# 二、硬件结构

常见的**寄存器**种类：

* 通用寄存器，用来存放需要进行运算的数据，比如需要进行加和运算的两个数据。
* 程序计数器，用来存储 CPU 要执行下一条指令「所在的内存地址」，注意不是存储了下一条要执行的指令，此时指令还在内存中，程序计数器只是存储了下一条指令「的地址」。
* 指令寄存器，用来存放当前正在执行的指令，也就是指令本身，指令被执行完成之前，指令都存储在这里。

**总线**是用于 CPU 和内存以及其他设备之间的通信，总线可分为 3 种：

* 地址总线，用于指定 CPU 将要操作的内存地址；
* 数据总线，用于读写内存的数据；
* 控制总线，用于发送和接收信号，比如中断、设备复位等信号，CPU 收到信号后自然进行响应，这时也需要控制总线；

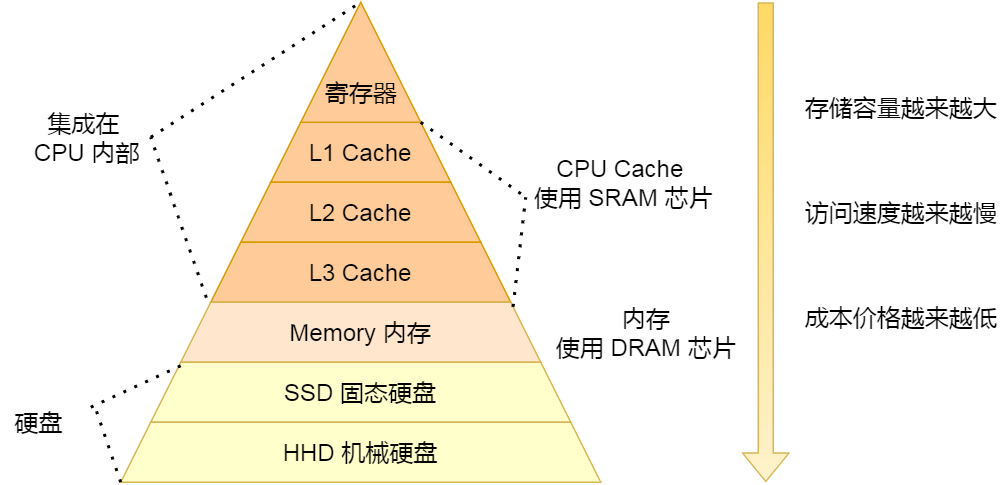
CPU 要读写内存数据的时候，一般需要通过下面这三个总线：

* 首先要通过「地址总线」来指定内存的地址；
* 然后通过「控制总线」控制是读或写命令；
* 最后通过「数据总线」来传输数据；

CPU 执行程序的过程如下：

* 第一步，CPU 读取「程序计数器」的值，这个值是指令的内存地址，然后 CPU 的「控制单元」操作「地址总线」指定需要访问的内存地址，接着通知内存设备准备数据，数据准备好后通过「数据总线」将指令数据传给 CPU，CPU 收到内存传来的数据后，将这个指令数据存入到「指令寄存器」。
* 第二步，「程序计数器」的值自增，表示指向下一条指令。这个自增的大小，由 CPU 的位宽决定，比如 32 位的 CPU，指令是 4 个字节，需要 4 个内存地址存放，因此「程序计数器」的值会自增 4；
* 第三步，CPU 分析「指令寄存器」中的指令，确定指令的类型和参数，如果是计算类型的指令，就把指令交给「逻辑运算单元」运算；如果是存储类型的指令，则交由「控制单元」执行；

## 存储器的层次结构



当 CPU 需要访问内存中某个数据的时候，如果寄存器有这个数据，CPU 就直接从寄存器取数据即可，如果寄存器没有这个数据，CPU 就会查询 L1 高速缓存，如果 L1 没有，则查询 L2 高速缓存，L2 还是没有的话就查询 L3 高速缓存，L3 依然没有的话，才去内存中取数据。

## CPU Cache 的数据写入

数据写入 Cache 之后，内存与 Cache 相对应的数据将会不同，这种情况下 Cache 和内存数据都不一致了，于是我们肯定是要把 Cache 中的数据同步到内存里的。

问题来了，那在什么时机才把 Cache 中的数据写回到内存呢？为了应对这个问题，下面介绍两种针对写入数据的方法：

* 写直达（Write Through）
* 写回（Write Back）

### [#](https://xiaolincoding.com/os/1_hardware/cpu_mesi.html#%E5%86%99%E7%9B%B4%E8%BE%BE)写直达

**把数据同时写入内存和 Cache 中**

* 如果数据已经在 Cache 里面，先将数据更新到 Cache 里面，再写入到内存里面；
* 如果数据没有在 Cache 里面，就直接把数据更新到内存里面。

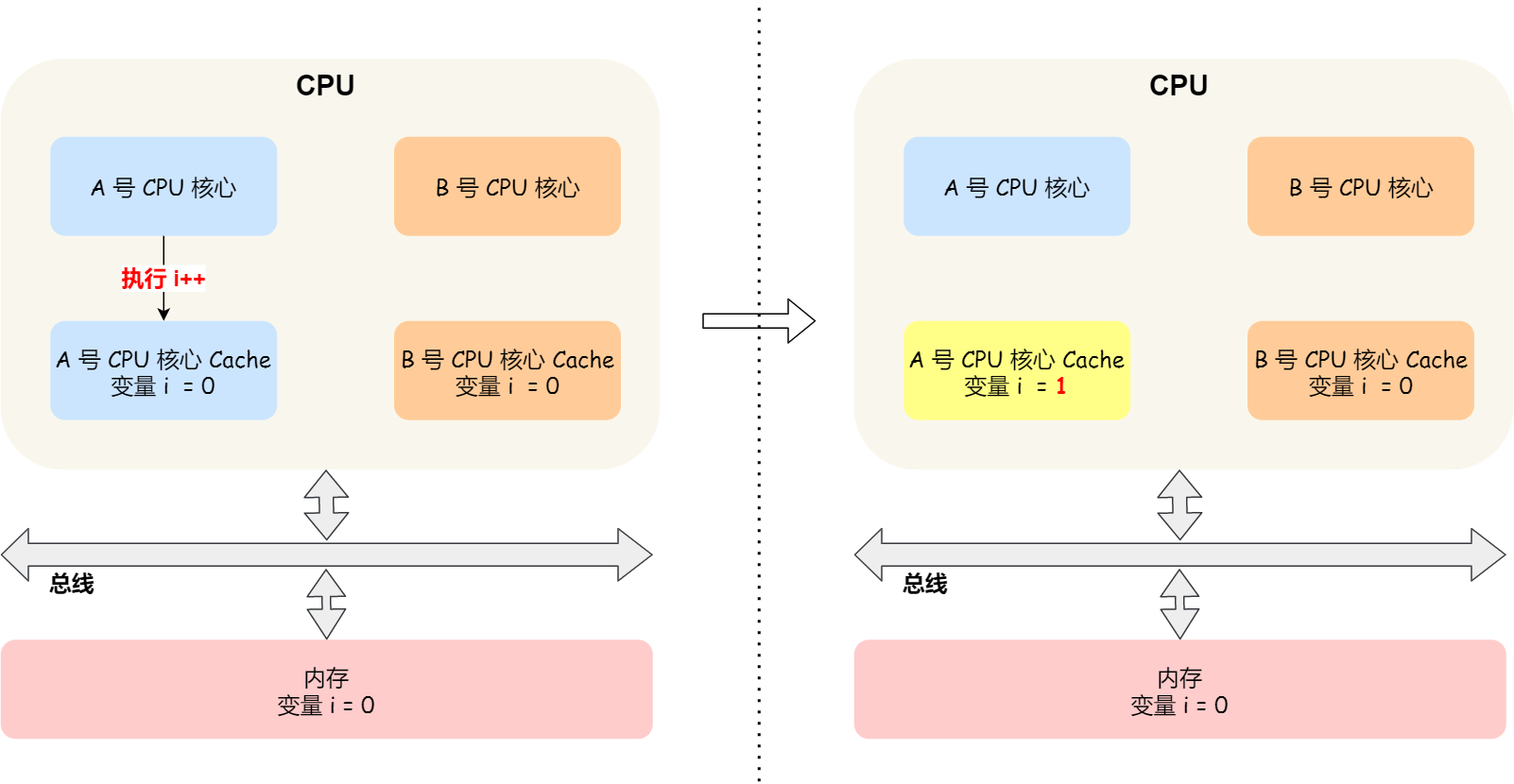
### 写回

**当发生写操作时，新的数据仅仅被写入 Cache Block 里，只有当修改过的 Cache Block「被替换」时才需要写到内存中**

* 如果当发生写操作时，数据已经在 CPU Cache 里的话，则把数据更新到 CPU Cache 里，同时标记 CPU Cache 里的这个 Cache Block 为脏（Dirty）的，这个脏的标记代表这个时候，我们 CPU Cache 里面的这个 Cache Block 的数据和内存是不一致的，这种情况是不用把数据写到内存里的；
* 如果当发生写操作时，数据所对应的 Cache Block 里存放的是「别的内存地址的数据」的话，就要检查这个 Cache Block 里的数据有没有被标记为脏的：
  + 如果是脏的话，我们就要把这个 Cache Block 里的数据写回到内存，然后再把当前要写入的数据，先从内存读入到 Cache Block 里（注意，这一步不是没用的），然后再把当前要写入的数据写入到 Cache Block，最后也把它标记为脏的；
  + 如果不是脏的话，把当前要写入的数据先从内存读入到 Cache Block 里，接着将数据写入到这个 Cache Block 里，然后再把这个 Cache Block 标记为脏的就好了

## 缓存一致性问题

 CPU 都是多核的，由于 L1/L2 Cache 是多个核心各自独有的，那么会带来多核心的**缓存一致性（Cache Coherence）** 的问题



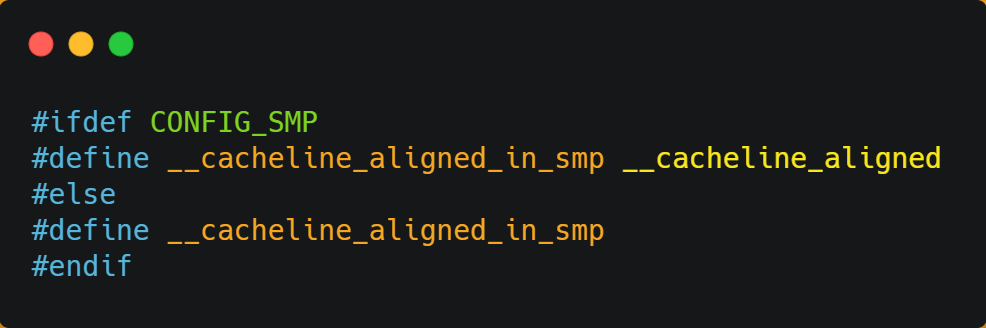
要保证做到下面这 2 点：

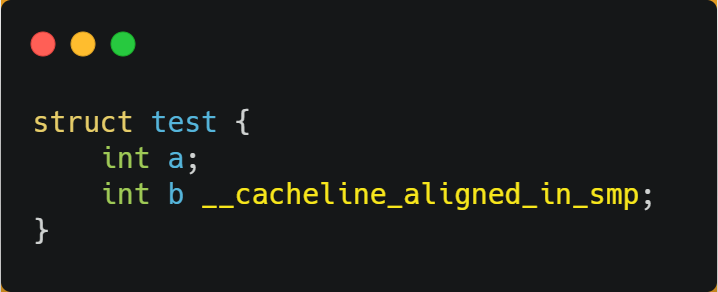
* 第一点，某个 CPU 核心里的 Cache 数据更新时，必须要传播到其他核心的 Cache，这个称为写传播（*Write Propagation*）；
* 第二点，某个 CPU 核心里对数据的操作顺序，必须在其他核心看起来**顺序是一样**的，这个称为事务的串行化（*Transaction Serialization*）。

### 伪共享

多个线程同时读写同一个 Cache Line 的不同变量时，而导致 CPU Cache 失效的现象称为**伪共享（False Sharing）**。

对于多个线程共享的热点数据，即经常会修改的数据，应该避免这些数据刚好在同一个 Cache Line 中，否则就会出现为伪共享的问题。





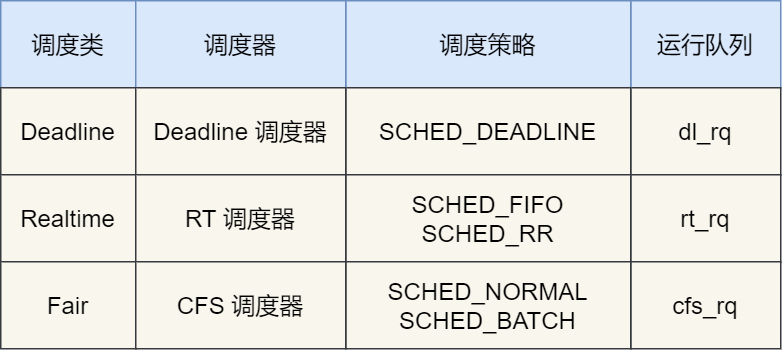
针对在同一个 Cache Line 中的共享的数据，如果在多核之间竞争比较严重，为了防止伪共享现象的发生，可以采用上面的宏定义使得变量在 Cache Line 里是对齐的。

## CPU 如何选择线程的？

在 Linux 内核中，进程和线程都是用 task\_struct 结构体表示的，区别在于线程的 task\_struct 结构体里部分资源是共享了进程已创建的资源，比如内存地址空间、代码段、文件描述符等，所以 Linux 中的线程也被称为轻量级进程

Linux 内核里的调度器，调度的对象就是 task\_struct

### 调度类



Deadline 和 Realtime是应用于实时任务的：

* *SCHED\_DEADLINE*：是按照 deadline 进行调度的，距离当前时间点最近的 deadline 的任务会被优先调度；
* *SCHED\_FIFO*：对于相同优先级的任务，按先来先服务的原则，但是优先级更高的任务，可以抢占低优先级的任务，也就是优先级高的可以「插队」；
* *SCHED\_RR*：对于相同优先级的任务，轮流着运行，每个任务都有一定的时间片，当用完时间片的任务会被放到队列尾部，以保证相同优先级任务的公平性，但是高优先级的任务依然可以抢占低优先级的任务；

Fair 调度类是应用于普通任务，都是由 CFS 调度器管理的，分为两种调度策略：

* *SCHED\_NORMAL*：普通任务使用的调度策略；
* *SCHED\_BATCH*：后台任务的调度策略，不和终端进行交互，因此在不影响其他需要交互的任务，可以适当降低它的优先级。

CFS：**完全公平调度（Completely Fair Scheduling）**

这个算法的理念是想让分配给每个任务的 CPU 时间是一样，于是它为每个任务安排一个虚**拟运行时间 vruntime**，如果一个任务在运行，其运行的越久，该任务的 vruntime 自然就会越大，而没有被运行的任务，vruntime 是不会变化的。

那么，**在 CFS 算法调度的时候，会优先选择 vruntime 少的任务**，以保证每个任务的公平性。

普通任务之间还是有优先级区分的，所以在计算虚拟运行时间 vruntime 还要考虑普通任务的**权重值**

### CPU 运行队列

每个 CPU 都有自己的**运行队列（Run Queue, rq）**，包含三个运行队列，Deadline 运行队列 dl\_rq、实时任务运行队列 rt\_rq 和 CFS 运行队列 cfs\_rq，其中 cfs\_rq 是用红黑树来描述的，按 vruntime 大小来排序的，最左侧的叶子节点，就是下次会被调度的任务。

这几种调度类是有优先级的，优先级如下：**Deadline > Realtime > Fair**，这意味着 Linux 选择下一个任务执行的时候，会按照此优先级顺序进行选择，也就是说先从 dl\_rq 里选择任务，然后从 rt\_rq 里选择任务，最后从 cfs\_rq 里选择任务。因此，**实时任务总是会比普通任务优先被执行**。

### 调整优先级

普通任务之间：调整nice值

普通任务也可以改变调度策略变成实时任务

# 2.6 什么是软中断？

## 中断是什么？

在计算机中，中断是系统用来响应**硬件设备**请求的一种机制，操作系统收到硬件的中断请求，会打断正在执行的进程，然后调用内核中的中断处理程序来响应请求。

中断是一种**异步**的事件处理机制，可以提高系统的并发处理能力。操作系统收到了中断请求，会打断其他进程的运行，所以**中断请求的响应程序，也就是中断处理程序，要尽可能快的执行完，这样可以减少对正常进程运行调度的影响。**

中断处理程序在响应中断时，可能还会「**临时关闭中断**」，这意味着，如果**当前中断处理程序没有执行完之前，系统中其他的中断请求都无法被响应**，也就说中断有可能会丢失，所以中断处理程序要短且快。

## 什么是软中断？

Linux 系统**为了解决中断处理程序执行过长和中断丢失的问题，将中断过程分成了两个阶段，分别是「上半部和下半部分」**。

* **上半部用来快速处理中断**，一般会**暂时关闭中断请求**，主要负责处理跟硬件紧密相关或者时间敏感的事情。**也就是硬中断**，主要是负责耗时短的工作，特点是**快速执行**。
* **下半部用来延迟处理上半部未完成的工作**，一般以「内核线程」的方式运行。**也就是软中断**，主要是负责上半部未完成的工作，通常都是耗时比较长的事情，特点是**延迟执行**；

举例：

网卡收到网络包后，通过 DMA 方式将接收到的数据写入内存，接着会通过**硬件中断**通知内核有新的数据到了，于是内核就会调用对应的中断处理程序来处理该事件，这个事件的处理也是会分成上半部和下半部。

上部分要做的事情很少，会先禁止网卡中断，避免频繁硬中断，而降低内核的工作效率。接着，内核会触发一个**软中断**，把一些处理比较耗时且复杂的事情，交给「软中断处理程序」去做，也就是中断的下半部，其主要是需要从内存中找到网络数据，再按照网络协议栈，对网络数据进行逐层解析和处理，最后把数据送给应用程序。

还有一个区别，**硬中断（上半部）是会打断 CPU 正在执行的任务，然后立即执行中断处理程序，而软中断（下半部）是以内核线程的方式执行**，并且每一个 CPU 都对应一个软中断内核线程，名字通常为「ksoftirqd/CPU 编号」，比如 0 号 CPU 对应的软中断内核线程的名字是 ksoftirqd/0。软中断不只是包括硬件设备中断处理程序的下半部，一些内核自定义事件也属于软中断，比如**内核调度**等、**RCU 锁**（内核里常用的一种锁）等。

# 3.1 Linux 内核 vs Windows 内核