虚拟 CPU 时间）来确定谁来调度。从而到达所谓的公平性。

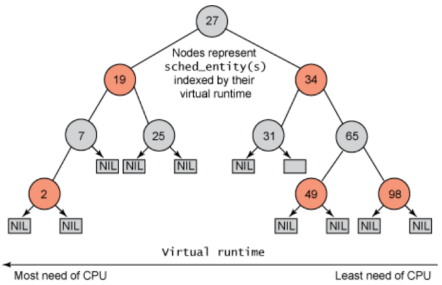
CFS 调度



绝对公平性：

- 把 CPU 当做一种资源，并记录下每一个进程对该资源使用的情况，在调度时，调度器总是选择消耗资源最少的进程来运行。
- 但这种绝对的公平有时也是一种不公平，因为有些进程的工作比其他进程更重要，我们希望能按照权重来分配 CPU 资源。

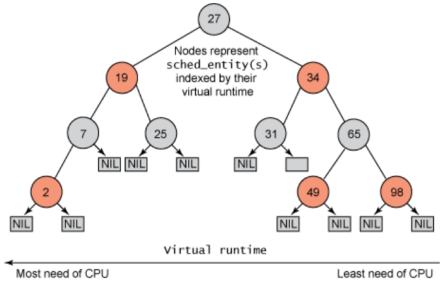
CFS 调度



相对公平性：

- 为了区别不同优先级的进程，就是会根据各个进程的权重分配运行时间
- 分配给进程的运行时间 = 调度周期 * 进程权重 / 所有进程权重之和
- 调度周期：将所处于 TASK_RUNNING 态进程都调度一遍的时间

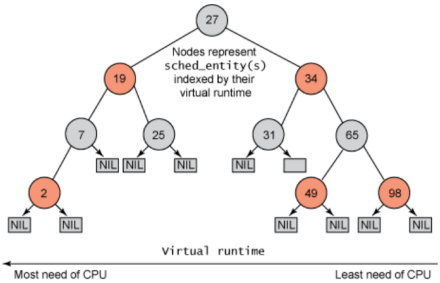
CFS 调度



相对公平性：

- 比如系统中只两个进程 A, B，权重分别为 1 和 2，假设调度周期设为 30ms，
- A 的 CPU 时间为:30ms * $(1/(1+2)) = 10$ms
- B 的 CPU 时间为：30ms * $(2/(1+2)) = 20$ms
- 在这 30ms 中 A 将运行 10ms，B 将运行 20ms

CFS 调度

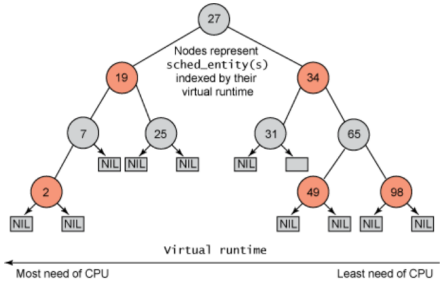


实现原理

Linux 通过引入 virtual runtime(vruntime)

- vruntime = 实际运行时间 * 1024 / 进程权重
- 谁的 vruntime 值较小就说明它以前占用 cpu 的时间较短，受到了“不公平”对待，因此下一个运行进程就是它
- 这样既能公平选择进程，又能保证高优先级进程获得较多的运行时间。

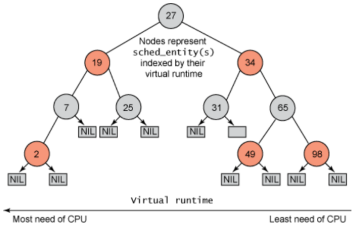
CFS 调度



具体实现

- Linux 采用了一颗红黑树（对于多核调度，实际上每一个核有一个自己的红黑树），记录下每一个进程的 vruntime
- 需要调度时，从红黑树中选取一个 vruntime 最小的进程出来运行

CFS 调度

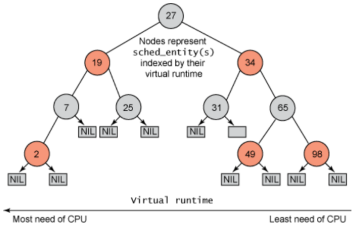


具体实现：权重如何决定？

- 权重由 nice 值确定，权重跟进程 nice 值之间有一一对应的关系
- 通过全局数组 prio_to_weight 来转换，nice 值越大，权重越低。

```
nice值共有40个，与权重之间，每一个nice值相差10%左右。
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```

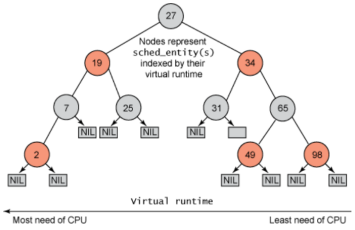
CFS 调度



具体实现：新创建进程的 vruntime 是多少？

- 假如新进程的 vruntime 初值为 0 的话，比老进程的值小很多，那么它在相当长的时间内都会保持抢占 CPU 的优势，老进程就要饿死了，这显然是不公平的。
- 每个 CPU 的运行队列 cfs_rq 都维护一个 min_vruntime 字段，记录该运行队列中所有进程的 vruntime 最小值，新进程的初始 vruntime 值就以它所在运行队列的 min_vruntime 为基础来设置，与老进程保持在合理的差距范围内。

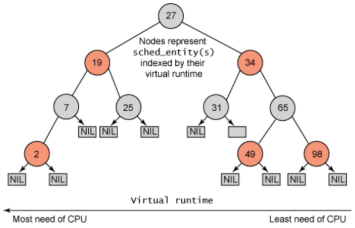
CFS 调度



具体实现：休眠进程的 vruntime 一直保持不变吗？

- 如果休眠进程的 vruntime 保持不变，而其他运行进程的 vruntime 一直在推进，那么等到休眠进程终于唤醒的时候，它的 vruntime 比别人小很多，会使它获得长时间抢占 CPU 的优势，其他进程就要饿死了。
- 在休眠进程被唤醒时重新设置 vruntime 值，以 min_vruntime 值为基础，给予一定的补偿，但不能补偿太多。

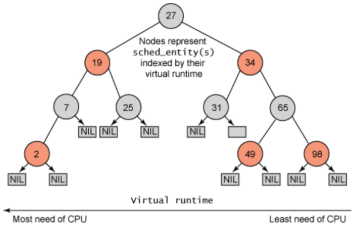
CFS 调度



具体实现：休眠进程在唤醒时会立刻抢占 CPU 吗？

- 休眠进程在醒来的时候有能力抢占 CPU 是大概率事件，这也是 CFS 调度算法的本意，即保证交互式进程的响应速度，因为交互式进程等待用户输入会频繁休眠。
- 主动休眠的进程同样也会在唤醒时获得补偿，这类进程往往并不要求快速响应，它们同样也会在每次唤醒并抢占，这有可能会导致其它更重要的应用进程被抢占，有损整体性能。
- sched_features 的 WAKEUP_PREEMPT 位

CFS 调度



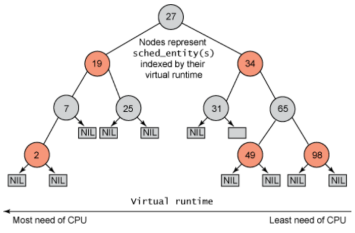

```
# grep min_vruntime /proc/sched_debug
.min_vruntime : 12403175.972743
.min_vruntime : 14422108.528121

# grep min_vruntime /proc/sched_debug
.min_vruntime : 12403175.972743
.min_vruntime : 14422108.528121
```

具体实现：进程从一个 CPU 迁移到另一个 CPU 上的时候 vruntime 会不会变？

- 在多 CPU 的系统上，不同的 CPU 的负载不一样，有的 CPU 更忙一些，而每个 CPU 都有自己的运行队列，每个队列中的进程的 vruntime 也走得有快有慢，比如我们对比每个运行队列的 min_vruntime 值，都会有不同
- 当进程从一个 CPU 的运行队列中出来时，它的 vruntime 要减去队列的 min_vruntime 值；
- 当进程加入另一个 CPU 的运行队列时，它的 vruntime 要加上该队列的 min_vruntime 值。

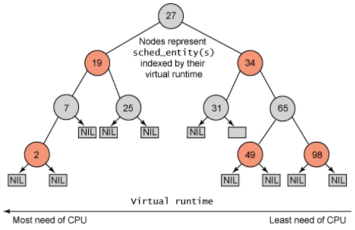
CFS 调度



具体实现：vruntime 溢出问题

- 红黑树中实际的作为 key 的不是 vruntime 而是 vruntime - min_vruntime。min_vruntime 是当前红黑树中最小的 key
- vruntime 的类型 usigned long
- 进程的虚拟时间是一个递增的正值，因此它不会是负数，但是它有它的上限，就是 unsigned long 所能表示的最大值，如果溢出了，那么它就会从 0 开始回滚，如果这样的话，结果会怎样？

CFS 调度



具体实现：vruntime 溢出问题

一个例子

unsigned char a = 251, b = 254;
b += 5;
//b 回滚了，导致 a 大于 b，应该是 b 比 a 大 8
//怎么做到真正的结果呢？改为以下：
unsigned char a = 251, b = 254;
b += 5;
signed char c = a - 250, d = b - 250;
//到此判断 c 和 d 的大小

第十二讲：多处理器调度

第 5 节：BFS 调度算法

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

BFS 工作原理

出处：Analysis of the BFS Scheduler in FreeBSD

BFS 调度算法是一种时间片轮转算法的变种.

在多处理机情况的单就绪队列（双向链表）选择，增加了队列互斥访问的开销，但减少了负载均衡算法开销。

BFS 就绪队列

线程优先级：有 103 个优先级

- 100 个静态的实时优先级；
- 3 个普通优先级 SCHEDISO (isochronous)、SCHEDNORMAL 和 SCHEDIDLEPRIO (idle priority scheduling)；

单就绪队列

- 所有 CPU 共享一个双向链表结构的单就绪队列；
- 所有线程按优先级排队；
- 相同优先级的每个线程有一个时间片长度和虚拟截止时间；

虚拟截止时间（Virtual Deadline）

时间片大小

时间片大小由算法参数指定，可在 1ms 到 1000ms 间选择，缺省设置为 6ms；

- 它是一个关于就绪队列中线程等待 CPU 最长时间的排序，并不是真实的截止时间；
- 线程时间片用完时，重新计算虚拟截止时间；
- 事件等待结束时，虚拟截止时间保持不变，以抢先相同优先级的就绪线程；
- 为了让线程在上次运行的 CPU 上运行，不同 CPU 对线程的虚拟截止时间加一个权重；

虚拟截止时间计算

依据当前时间、线程优先级和时间片设置计算；

```
1 offset = jiffies + (prior_atio * rr_interval)
2 prioratio increases by 10% for every nice level
```

虚拟截止时间计算结果：https://wikimili.com/en/Brain_Fuck_Scheduler

相关线程状态置换

- 时间片用完：重新设置虚拟截止时间后，插入就绪队列；
- 等待事件出现：虚拟截止时间保持不变，抢先低优先级线程或插入就绪队列；

BFS 与 CFS 的性能对比 (2012)

测试硬件环境

测试用例集

- Linux kernel v3.6.2.2 的 GCC 编译
- Linux kernel v3.6.2 内核源代码树的 lrzip 压缩
- 从 720p 到 360p 的 MPEG2 视频 ffmpeg 压缩

压缩测试

编译测试

视频编码测试