操作系统八股文

更新记录：

2022 年 8 月 6 日：在彭博操作系统.pdf 的基础上完成初稿

2022 年 8 月 7 日：

1、 根据 byx 的意见在进程间通信部分加入套接字；2、在 IO 部分加入 MMIO 部分

2022 年 8 月 17 日：

1、加入了对 page cache 的过程说明；2、加入了内存屏障的部分

进程和线程

进程是资源分配的最小单位，而线程是 CPU 调度的最小单位。多线程之间共享同　个进程的地址空间，线程间通信简单，线程创建、销毁和切换简单，速度快，占用内存少，但是线程间会相互影响，

　个线程意外终止会导致同　个进程的其他线程也终止。而多进程间拥有各自独立的运行地址空间， 进程间不会相互影响，程序可靠性强，但是进程创建、销毁和切换复杂，速度慢，占用内存多，进程间通信复杂。

进程和线程的实现

在内存中都会有　个 task_struct 进程描述符与之对应。进程描述符包含了内核管理进程所有有用的信息，包括调度参数、打开文件描述符等等。进程描述符从进程创建开始就　直存在于内核堆栈中。Linux 是通过 PID 来区分不同的进程，内核会将所有进程的任务结构组成为　个双向链表。PID 能够直接被映射称为进程的任务结构所在的地址，从而不需要遍历双向链表直接访问。

从内核来讲，并没有线程的概念，linux 把线程当做进程来实现。内核并没有特殊的调度算法或者定义特别的数据结构来表示线程。线程仅被看做　个与其它进程共享某些资源的进程。每个线程都拥有唯　属于自己的 task_struct，所以在内核中，它看起来像是　个普通的进程（只是它和其他　些进程共享某些资源），只是它没有自己独立的内存地址空间。

协程

协程，是　种比线程更加轻量级的存在。　个线程也可以拥有多个协程。

系统在线程等待 IO 的时候，会阻塞当前线程，切换到其它线程，这样在当前线程等待 IO 的过

程中，其它线程可以继续执行。当系统线程较少的时候没有什么问题，但是当线程数量非常多的时候，会产生问题。 是系统线程会占用非常多的内存空间， 是过多的线程切换会占用大量的系统时间。线程往往需要对公共数据加锁，锁会导致线程调度。用户的线程是在用户态执行，而线程调度和管理是在内核态实现，所以线程调度需要从用户态转到内核态，再从内核态转到用户态。切到内核态时需要保存用户态上下文，再切到用户态时，需要恢复用户态上下文，而线程的用户态上下文比协程上下文大得多。

协程运行在线程之上，当 个协程执行完成后，可以选择主动让出，让另 个协程运行在当前线程之上。协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程，而且协程的切换在用户态完成，切换的代价比线程从用户态到内核态的代价小很多。

协程的注意事项

什么是协程？

实际上协程并不是什么银弹，协程只有在等待 IO 的过程中才能重复利用线程，上面我们已经讲过了，线程在等待 IO 的过程中会陷入阻塞状态，意识到问题没有？

假设协程运行在线程之上，并且协程调用了一个阻塞 IO 操作，这时候会发生什么？实际上操作系统并不知道协程的存在，它只知道线程，因此在协程调用阻塞 IO 操作的时候，操作系统会让线程进入阻塞状态，当前的协程和其它绑定在该线程之上的协程都会陷入阻塞而得不到调度，这往往是不能接受的。

因此在协程中不能调用导致线程阻塞的操作。也就是说，协程只有和异步 IO 结合起来，才能发挥最大的威力。

那么如何处理在协程中调用阻塞 IO 的操作呢？一般有 2 种处理方式：

1. 在调用阻塞 IO 操作的时候，重新启动一个线程去执行这个操作，等执行完成后，协程再去读取结果。这其实和多线程没有太大区别。
2. 对系统的 IO 进行封装，改成异步调用的方式，这需要大量的工作，最好寄希望于编程语言原生支持。

协程对计算密集型的任务也没有太大的好处，计算密集型的任务本身不需要大量的线程切换，因此协程的作用也十分有限，反而还增加了协程切换的开销。

协程相对于多线程的优缺点

协程的概念，为什么要用协程，以及协程的使用

优点：

多线程编程是比较困难的， 因为调度程序任何时候都能中断线程， 必须记住保留锁， 去保护程序中重要部分， 防止多线程在执行的过程中断。

而协程默认会做好全方位保护， 以防止中断。我们必须显示产出才能让程序的余下部分运行。对协程来说，无需保留锁， 而在多个协程线程之间同步操作，协程自身就会同步，因为在任意时刻，只有一个协程运行。总结下大概下面几点：

1. 无需系统内核的上下文切换，减小开销；
2. 无需原子操作锁定及同步的开销，不用担心资源共享的问题；
3. 单线程即可实现高并发，单核 CPU 即便支持上万的协程都不是问题，所以很适合用于高并发处理，尤其是在应用在网络爬虫中。

缺点：

1. 无法直接使用 CPU 的多核协程的本质是个单线程，它不能同时用上单个 CPU 的多个核，协程需要和进程配合才能运行在多 CPU 上。
2. 处处都要使用非阻塞代码

线程共享和独有资源

线程共享的有堆，数据段，代码段、全局变量、静态变量、进程的公共数据、进程打开的文件描述符、信号、进程的当前目录和进程用户 ID 与进程组 ID。

线程独占的有栈。

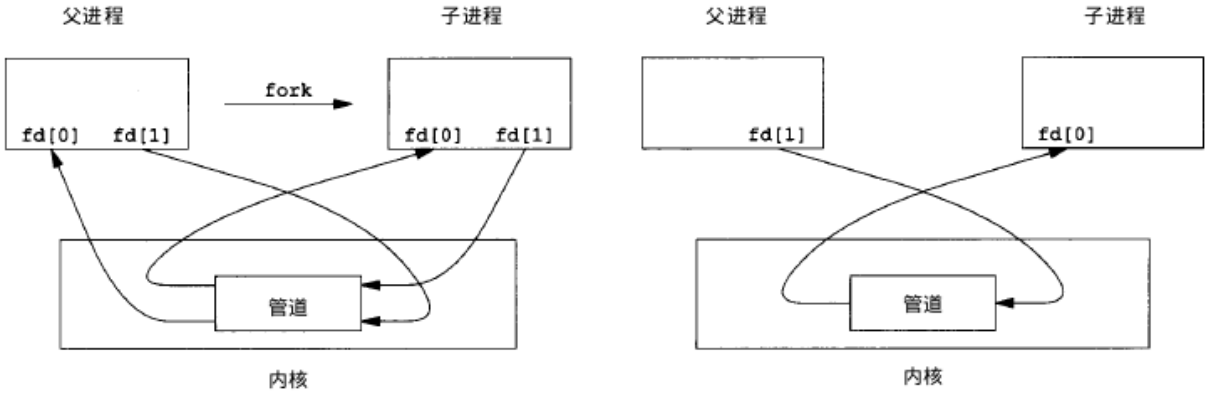
一般默认情况下，线程栈是在进程的堆中分配栈空间，每个线程拥有独立的栈空间，为了避免线程之间的栈空间踩踏，线程栈之间还会有以小块 guardsize 用来隔离保护各自的栈空间，一旦另一个线程踏入到这个隔离区，就会引发段错误。

进程之间的通信

1. 管道（匿名管道和有名管道，有名管道可用于非血缘关系进行通信）：实为内核使用环形队列机制，借助内核缓冲区(4k) 实现。特点：面向字节流，双工，单向通信，两个管道实现双向通信。
2. 消息队列：消息队列提供了　个从　个进程向另外　个进程发送　块数据的方法。 特点：消息队列可以认为是　个全局的　个链表，链表节点中存放着数据报的类型和内容，有消息队列的标识符进行标记。消息队列允许　个或多个进程写入或者读取消息。消息队列可实现双向通信。
3. 信号量：信号量用来管理临界资源的。它本身只是一种外部资源的标识，不具有数据交换功能，而是通过控制其他的通信资源实现（父子）进程间通信。
4. 共享内存：将同　块物理内存　块映射到不同的进程的虚拟地址空间中，实现不同进程间对同　资源的共享。特点：不用从用户态到内核态的频繁切换和拷贝数据，直接从内存中读取就可以。共享内存是临界资源，所以需要操作时必须要保证原子性。使用信号量或者互斥锁都可以。
5. 套接字通信： socket 套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。绑定 - 监听 / 链接服务器 - 读写 - 关闭

浅析进程间通信的几种方式（含实例源码）

管道（匿名管道）：

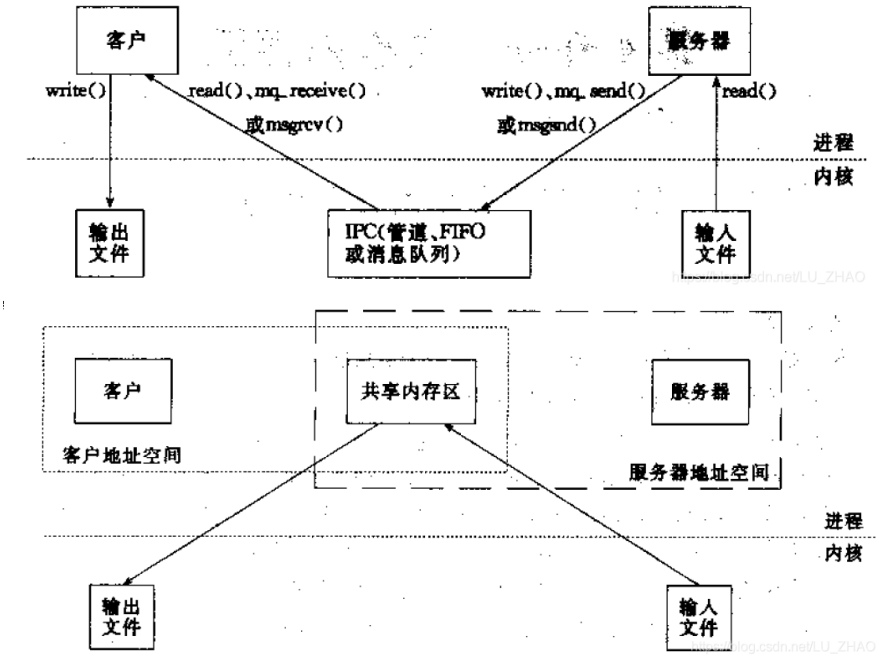


左图 - fork之后的半双工管道　　右图 - 从父进程到子进程的管道

命名管道（FIFO）：

- 命名管道是一个存在于硬盘上的文件，而管道是存在于内存中的特殊文件。所以当使用命名管道的时候必须先 open 将其打开。
- 命名管道可以用于任何两个进程之间的通信，不管这两个进程是不是父子进程，也不管这两个进程之间有没有关系。

【操作系统】 为什么说共享内存是最快的一种IPC方式呢？



线程之间的通信

- 线程间的通信：信号量，可用于线程同步，通过 PV 操作。P(SV):如果信号量 SV　大于 0，将它减　；如果 SV　值为 0，则挂起该线程。V(SV)：如果有其他进程因为等待 SV 而挂起，则唤醒，然后将 SV+1； 否则直接将 SV+1。

- 互斥量：当进入临界区 时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。
- 条件变量：条件变量提供　种线程间通信机制，当某个共享数据达到某个值时，唤醒等待这个共享数据的　个/多个线程。

进程的上下文切换

进程的上下文切换，实质上就是被中断运行进程与待运行进程的上下文切换。

1. 切换新的页表，然后使用新的虚拟地址空间。（线程共享同　虚拟地址空间，不需要这步)
2. 切换内核栈，加入新的内容(PCB 控制块，资源相关)，硬件上下文(处理器的状态) 切换
   a. EIP 寄存器，用来存储 CPU 要读取指令的地址
   b. ESP 寄存器：指向当前线程栈的栈顶位置
   c. 其他通用寄存器的内容：包括代表函数参数的 rdi、rsi 等等
   d. 线程栈中的内存内容。这些数据内容， 般将其称为"上下文"或者"现场"。

进程的上下文切换

- 上下文切换：操作系统通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为进程的上下文切换。
- 进程的上下文：进程的物理实体（代码和数据等）和支持运行的环境合称为进程的上下文。进程的上下文包括用户级上下文和系统级上下文。
  - 用户级上下文：由用户的程序块、数据块、运行时的堆和用户栈（统称为用户堆栈）等组成的用户空间信息被称为用户级上下文。
  - 系统级上下文：由进程标识信息、进程现场信息、进程控制信息（包含进程表、页表、打开文件表等）和系统内核栈等组成的内核空间信息被称为系统级上下文。
- 寄存器上下文：处理器中各个寄存器的内容被称为寄存器上下文（或硬件上下文）。

系统级上下文	进程标识信息
	进程现场信息
	进程控制信息
	系统内核级
用户级上下文	用户堆栈
	用户数据块
	用户程序块
	共享地址空间

进程、线程上下文切换

中断上下文切换

跟进程上下文不同，中断上下文切换不涉及进程的用户态。所以，即便中断过程打断了一个正处在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。中断上下文，只包括内核态中断服务程序执行所必需的状态，也就是 CPU 寄存器、内核堆栈、硬件中断参数等。

线程上下文切换

内核中的任务调度实际是在调度线程，进程只是给线程提供虚拟内存、全局变量等资源。线程上下文切换时，共享相同的虚拟内存和全局变量等资源不需要修改。而线程自己的私有数据，如栈和寄存器等，上下文切换时需要保存。

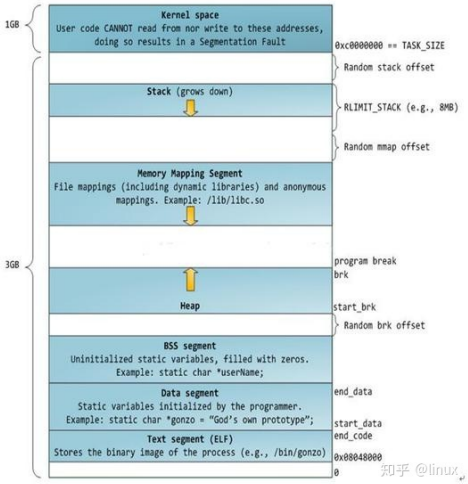
切换的性能消耗

一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲（processor' s Translation Lookaside Buffer (TLB)）会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题。

进程的地址空间布局

Linux 对进程地址空间有个标准布局，地址空间中由各个不同的内存段组成 (Memory Segment)，主要的内存段如下：

- 程序段 (Text Segment)：可执行文件代码的内存映射
- 数据段 (Data Segment)：可执行文件的已初始化全局变量的内存映射
- BSS 段 (BSS Segment)：未初始化的全局变量或者静态变量（用零页初始化）
- 堆区 (Heap)：存储动态内存分配，匿名的内存映射
- 栈区 (Stack)：进程用户空间栈，由编译器自动分配释放，存放函数的参数值、局部变量的值等
- 映射段(Memory Mapping Segment)：任何内存映射文件



自底向上的方式进行：（低到高）

- 代码段：主要是程序的代码以及编译时静态链接进来的库。这段内存大小在程序运行之前就已经确定，而且是只读，可能存在　些常量，比如字符串常量。
- 数据段：分为 data 和 bss 两个段，表现为静态内存段，data 段存放已初始化的全局变量（静态内 存分配的变量和初始化全局变量 ）。bss 段存放未初始化的全局变量。
- 堆 段：用于程序动态内存分配和管理，如何分配和管理由程序的开发者决定，大小不固定（跟机器内存有关系），可以动态伸缩。
- 映射段：该内存区域存放链接其它动态程序库的向量，共享内存映射向量等等。
- 栈 段：分配和释放。
- 输入的环境变量和参数段：主要内存程序执行时的环境变量，输入参数等等。

函数调用的压栈出栈过程

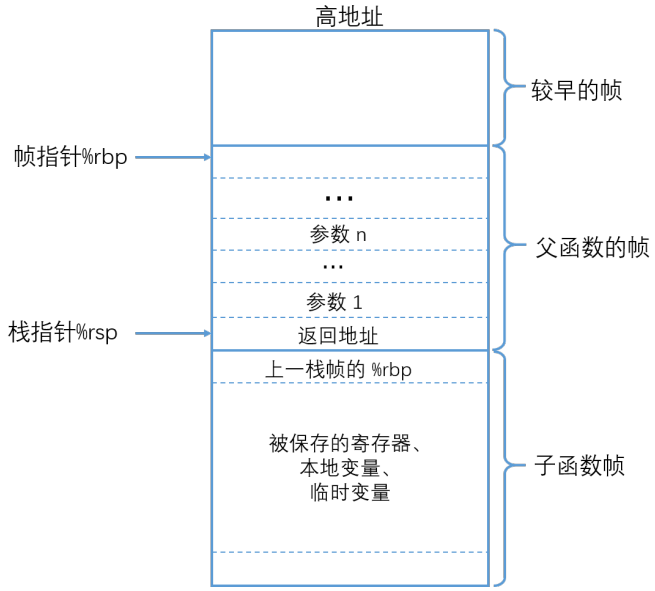
esp 指向栈中的下一个可用空间

栈帧详解 （图中错误，鉴于 ebp 是在 callee 中入栈的，应该归于新栈帧）

什么是栈帧 （图中错误，返回地址应该先于旧栈底指针 ebp 入栈）

【C语言】函数调用的参数压栈（详解）

x86-64 下函数调用及栈帧原理



一次调用的栈帧由如下部分组成（从底到顶）:

- 局部变量
- 被调用者参数（从右向左）
- 返回地址，即调用时的下一条指令 ip
- 调用者的栈底指针 ebp

函数调用过程：

- 参数入栈：将参数按照调用约定(C 是从右向左)依次压入系统栈中
- 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
- 代码跳转：处理器将代码区跳转到被调用函数的入口处
- 栈帧调整：
  - 将调用者的 ebp 压栈处理，保存指向栈底的 ebp 的地址（方便函数返回之后的现场恢复），此时 esp 指向新的栈顶位置：push ebp
  - 将当前栈帧切换到新栈帧(将 eps 值装入 ebp，更新栈帧底部)，这时 ebp 指向栈顶，而此时栈顶就是 old ebp： mov ebp, esp
  - 给新栈帧中的局部变量分配空间：sub esp, XXX

函数返回过程：

- 保存被调用函数的返回值到 eax 寄存器中：mov eax, xxx
- 恢复 esp 同时回收局部变量空间：mov ebp, esp
- 将上一个栈帧底部位置恢复到 ebp：pop ebp
- 弹出当前栈顶元素，从栈中取到返回地址，并跳转到该位置 ret

返回地址要先于旧 ebp 入栈

因为函数的调用可以分为三个阶段：

- 调用者执行：把函数参数压栈
- call 指令：将返回地址压栈，并转跳到新函数目标地址
- 被调用者执行：把旧的栈底压栈，设置新的栈底

```
push 3    // 阶段 1
push 4
call add_num    // 阶段 2

add_num proc
    push ebp    // 阶段 3
    mov ebp,esp
    mov eax,[ebp+8]
    add eax,[ebp+12]
    pop ebp
    ret
add_num endp
```

为什么栈状态的保存与更新要在 caller 中完成？更细化的说，如果 ebp 先于返回地址入栈会发生什么？（如 什么是栈帧 图中所示的结构）

个人感觉是出于功能的内聚性，使被调用的函数自洽地完成栈的新建和恢复过程，就像为什么寄存器要分为 caller save 和 callee save 一样

寄存器传参优化

linux X64函数参数传递过程研究

函数传参存在两种方式，一种是通过栈，一种是通过寄存器。对于 x64 体系结构，如果函数参数不大于 6 个时，使用寄存器传参，对于函数参数大于 6 个的函数，前六个参数使用寄存器传递，后面的使用栈传递。参数传递的规律是固定的，即前 6 个参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9，后面的依次从 “右向左” 放入栈中。

参数传递要从右到左

为了满足变长参数的情况，变长参数都是左侧固定，右侧变长（以 printf 为例）。

传参可以分为两个阶段：调用者以栈的形式压入，被调用者以偏移量的形式读取。如果从左向右压入，在读取阶段，不知道左侧的固定参数（例如 printf 中的 format）相对于栈顶的偏移量，因为后续压入的参数的数量是不确定的，对被调用者不可知。而如果从右向左压入，最左侧的确定的参数就会最接近栈顶，偏移量也固定，可以直接读出，并以此为指导进行后续不确定参数的偏移量和数量确定。

从左向右：| format | ... | ... | eps

从右向左：| ... | ... | format | eps

对于寄存器传参就无所谓先后。

函数调用完成后清理传入的参数

C/C++子函数参数传递，堆栈帧、堆栈参数详解

释放参数涉及两种子函数调用标准，一种是 STDCALL 标准，一种是 C 标准。两种在参数的堆栈传递细节几乎完全相同，不同的是释放参数的方式。

STDCALL 调用规范：

```
1  add_num proc
2     push ebp
3     mov ebp,esp
4     mov eax,[ebp+8]
5     add eax,[ebp+12]
6     pop ebp
7     ret 8
8  add_num endp
```

C 调用规范：

```
push 3
push 4
call add_num
add esp,8
```

两种方式的核心思想就是修改 esp，使 esp 指向堆栈参数 3 和 4 所在位置的前一个堆栈。但是 STDCALL 调用规范是在过程内部修改 esp（ret 8 为将堆栈中返回地址弹出到 EIP 后，再将 ESP 加 8）；C 调用规范是在子过程外部，在主调过程修改 esp。

另引用这两种方式的优缺点：

STDCALL 不仅减少了子程序调用产生的代码量（减少了一条指令），还保证了调用程序永远不会忘记清除堆栈。另一方面，C 调用规范允许子程序声明不同数量的参数，主调程序可以决定传递多少个参数。C 语言的 printf 函数就是一个例子

函数的返回

x86-64 下函数调用及栈帧原理

函数执行完成后，要将返回结果保存在 rax 寄存器中，在恢复 caller 的栈结构（ebp，esp）后一步骤可由 leave 指令完成

寄存器状态的保存

x86-64 下函数调用及栈帧原理

这里还要区分一下“Caller Save”和”Callee Save”寄存器，即寄存器的值是由”调用者保存“还是由”被调用者保存“。当产生函数调用时，子函数内通常也会使用到通用寄存器，那么这些寄存器中之前保存的调用者(父函数）的值就会被覆盖。为了避免数据覆盖而导致从子函数返回时寄存器中的数据不可恢复，CPU 体系结构中就规定了通用寄存器的保存方式。

如果一个寄存器被标识为”Caller Save”，那么在进行子函数调用前，就需要由调用者提前保存好这些寄存器的值，保存方法通常是把寄存器的值压入堆栈中，调用者保存完成后，在被调用者（子函数）中就可以随意覆盖这些寄存器的值了。如果一个寄存被标识为“Callee Save”，那么在函数调用时，调用者就不必保存这些寄存器的值而直接进行子函数调用，进入子函数后，子函数在覆盖这些寄存器之前，需要先保存这些寄存器的值，即这些寄存器的值是由被调用者来保存和恢复的。

Why make some registers caller-saved and others callee-saved? Why not make the ca...

Having some call-preserved registers saves significant static code-size because it means you don't have to write store/load instructions before / after every function call. Just once for the whole function. Only once inside the callee, if at all. Most functions are called from

multiple call-sites; that's why they're functions instead of just getting inlined.

就像电梯用一个平均重量的重物作为负重一样，把寄存器分为 caller save 和 callee save 可以同时在调用者和被调用者中减少数据恢复所需要的指令数量。也在一定程度上让 caller 和 callee 使用的寄存器错开

进程栈、线程栈、内核栈、中断栈

浅谈Linux 中的进程栈、线程栈、内核栈、中断栈

进程栈

进程虚拟地址空间中的栈区，正指的是我们所说的进程栈。进程栈的初始化大小是由编译器和链接器计算出来的，但是栈的实时大小并不是固定的，Linux 内核会根据入栈情况对栈区进行动态增长（其实也就是添加新的页表）。但是并不是说栈区可以无限增长，它也有最大限制 RLIMIT_STACK (一般为 8M)，我们可以通过 ulimit 来查看或更改 RLIMIT_STACK 的值。

动态栈增长

进程在运行的过程中，通过不断向栈区压入数据，当超出栈区容量时，就会耗尽栈所对应的内存区域，这将触发一个 缺页异常 (page fault)。通过异常陷入内核态后，异常会被内核的 expand_stack() 函数处理，进而调用 acct_stack_growth() 来检查是否还有合适的地方用于栈的增长。

如果栈的大小低于 RLIMIT_STACK（通常为 8MB），那么一般情况下栈会被加长，程序继续执行，感觉不到发生了什么事情，这是一种将栈扩展到所需大小的常规机制。然而，如果达到了最大栈空间的大小，就会发生 栈溢出（stack overflow），进程将会收到内核发出的 段错误（segmentation fault）信号。

动态栈增长是唯一一种访问未映射内存区域而被允许的情形，其他任何对未映射内存区域的访问都会触发页错误，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。

线程栈

从 Linux 内核的角度来说，其实它并没有线程的概念。Linux 把所有线程都当做进程来实现，它将线程和进程不加区分的统一到了 task_struct 中。线程仅仅被视为一个与其他进程共享某些资源的进程，而是否共享地址空间几乎是进程和 Linux 中所谓线程的唯一区别。线程创建的时候，加上了 CLONE_VM 标记，这样 线程的内存描述符 将直接指向 父进程的内存描述符。

虽然线程的地址空间和进程一样，但是对待其地址空间的 stack 还是有些区别的。对于 Linux 进程或者说主线程，其 stack 是在 fork 的时候生成的，实际上就是复制了父亲的 stack 空间地址，然后写时拷贝 (cow) 以及动态增长。然而对于主线程生成的子线程而言，其 stack 将不再是这样的了，而是事先固定下来的，使用 mmap 系统调用，它不带有 VM_STACK_FLAGS 标记。

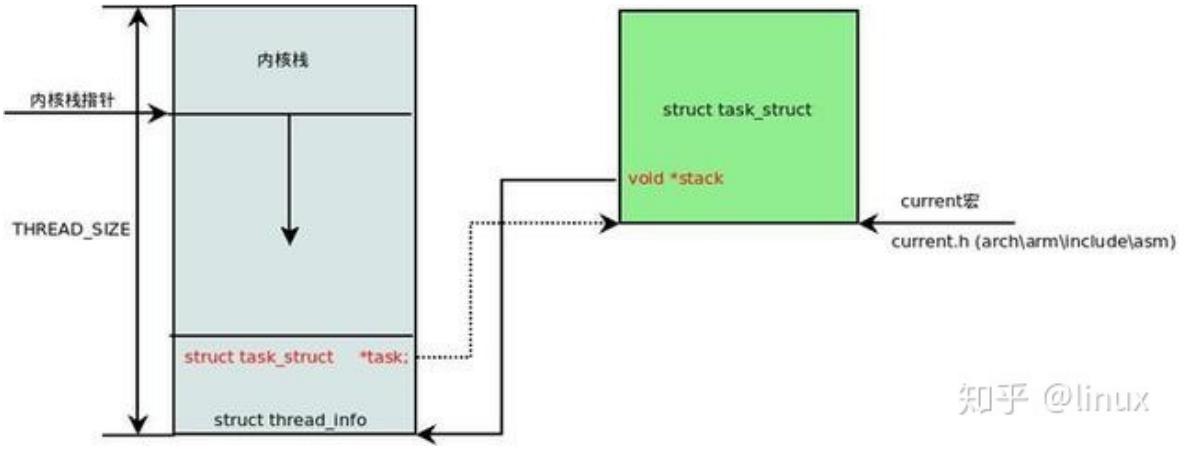
由于线程的 mm->start_stack 栈地址和所属进程相同，所以线程栈的起始地址并没有存放在 task_struct 中，应该是使用 pthread_attr_t 中的 stackaddr 来初始化 task_struct->thread->sp（sp 指向 struct pt_regs 对象，该结构体用于保存用户进程或者线程的寄存器现场）。这些都不重要，重要的是，线程栈不能动态增长，一旦用尽就没了，这是和生成进程的 fork 不同的地方。由于线程栈是从进程的地址空间中 map 出来的一块内存区域，原则上是线程私有的。但是同一个进程的所有线程生成的时候浅拷贝生成者的 task_struct 的很多字段，其中包括所有的 vma，如果愿意，其它线程也还是可以访问到的，于是一定要注意。（为了防止线程间的内存踩踏，在每个线程 mmap 出的线程栈之间都会预留空挡用于保护）

进程内核栈

在每一个进程的生命周期中，必然会通过到系统调用陷入内核。在执行系统调用陷入内核之后，这些内核代码所使用的栈并不是原先进程用户空间中的栈，而是一个单独内核空间的栈，这个称作进程内核栈。进程内核栈在进程创建的时候，通过 slab 分配器从 thread_info_cache 缓存池中分配出来，其大小为 THREAD_SIZE，一般来说是一个页大小 4K（应该是 8K）；

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

thread_union 进程内核栈 和 task_struct 进程描述符有着紧密的联系。由于内核经常要访问 task_struct，高效获取当前进程的描述符是一件非常重要的事情。因此内核将进程内核栈的头部一段空间，用于存放 thread_info 结构体，而此结构体中则记录了对应进程的描述符，两者关系如下图（对应内核函数为 dup_task_struct()）：



有了上述关联结构后，内核可以先获取到栈顶指针 esp，然后通过 esp 来获取 thread_info。这里有一个小技巧，直接将 esp 的地址与上 ~(THREAD_SIZE - 1) 后即可直接获得 thread_info 的地址。由于 thread_union 结构体是从 thread_info_cache 的 Slab 缓存池中申请出来的，而 thread_info_cache 在 kmem_cache_create 创建的时候，保证了地址是 THREAD_SIZE 对齐

的。因此只需要对栈指针进行 THREAD_SIZE 对齐，即可获得 thread_union 的地址，也就获得了 thread_union 的地址。成功获取到 thread_info 后，直接取出它的 task 成员就成功得到了 task_struct。其实上面这段描述，也就是 current 宏的实现方法：

```
register unsigned long current_stack_pointer asm ("sp");

static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)
                (current_stack_pointer & ~(THREAD_SIZE - 1));
}

#define get_current() (current_thread_info()->task)

#define current get_current()
```

Linux 内核栈空间连续

- 4K stacks by default?
- Expanding the kernel stack
- Kernel Small Stacks

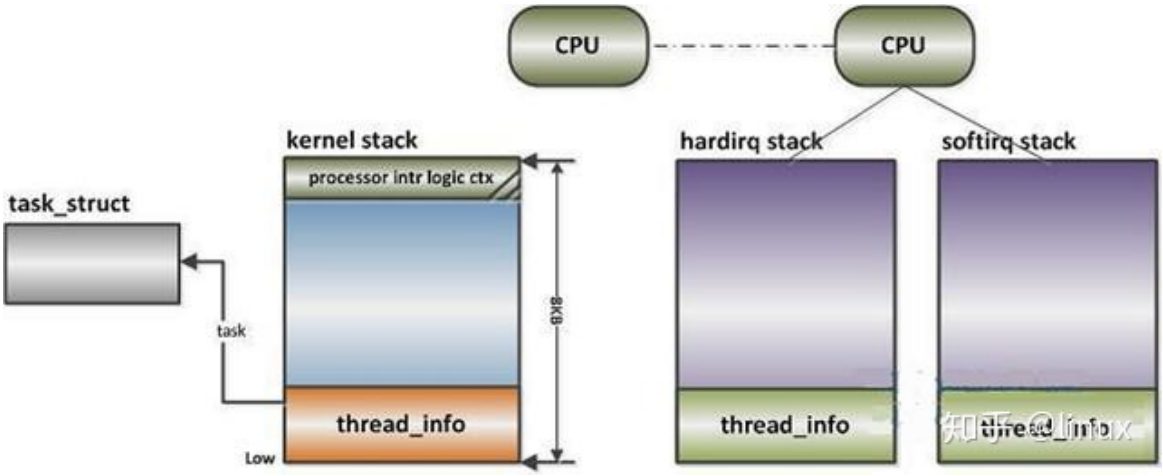
内核栈 8K，由两个连续的物理页框（order-1）组成，所以在内存碎片化严重的时候会发生进程创建失败，因为分不出连续页框了。之所以要设置为 8K 是因为 4K 可能会溢出。
（Expanding the kernel stack：As recently as 2008 some developers were trying to shrink the stack to 4KB, but that effort eventually proved to be unrealistic. Modern kernels can end up creating surprisingly deep call chains that just do not fit into a 4KB stack. ）

中断栈

进程陷入内核态的时候，需要内核栈来支持内核函数调用。中断也是如此，当系统收到中断事件后，进行中断处理的时候，也需要中断栈来支持函数调用（而不是中断嵌套）。由于系统中断的时候，系统当然是处于内核态的，所以中断栈是可以和内核栈共享的。但是具体是否共享，这和具体处理架构密切相关。

X86 上中断栈就是独立于内核栈的；独立的中断栈所在内存空间的分配发生在 arch/x86/kernel/irq_32.c 的 irq_ctx_init()函数中（如果是多处理器系统，那么每个处理器都会

有一个独立的中断栈），函数使用 __alloc_pages 在低端内存区分配 2 个物理页面，也就是 8KB 大小的空间。有趣的是，这个函数还会为 softirq 分配一个同样大小的独立堆栈。如此说来，softirq 将不会在 hardirq 的中断栈上执行，而是在自己的上下文中执行。



而 ARM 上中断栈和内核栈则是共享的；中断栈和内核栈共享有一个负面因素，如果中断发生嵌套，可能会造成栈溢出，从而可能会破坏到内核栈的一些重要数据，所以栈空间有时候难免会捉襟见肘。

Linux 为什么需要区分这些栈？

为什么需要区分这些栈，其实都是设计上的问题。这里就我看到过的一些观点进行汇总，供大家讨论：

1. 为什么需要单独的进程内核栈？所有进程运行的时候，都可能通过系统调用陷入内核态继续执行。假设第一个进程 A 陷入内核态执行的时候，需要等待读取网卡的数据，主动调用 schedule() 让出 CPU（本质就是每一次内核的栈帧不一定被完全完成并退出）；此时调度器唤醒了另一个进程 B，碰巧进程 B 也需要系统调用进入内核态。那问题就来了，如果内核栈只有一个，那进程 B 进入内核态的时候产生的压栈操作，必然会破坏掉进程 A 已有的内核栈数据；一但进程 A 的内核栈数据被破坏，很可能导致进程 A 的内核态无法正确返回到对应的用户态了；
2. 为什么需要单独的线程栈？Linux 调度程序中并没有区分线程和进程，当调度程序需要唤醒”进程”的时候，必然需要恢复进程的上下文环境，也就是进程栈；但是线程和父进程完全共享一份地址空间，如果栈也用同一个那就会遇到以下问题。假如进程的栈指针初始值为 0x7ffc80000000；父进程 A 先执行，调用了一些函数后栈指针 esp 为 0x7ffc8000FF00，此时父进程主动休眠了；接着调度器唤醒子线程 A1：

   此时 A1 的栈指针 esp 如果为初始值 0x7ffc80000000，则线程 A1 一但出现函数调用，必然会破坏父进程 A 已入栈的数据。如果此时线程 A1 的栈指针和父进程最后更新的值一致，esp 为 0x7ffc8000FF00，那线程 A1 进行一些函数调用后，栈指针 esp 增加到 0x7ffc8000FFFF，然后线程 A1 休眠；调度器再次换成父进程 A 执行，那这个时候父进程的栈指针是应该为 0x7ffc8000FF00 还是 0x7ffc8000FFFF 呢？无论栈指针被设置到哪个值，都会有问题不是吗？

3. 进程和线程是否共享一个内核栈？No，线程和进程创建的时候都调用 dup_task_struct 来创建 task 相关结构体，而内核栈也是在此函数中 alloc_thread_info_node 出来的。因此虽然线程和进程共享一个地址空间 mm_struct，但是并不共享一个内核栈。（内核栈与 tast_struct 相关联）
4. 为什么需要单独中断栈？这个问题其实不对，ARM 架构就没有独立的中断栈。

进程状态切换(创建、就绪、阻塞、运行、终止)

引起进程状态转换的具体原因如下：

运行态 →阻塞态：等待使用资源；如等待外设传输；等待人工干预。

阻塞态 →就绪态：资源得到满足；如外设传输结束；人工干预完成。

运行态 →就绪态：运行时间片到；出现有更高优先权进程。

就绪态 →运行态：CPU 空闲时选择一个就绪进程。

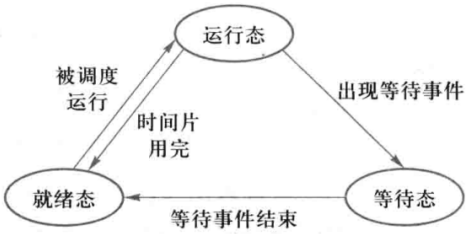


图 2-4　进程三态模型及其状态转换

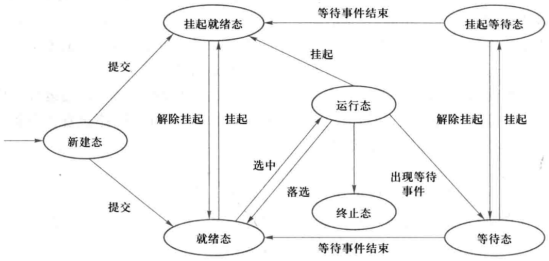


图 2-5　具有挂起功能的系统的进程状态及其转换示意图

三态模型和五态模型都是假设所有进程都在内存中的事实上有序不断的创建进程，当系统资源尤其是内存资源已经不能满足进程运行的要求时，必须把某些进程挂起（suspend），对换到磁盘对换区中，释放它占有的某些资源，暂时不参与低级调度。起到平滑系统操作负荷的目的。

引起进程挂起的原因是多样的，主要有：

1. 终端用户的请求。当终端用户在自己的程序运行期间发现有可疑问题时，希望暂停使自己的程序静止下来。亦即，使正在执行的进程暂停执行；若此时用户进程正处于就绪状态而未执行，则该进程暂不接受调度，以便用户研究其执行情况或对程序进行修改。我们把这种静止状态成为“挂起状态”。
2. 父进程的请求。有时父进程希望挂起自己的某个子进程，以便考察和修改子进程，或者协调各子进程间的活动。
3. 负荷调节的需要。当实时系统中的工作负荷较重，已可能影响到对实时任务的控制时，可由系统把一些不重要的进程挂起，以保证系统能正常运行。
4. 操作系统的需要。操作系统有时希望挂起某些进程，以便检查运行中的资源使用情况或进行记账。
5. 对换的需要。为了缓和内存紧张的情况，将内存中处于阻塞状态的进程换至外存上。

线程模型

- 对 模型： 般直接使用 API 或系统调用创建的线程均为 对 的线程。 个用户使用的线程就唯 对应 个内核使用的线程。
  - 优点：在多核处理器的硬件的 持下，内核空间线程模型 持了真正的并行，当 个线程被阻塞后， 允许另 个线程继续执行，所以并发能 较强。
  - 缺点：每创建 个用户级线程都需要创建 个内核级线程与其对应，这样创建线程的开销比较大，会影响到应用程序的性能。
- 多对 模型：多对 模型将多个用户线程映射到 个内核线程上，线程之间的切换由用户的代码来进行。
  - 优点：因此相对于 对 模型，多对 模型的线程切换要快速许多。
  - 缺点：多对 模型 大问题是，如果其中 个用户线程阻塞，那么所有的线程将都无法执行，因为此时内核里的线程也会随之阻塞。另外，在多处理器系统上，处理器的增多线程性能也不会有明显的帮 助。但同时，多对 模型得到的好处是高效的上下文切换和 乎无限制的线程数量。
- 多对多模型：结合了多对 模型和 对 模型的特点，将多个用户线程映射到少数但不止 个内核线程上。

用户级别线程和内核级别线程的区别和联系

- 本质：
  - 用户级线程的实现就是把整个线程实现部分放在用户空间中，内核对线程 无所知，内核看到的就是 个单线程进程。
  - 内核线程：由操作系统内核创建和撤销。内核维护进程及线程的上下文信息以及线程切换。 个内核线程由于 I/O 操作而阻塞，不会影响其它线程的运行。
- 调度：
  - 对于用户级线程，操作系统内核不可感知，调度需要由开发者自己实现
  - 内核级线程则与之相反，将调度全权交由操作系统内核来完成。
- 开销：在用户空间创建线程的开销相比之下会比内核空间小很多。
- 性能：用户级线程的切换发生在用户空间，这样的线程切换 少比陷入内核要快 个数量级，不需要陷入内核、不需要上下文切换，这就使得线程调度非常快。

进程加载过程

在进程创建的过程中，程序内容被映射到进程的虚拟内存空间，为了让 个很大的程序在有限的物理内存空间运行，把这个程序的开始部分先加载到物理内存空间运行，因为操作系统处理的是进程的虚拟地址，如果在进行虚拟到物理地址的转换工程中，发现要访问的页面不在内存时，这个时候就会发生缺页异常(nopage)，接着操作系统就会把磁盘上还没有加载到内存中的

数据加载到物理内存中，对应的进程页表进行更新。

虚拟内存

为了防止不同进程同　时刻在物理内存中运行而对物理内存的争夺，采用了虚拟内存。

虚拟内存技术使得不同进程在运行过程中，它所看到的是自己独自占有了当前系统的内存。所有进程共享同　物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，等到运行到对应的程序时，才会通过缺页中断，来拷贝数据。

缺页中断

在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存时，会产生　次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的　页，将其调入内存。

中断的四个步骤：1 保护 cpu 现场，2 分析中断原因，3 转入缺页中断处理程序进行处理，4 恢复 cpu 现场，继续执行。

页面调度的算法 LRU、LFU、先进先出算法

页表寻址

操作系统为每　个进程维护了　个从虚拟地址到物理地址的映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的　个映射。页表中的每　项都记录了页表号和页内偏移。CPU 中有　个页表寄存器，里面存放着当前进程页表的起始地址和页表长度，将计算的页表号和页表长度进行对比，确认在页表范围内，然后将页表号和页表项长度相乘，得到目标页相对于页表基地址的偏移量，最后加上页内偏移量就可以访问到相对应物理地址了

CPU 访问的虚拟地址：A 、页面大小：L 、页表号：(A/L) 、页内偏移：(A%L)

写时复制

系统调用 fork() 创建了父进程的　个复制，以作为子进程。传统上，fork() 为子进程创建　个父进程地址空间的副本，复制属于父进程的页面。然而，考虑到许多子进程在创建之后立即调用系统调用 exec()，父进程地址空间的复制可能没有必要。

因此，可以采用　种称为写时复制的技术，它通过允许父进程和子进程最初共享相同的页面来工作。这些共享页面标记为写时复制，这意味着如果任何　个进程写入共享页面，那么就创建共享页面的副本。

死锁

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的相互等待的现象。死锁产生的四个条件：

1. 互斥：一个资源每次只能被一个进程使用。
2. 请求和保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不可剥夺：进程已获得的资源，在未使用完之前，不能强行剥夺。
4. 环路等待：若干进程之间形成一种头尾相接的循环等待资源关系。

死锁解决方法：

1. 资源　次性分配：从而剥夺请求和保持条件
2. 可剥夺资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件
3. 资源有序分配法：系统给每类资源赋予　个序号，每个进程按编号递增的请求资源，释放则相反，从而破坏环路等待的条件

可重入锁

重入锁实现可重入性原理或机制是：每　个锁关联　个线程持有者和计数器，当计数器为 0 时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某　线程请求成功后，会记下锁的持有线程，并且将计数器置为 1；此时其它线程请求该锁，则必须等待；而该持有锁的线程如果再次请求这个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数 器会递减，如果计数器为 0，则释放该锁。

linux 锁机制

- 互斥锁：mutex，用于保证在任何时刻，都只能有　个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒
- 读写锁：处于读操作时，可以允许多个线程同时获得读操作。但是同　时刻只能有　个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。
- 自旋锁：在任何时刻同样只能有　个线程访问对象。但是当获取锁操作失败时，不会进入睡眠，而是会在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的消耗，在加锁时间短暂的环境下会极大的提高效率。但如果加锁时间过长，则会非常浪费 CPU 资源。

线程池的实现（互斥量和条件变量）

1. 设置　个生产者消费者队列，作为临界资源
2. 初始化 n 个线程，并让其运行起来，加锁去队列取任务运行
3. 当任务队列为空的时候，所有线程阻塞
4. 当生产者队列来了　个任务后，先对队列加锁，把任务挂在到队列上，然后使用条件变量去通知阻塞中的　个线程

阻塞 io 和非阻塞 io

- 阻塞 IO，指的是需要内核 IO 操作彻底完成后，才返回到用户空间执行用户的操作。
- 非阻塞 IO，指的是用户空间的程序不需要等待内核 IO 操作彻底完成，可以立即返回用户空间执行用户操作，// 即处于非阻塞的状态，与此同时内核会立即返回给用户　个状态值。

内核态和用户态

因为操作系统的资源是有限的，如果访问资源的操作过多，必然会消耗过多的资源，而且如果不对这些操作加以区分，很可能造成资源访问的冲突。所以，为了减少有限资源的访问和使用冲突，对不同的操作赋予不同的执行等级，就是所谓特权的概念。与系统相关的　些特别关键的操作必须由最高特权的程序来完成。运行于用户态的进程可以执行的操作和访问的资源都会受到极大的限制，而运行在内核态的进程则可以执行任何操作并且在资源的使用上没有限制。很多程序开始时运行于用户态，但在执行的过程中，　些操作需要在内核权限下才能执行，这就涉及到　个从用户态切换到内核态的过程。

用户态和核心态（内核态）之间的区别是什么呢？ 权限不　样。

- 用户态的进程能存取它们自己的指令和数据，但不能存取内核指令和数据（或其他进程的指令和数据）。
- 内核态下的进程能够存取特权指令，在用户态下执行特权指令会引起错误。

用户态到内核态：中断、系统调用、异常（缺页异常）

内核抢占

内核抢占机制(preempt)

linux 内核抢占那些事

Linux2.4 不支持内核抢占，即是说进程通过系统调用陷入到内核态的时候，不可以被其他的进程抢占。如果有更高优先级的进程，只有在系统调用返回用户空间的时候，才可被调度程序调

度，由高优先级的进程占用 cpu。这里的“不可以被其他进程抢占”当然不包括中断上下文，无论内核态还是用户态，中断上下文都可以抢占进程上下文，中断上下文是拥有最高的权限，它甚至可以抢占其他的中断上下文。

Linux2.6 有一个 CONFIG_PREEMPT 的选项，打开该选项后，linux kernel 就支持了内核代码的抢占。对于抢占式内核而言，即便是从中断上下文返回内核空间的进程上下文，只要内核代码不在临界区内，就可以发生调度，让最高优先级的任务调度执行。例如高优先级的进程可以抢占内核态的系统调用，而不必等系统调用执行完返回用户空间才抢占。

早期的 linux 内核是不支持抢占的，这样就会存在两个问题：

1. 2.6 内核之前一个进程从用户态进入内核后，别的进程只有等到它退出内核态才能有机会得到执行，这样就会存在时延问题；
2. 可能存在优先级反转的问题，例如一个低优先级的用户进程由于执行中断等原因是一个高优先级的任务得不到及时的响应。

简而言之，抢占就是为了让更高优先级的任务得到及时调度。

禁止内核抢占的情况

1. 内核执行中断处理例程时不允许内核抢占，中断返回时再执行内核抢占。
2. 当内核执行软中断或 tasklet 时，禁止内核抢占，软中断返回时再执行内核抢占。
3. 在临界区禁止内核抢占，临界区保护函数通过抢占计数宏控制抢占，计数大于 0，表示禁止内核抢占。

为保证 Linux 内核在以上情况下不会被抢占，抢占式内核使用了一个变量 preempt_ count，称为内核抢占锁。这一变量被设置在进程的 PCB 结构 task_struct 中。每当内核要进入以上几种状态时，变量 preempt_ count 就加 1，指示内核不允许抢占。每当内核从以上几种状态退出时，变量 preempt_ count 就减 1，同时进行可抢占的判断与调度。

调度的时机

抢占情况下：

1. 中断返回内核空间：检查 preempt_count 是否为 0 和 TIF_NEED_RESCHED 是否为 1
2. 中断或异常返回到 user space：检查 TIF_NEED_RESCHED 是否为 1
3. 显式或者隐式调 preempt_enable()函数：检查 preempt_count 是否为 0 和 TIF_NEED_RESCHED 是否为 1
4. 使能软中断：检查 preempt_count 是否为 0 和 TIF_NEED_RESCHED 是否为 1
5. 自己主动 schedule()

非抢占情况下：

1. 通过 schedule()函数自愿地启动一次调度（hy:或者阻塞、睡眠时）
2. 每次从内核态返回到用户态时。

中断、异常和系统调用

中断、异常、和系统调用的区别总结

异常与过程调用的区别

1. 返回地址，过程调用一定是将返回地址(逻辑流下一条指令)压栈，而异常可能是下一条指令，可能是当前指令。
2. 处理器会把状态字寄存器压栈，而过程调用不会（这里考研经常考）
3. 异常流处理运行在内核模式下
4. 如果控制从用户程序转到内核，所有的这些项目都被压倒内核栈中。

异常流的分类：

类别	原因	异步/同步	返回行为
中断(interrupt)	来自处理机外部设备的信号(e.g. IO)	异步	总是下一条指令
陷阱(trap)	有意的异常	同步	总是下一条指令
故障(fault)	潜在可恢复错误	同步	可能返回当前指令
终止(abort)	不可恢复错误	同步	杀死程序，不返回

- 中断：这里的中断是特指来自外部的信号中断当前指令序列的执行。通常国内教材只有在特指外部中断时才指的是这个，而内部异常通常指的就是后三种.
- 陷阱：陷阱是有意的异常，重要的用途就是在用户程序和内核之间提供像过程调用一样的接口，称为系统调用，让用户程序能够访问操作系统提供服务，所以系统调用是陷阱的一种，同时陷阱只是内部异常的一种。在 IA-32 中陷阱指令有 INT n, int 3,into, bound etc.
- 故障：故障由错误引起，常见的比如 segment fault, 缺页异常，如果系统能够处理，那就会回到当前指令继续执行，否则终止程序。
- 终止：就是不可恢复了。

面试考点——中断和异常的区别

可以把中断分为外中断和内中断：

- 外中断：就是我们指的中断，是指由于外部设备事件所引起的中断，如通常的磁盘中断、打印机中断等；
- 内中断：就是异常，是指由于 CPU 内部事件所引起的中断，如程序出错(非法指令、地址越界)。内中断(trap)也被译为“捕获”或“陷入”。

异常是由于执行了现行指令所引起的。由于系统调用引起的中断属于异常。

中断则是由于系统中某事件引起的，该事件与现行指令无关。

- 相同点：都是 CPU 对系统发生的某个事情做出的一种反应。
- 区别：中断由外因引起，异常由 CPU 本身原因引起。

系统中寄存器种类

CPU中的主要寄存器：有六类寄存器：指令寄存器（IR）、程序计数器（PC）、地址寄...

在 CPU 中至少要有六类寄存器：指令寄存器（IR）、程序计数器（PC）、地址寄存器（AR）、数据寄存器（DR）、累加寄存器（AC）、程序状态字寄存器（PSW）。

1. 数据寄存器

数据寄存器（Data Register，DR）又称数据缓冲寄存器，其主要功能是作为 CPU 和主存、外设之间信息传输的中转站，用以弥补 CPU 和主存、外设之间操作速度上的差异。

数据寄存器用来暂时存放由主存储器读出的一条指令或一个数据字；反之，当向主存存入一条指令或一个数据字时，也将它们暂时存放在数据寄存器中。

数据寄存器的作用是 ：

（1）作为 CPU 和主存、外围设备之间信息传送的中转站；

（2）弥补 CPU 和主存、外围设备之间在操作速度上的差异；

（3）在单累加器结构的运算器中，数据寄存器还可兼作操作数寄存器。

2. 指令寄存器

指令寄存器（Instruction Register，IR）用来保存当前正在执行的一条指令。

当执行一条指令时，首先把该指令从主存读取到数据寄存器中，然后再传送至指令寄存器。

指令包括操作码和地址码两个字段，为了执行指令，必须对操作码进行测试，识别出所要求的操作，指令译码器（Instruction Decoder，ID）就是完成这项工作的。指令译码器对指令寄存器的操作码部分进行译码，以产生指令所要求操作的控制电位，并将其送到微操作控制线路上，在时序部件定时信号的作用下，产生具体的操作控制信号。

指令寄存器中操作码字段的输出就是指令译码器的输入。操作码一经译码，即可向操作控制器发出具体操作的特定信号。

3. 程序计数器

程序计数器（Program Counter，PC）用来指出下一条指令在主存储器中的地址。

在程序执行之前，首先必须将程序的首地址，即程序第一条指令所在主存单元的地址送入 PC，因此 PC 的内容即是从主存提取的第一条指令的地址。

当执行指令时，CPU 能自动递增 PC 的内容，使其始终保存将要执行的下一条指令的主存地址，为取下一条指令做好准备。若为单字长指令，则(PC)+1àPC，若为双字长指令，则(PC)+2àPC，以此类推。

但是，当遇到转移指令时，下一条指令的地址将由转移指令的地址码字段来指定，而不是像通常的那样通过顺序递增 PC 的内容来取得。

因此，程序计数器的结构应当是具有寄存信息和计数两种功能的结构。

4. 地址寄存器

地址寄存器（Address Register，AR）用来保存 CPU 当前所访问的主存单元的地址。

由于在主存和 CPU 之间存在操作速度上的差异，所以必须使用地址寄存器来暂时保存主存的地址信息，直到主存的存取操作完成为止。

当 CPU 和主存进行信息交换，即 CPU 向主存存入数据/指令或者从主存读出数据/指令时，都要使用地址寄存器和数据寄存器。

如果我们把外围设备与主存单元进行统一编址，那么，当 CPU 和外围设备交换信息时，我们同样要使用地址寄存器和数据寄存器。

5. 累加寄存器

累加寄存器通常简称累加器（Accumulator，AC），是一个通用寄存器。

累加器的功能是：当运算器的算术逻辑单元 ALU 执行算术或逻辑运算时，为 ALU 提供一个工作区，可以为 ALU 暂时保存一个操作数或运算结果。

显然，运算器中至少要有一个累加寄存器。

6. 程序状态字寄存器

程序状态字（Program Status Word，PSW）用来表征当前运算的状态及程序的工作方式。

程序状态字寄存器用来保存由算术/逻辑指令运行或测试的结果所建立起来的各种条件码内容，如运算结果进/借位标志（C）、运算结果溢出标志（O）、运算结果为零标志（Z）、运算结果为负标志（N）、运算结果符号标志（S）等，这些标志位通常用 1 位触发器来保存。

除此之外，程序状态字寄存器还用来保存中断和系统工作状态等信息，以便 CPU 和系统及时了解机器运行状态和程序运行状态。

因此，程序状态字寄存器是一个保存各种状态条件标志的寄存器。

系统调用与库函数的区别

系统调用(System call) 是程序向系统内核请求服务的方式。可以包括硬件相关的服务(例如，访问硬盘等)，或者创建新进程，调度其他进程等。系统调用是程序和操作系统之间的重要接

口。

库函数：把 一些常用的函数编写完放到 个文件里，编写应用程序时调用，这是由第三方提供的，发生在用户地址空间。

- 在移植性方面，不同操作系统的系统调用 般是不同的，移植性差；
- 在调用开销方面，系统调用需要在用户空间和内核环境间切换，开销较大；而库函数调用属于“过程调用”，开销较小。

守护、僵尸、孤儿进程的概念

- 守护进程：运行在后台的 种特殊进程，独立于控制终端并周期性地执行某些任务。
- 僵尸进程： 个进程 fork 子进程，子进程退出，而父进程没有 wait/waitpid 子进程，那么子进程的进程描述符仍保存在系统中，这样的进程称为僵尸进程。
- 孤儿进程： 个父进程退出，而它的 个或多个子进程还在运行，这些子进程称为孤儿进程。孤儿进程将由 init（1 号） 进程收养并对它们完成状态收集工作。每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init ，而 init 进程会循环地 wait() 它的已经退出的子进程。

中断

中断是指 CPU 对系统发生的某个事件做出的 种反应，CPU 暂停正在执行的程序，保存现场后自动去执行相应的处理程序，处理完该事件后再返回中断处继续执行原来的程序。中断 般分为三类，

- 种是由 CPU 外部引起的，称为外中断。如 I/O 中断、时钟中断，
- 种是来自 CPU 内部事件或程序执行中引起的中断，例如程序非法操作，地址越界、浮点溢出）称为内中断，或者（异常，陷入），
- 最后 种是在程序中使用了系统调用引起的。

而中断处理 般分为中断响应和中断处理两个步骤，中断响应由硬件实施，中断处理主要由软件实施。

中断处理程序的处理过程：

1. 测定是否有未响应的中断信号。
2. 保护被中断进程的 CPU 环境。
3. 转入相应的设备处理程序。
4. 中断处理。
5. 恢复 CPU 的现场并退出中断。

Linux 不允许中断嵌套

genirq: Run irq handlers with interrupts disabled

Running interrupt handlers with interrupts enabled can cause stack overflows.

中断的上半部和下半部

linux kernel的中断子系统之（八）：softirq

对于中断处理而言，linux 将其分成了两个部分，一个叫做中断 handler（top half），是全程关闭中断的，另外一部分是 deferable task（bottom half），属于不那么紧急需要处理的事情。在执行 bottom half 的时候，是开中断的。有多种 bottom half 的机制，例如：softirq、tasklet、workqueue 或是直接创建一个 kernel thread 来执行 bottom half（这在旧的 kernel 驱动中常见，现在，一个理智的 driver 厂商是不会这么做的）。

为何有 top half 和 bottom half

在把中断处理过程划分成 top half 和 bottom half 之后，关中断的 top half 被瘦身，可以非常快速的执行完毕，大大减少了系统关中断的时间，提高了系统的性能。

例子见原文

为何有 softirq 和 tasklet

现在的 linux kernel 提供了三种 bottom half 的机制，来应对不同的需求。

workqueue 和 softirq、tasklet 有本质的区别：workqueue 运行在 process context，而 softirq 和 tasklet 运行在 interrupt context。因此，出现 workqueue 是不奇怪的，在有 sleep 需求的场景中，defering task 必须延迟到 kernel thread 中执行，也就是说必须使用 workqueue 机制。

softirq 更倾向于性能，而 tasklet 更倾向于易用性。

为了性能，同一类型的 softirq 有可能在不同的 CPU 上并发执行，这给使用者带来了极大的痛苦，因为驱动工程师在撰写 softirq 的回调函数的时候要考虑重入，考虑并发，要引入同步机制。但是，为了性能，我们必须如此。

如果一个 tasklet 在 processor A 上被调度执行，那么它永远也不会同时在 processor B 上执行，也就是说，tasklet 是串行执行的（注：不同的 tasklet 还是会并发的），不需要考虑重入的问题。

操作系统常见的进程调度方法

1. 先来先去服务

2. 时间片轮转法
3. 多级反馈队列算法
4. 最短进程优先
5. 最短剩余时间优先
6. 最高响应比优先 根据比率：R=(w+s)/s （ R 为响应比，w 为等待处理的时间，s 为预计的服务时间）

多级反馈队列算法

1. 设置多个就绪队列，并为各个队列赋予不同的优先级。在优先权越高的队列中，为每个进程所规定的执行时间片就越小。
2. 当 个新进程进入内存后， 先放入第 队列的末尾，按照先来先去原则排队等候调度。如果他能 在 个时间片中完成，便可撤离；如果未完成，就转入第 队列的末尾，同样等待调度 如此下去，当 个长作业（进程）从第 队列依次将到第 n 队列（最后队列）后，便按第 n 队列时间片轮转运行。
3. 仅当第 队列空闲的时候，调度程序才调度第 队列中的进程运行；仅当第 1 到（i-1）队列空时， 才会调度第 i 队列中的进程运行，并执行相应的时间片轮转。
4. 如果处理机正在处理第 i 队列中某进程， 有新进程进入优先权较高的队列，则此新队列抢占正在运行的处理机，并把正在运行的进程放在第 i 队列的队尾。

操作系统内存分配

1. 连续分配方式（ 外部碎片)
2. 基本分页存储管理方式。（ 内部碎片 ）
3. 基本分段存储管理方式. (外部碎片)
4. 段页式存储管理方式

基本分页存储管理方式

分页存储管理是将 个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页，并为各页加以 编号，从 0 开始，如第 0 页、第 1 页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理) 块或页框(frame)，也同样为它们加以编号，如 0#块、1#块等等。在为进程分配内存时， 以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后 页经常装不满 块而形成了不可利用的碎片，称之为“ 页内碎片“

基本分段存储管理方式

分段存储管理方式的目的，主要是为了满足用户（程序员）在编程和使用上多方面的要求，其中有些要求是其他 种存储管理方式所难以满足的。因此，这种存储管理方式已成为当今所有存储管理方式的基础。

1. 方便编程
2. 信息共享：分页系统中的“页”只是存放信息的物理单位（块），并无完整的意义，不便于实现共享；然而段却是信息的逻辑单位。由此可知，为了实现段的共享，希望存储器管理能与用户程序分段的组织方式相适应。
3. 信息保护
4. 动态增长
5. 动态链接

分页和分段的主要区别

- 两者相似之处：两者都采用离散分配方式，且都要通过地址映射机构来实现地址变换。
- 两者不同之处：
  - 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。 或者说，分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它含有 组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。
  - 页的大小固定且由系统决定，而段的长度却不固定。

缓存（cache）与缓冲（buffer）的主要区别

- Buffer 的核心作用是用来缓冲，缓和冲击。比如每秒要写 100 次硬盘，对系统冲击很大，浪费了大量时间在忙着处理开始写和结束写这两件事嘛。用个 buffer 暂存起来，变成每 10 秒写 次硬盘，对系统的冲击就很小，写入效率高了，极大缓和了冲击。
- Cache 的核心作用是加快取用的速度。比如 个很复杂的计算做完了，下次还要用结果，就把结果放手边 个好拿的地方存着，下次不用再算了。加快了数据取用的速度。

~~简单来说就是 buffer 偏重于写，而 cache 偏重于读。~~

swap 区

swap 分区通常被称为交换分区，这是 块特殊的硬盘空间，即当实际内存不够用的时候，操作系统会从内存中取出 部分暂时不用的数据，放在交换分区中，从而为当前运行的程序腾出足够的内存空间。

使用 swap 交换分区，显著的优点是，通过操作系统的调度，应用程序实际可以使用的内存空间将远远超过系统的物理内存。但是频繁地读写硬盘，会显著降低操作系统的运行速度，这也是使用 swap 交换分区最大的限制。

【譯】替 swap 辯護：常見的誤解

- 交换区是允许公平地回收内存的有用工具，但是它的目的经常被人误解，导致它在业内这种负面声誉。如果 你是按照原本的目的使用交换区的话—作爲增加内存回收公平性的方式—你会发现它是很有效的工具而不是阻碍。
- 禁用交换区并不能在内存竞争的时候防止磁盘 I/O 的问题，它只不过把匿名页面的磁盘 I/O 变成了文件页面的 磁盘 I/O。这不仅更低效，因爲我们回收内存的时候能选择的页面范围更小了，而且它可能是导致高度内存竞争 状态的元凶。
- 有交换区会导致系统更慢地使用 OOM 杀手，因爲在缺少内存的情况下它提供了另一种更慢的内存， 会持续地内存颠簸（强于 OOM）—内核调用 OOM 杀手只是最后手段，会晚于所有事情已经被搞得一团糟之后。 解决方案取决于你的系统：
  - 你可以预先更具每个 cgroup 的或者系统全局的内存压力改变系统负载。这能防止我们最初进入内存竞争 的状态，但是 Unix 的历史中一直缺乏可靠的内存压力检测方式。希望不久之后在有了 缺页检测 这样的性能指标之后能改善这一点。
  - 你可以使用 memory.low 让内核不倾向于回收（进而交换）特定一些 cgroup 中的进程， 允许你在不禁用交换区的前提下保护关键后台服务。

slab 分配器

slab 分配器有以下三个基本目标

1. 减少伙伴算法在分配小块连续内存时所产生的内部碎片；
2. 将频繁使用的对象缓存起来，减少分配、初始化和释放对象的时间开销。slab 分配器并不丢弃已经分配的对象，而是释放并把它们保存在内存中。slab 分配对象时，会使用最近释放的对象的内存块， 因此其驻留在 cpu 高速缓存中的概率会大大提高。(slab 三个链表 full partial empty)
3. 通过着色技术调整对象以更好的使用硬件高速缓存(cache)；

slab 染色

问题：缓存颠簸（乒乓效应），即 cpu 读取内存里的东西时，并不会直接去内存去读取，这样会导致读取的数据很慢。cpu 会到　级缓存读取所需要的数据，而　级缓存则会去内存里面读取数据，读取的方式是通过缓存行(cache line) 的形式来进行读取。当　级缓存内的数据需要置换时，则会将缓存内的数据置换到　级缓存内，然后依次类推到内存中。高速缓存和物理内存的映射是固定的，根据高速缓存的大小进行映射。如果持续读两个内存地址不同但映射到相同 cache line 时候就需要频繁的换出。

解决的方法：将第　块读取的数据前加　个偏移，让它移到第 1 块缓存行上面，两块数据分别缓存在 cache line 的不同位置，那么读取数据的时候就不会造成不必要的数据交换。着色即为添加偏移

出于对内存使用率的极致追求，slub 去除了 slab 的着色做法，取而代之是 slub 复用，通过 slub 复用减轻 cache 冲突的情况。

slab （slub）分配回收过程

内核内存分配器SLAB和SLUB

slab slob slub

Slob, Slab VS Slub

- Slab is the original, based on Bonwick's seminal paper and available since Linux kernel version 2.2. It is a faithful implementation of Bonwick's proposal, augmented by the multiprocessor changes described in Bonwick's follow-up paper2.
- Slub is the next-generation replacement memory allocator, which has been the default in the Linux kernel since 2.6.23. It continues to employ the basic "slab" model, but fixes several deficiencies in Slab's design, particularly around systems with large numbers of processors. Slub is simpler than Slab.
- SLOB (Simple List Of Blocks) is a memory allocator optimized for embedded systems with very little memory–on the order of megabytes. It applies a very simple first-fit algorithm on a list of blocks, not unlike the old K&R-style heap allocator. In eliminating nearly all of the overhad from the memory allocator, SLOB is a good fit for systems under extreme memory constraints, but it offers none of the benefits described in 1 and can suffer from pathological fragmentation.

slab slub slob 源代码分析

深入理解Linux内存管理（八）slab，slob和slub介绍

slub 详解（必读）

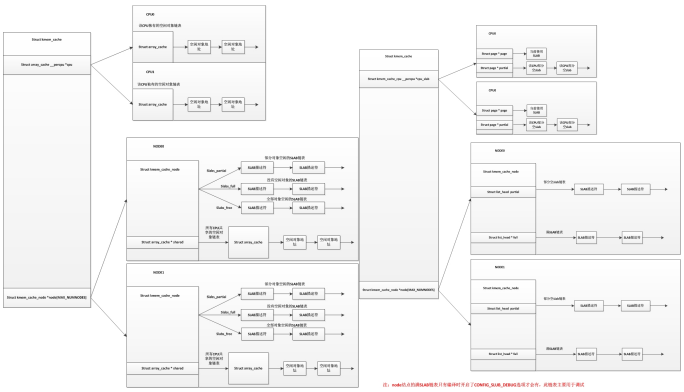
linux内存源码分析 - SLUB分配器概述

我们先说说 slab 分配器的弊端，我们知道 slab 分配器中每个 node 结点有三个链表，分别是空闲 slab 链表，部分空 slab 链表，已满 slab 链表，这三个链表中维护着对应的 slab 缓冲区。我们也知道 slab 缓冲区的内存是从伙伴系统中申请过来的，我们设想一个情景，如果没有内存回收机制的情况下，只要申请的 slab 缓冲区就会存入这三个链表中，并不会返回到伙

伴系统里，如果这个类型的 SLAB 迎来了一个分配高峰期，将会从伙伴系统中获取很多页面去生成许多 slab 缓冲区，之后这些 slab 缓冲区并不会自动返回到伙伴系统中，而是会添加到 node 结点的这三个 slab 链表中去，这样就会有很多 slab 缓冲区是很少用到的。

而 slub 分配器把 node 结点的这三个链表精简为了一个链表，只保留了部分空 slab 链表，而 SLUB 中对于每个 CPU 来说已经不使用空闲对象链表，而是直接使用单个 slab，并且每个 CPU 都维护有自己的一个部分空链表。在 slub 分配器中，对于每个 node 结点，也没有了所有 CPU 共享的空闲对象链表。

左 slab，右 slub



TODO：slub 分配过程总结

SIGSTOP 和 SIGKILL 不能被捕获或者忽略。

fork、vfork 和 clone 的区别

fork() 函数

- 调用成功：返回两个值；
  - 父进程：返回子进程的 PID；
  - 子进程：返回 0；
- 失败：返回-1；

fork 创造的子进程复制了父亲进程的资源（写时复制技术），包括内存的内容，task_struct 内容（2 个进程的 pid 不同）。进程被存放在　个叫做任务队列的双向循环链表当中，这里是资源的复制不是指针的复制

vfork 是　个过时的应用，vfork 也是创建　个子进程，但是子进程共享父进程的空间。在 vfork 创建子进程之后，父进程阻塞，直到子进程执行了 exec() 或者 exit()。vfork 最初是因为 fork 没有实现 COW 机制，而很多情况下 fork 之后会紧接着 exec，而 exec 的执行相当于之前 fork 复制的空间全部变成了无用功，所以设计了 vfork。而现在 fork 使用了 COW 机制，唯的代价仅仅是复制父进程页表的代价，所以 vfork 不应该出现在新的代码之中。

clone 是 fork 的升级版本，不仅可以创建进程或者线程，还可以指定创建新的命名空间（namespace）、有选择的继承父进程的内存、甚 可以将创建出来的进程变成父进程的兄弟进程等等。clone 和 fork 最大不同在于 clone 不再复制父进程的栈空间，而是自己创建 个新的。

Linux 操作系统的开机流程详解

开机需要十步

- 第 步：开机自检（BIOS）就是开始工作之前先对自己的工具进行检查是否正常，如果正常那就可以进行接下来的步骤假如步正常就得检测哪里的问题进行处理。BIOS 其实就是主板上的 给自检程序， 开机先对主板上自带的和外接的 些开机必备的设备进行检测，像 CPU，显卡，内存，硬盘等设备的自检过程就是自检
- 第 步：MBR 引导，也就是根据装有 linux 系统的硬盘上的主引导区的记录进行引导，主引导记录处在硬盘上的第 个物理分区上，硬盘能够读取到数据也就是靠的这个最主要的 MBR 主引导记录，假如这给 512 字节丢失那这张硬盘就无法工作。它里面包含了硬盘的主引导程序和硬盘的分区表，分区表有四个分区记录每个分区占 16 个字节共 64 个字节，还有 446 字节放主引导程序，2 字节用作校验。
- 第三步：GRUB 菜单，也就是操作系统引导菜单。GRUB（GRand Unified Bootloader 简称“GRUB”）是一个来自 GNU 项目的多操作系统启动程序。GRUB 是多启动规范的实现，它允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。GRUB 可用于选择操作系统分区上的不同内核，也可用于向这些内核传递启动参数。
- 第四步：加载内核（kernel），也就是启动操作系统的核心
- 第五步：运行 INIT 进程，init 也就是主进程，它的 PID 号是 1 也就是第 个被运行的进程
- 第六步：读取/etc/inittab 配置文件，也就是 linux 开机时默认的启动模式。在/etc/inittab 这给配置文件下可以修改开机默认启动选项
- 第七步：执行/etc/rc.d/rc.sysinit 初始化脚本，也就是 init 进程的初始化用来执行 kernel 的任务
- 第八步：执行/etc/rc.d/rc 脚本，通过执行脚本找出默认启动模式选项要启动的进程
- 第九步：执行/etc/rc.d.rc.local 个人配置脚本，也就是用户想要开机自启动的命令或者进程都放在这个脚本文件内，这样设置开机自启方式比较安全。开机时这给脚本文件内的所有命令全部执行 遍
- 第十步：启动 mingetty 进程
- 这时用户就可以看到登录界面，就可以登录系统了

守护进程的创建步骤

守护进程是后台运行的、系统启动是就存在的、不予任何终端关联的，用于处理一些系统级别任务的特殊进程。

1. 创建子进程，终止父进程

   由于守护进程是脱离控制终端的，因此　先创建子进程，终止父进程，使得程序在 shell 终端里造成　个已经运行完毕的假象。之后所有的工作都在子进程中完成，而用户在 shell 终端里则可以执行其他的命令，从而使得程序以僵尸孤儿进程形式运行，在形式上做到了与控制终端的脱离。

2. 在子进程中创建新会话

   这个步骤是创建守护进程中最重要的　步，在这里使用的是系统函数 setsid。setsid 函数在调用 fork 函数时，子进程全盘拷贝父进程的会话期（session，是　个或多个进程组的集合）、进程组、控制终端等，虽然父进程退出了，但原先的会话期、进程组、控制终端等并没有改变，因此，那还不是真正意义上使两者独立开来。setsid 函数能够使进程完全独立出来，从而脱离所有其他进程的控制。

3. 改变工作目录

   使用 fork 创建的子进程也继承了父进程的当前工作目录。由于在进程运行过程中，当前目录所在的文件系统不能卸载，因此，把当前工作目录换成其他的路径，如“/”或“/tmp”等。改变工作目录的常见函数是 chdir。

4. 重设文件创建掩码

   文件创建掩码是指屏蔽掉文件创建时的对应位。由于使用 fork 函数新建的子进程继承了父进程的文件创建掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件创建掩码设置为 0，可以大大增强该守护进程的灵活性。设置文件创建掩码的函数是 umask，通常的使用方法为 umask(0)。

5. 关闭文件描述符

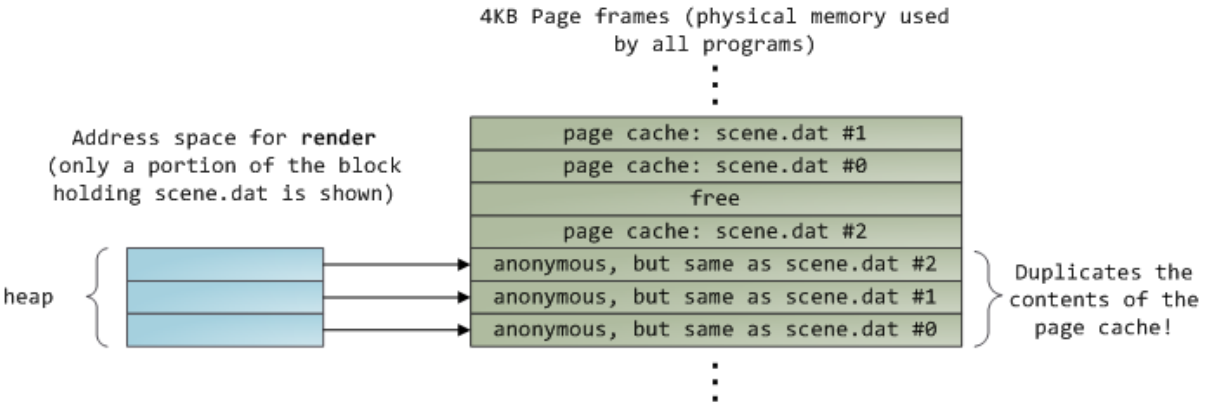
   用 fork 新建的子进程会从父进程那里继承　些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们　样消耗系统资源，可能导致所在的文件系统无法卸载。

从创建进程到程序运行，操作系统做了什么事情

从输入运行后，创建子进程，分配内存空间，设置 CPU 环境，等待调度，页面置换等过程。

文件读取


页面缓存、内存和文件之间的那些事



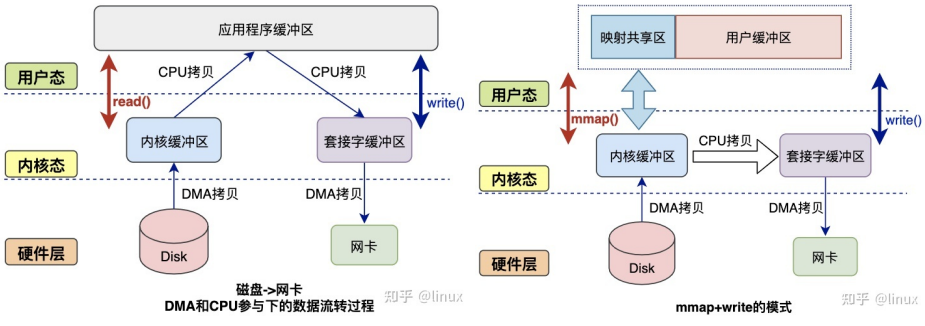
mmap

mmap 是　种内存映射文件的方法，即将　个文件映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中　段虚拟地址的　　对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这　段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用 read，write 等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。

mmap 和普通文件的区别

常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。写操作也是　样，待写入的 buffer 在内核空间不能直接访问，必须要先拷贝　内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

使用 mmap 操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用　次数据拷贝(磁盘一内存)，就可以完成文件的读取。



mmap 内存映射原理

1. 进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域
2. 调用内核空间的系统调用函数 mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的　　映射关系
3. 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝

mmap 和零拷贝

如果要读取　个文件并通过网络发送它，传统的 I/O 操作进行了 4 次用户空间与内核空间的上下文切换（用户态调用 read 进入内核态，内核态从 read 返回到用户态，write 同理），以及 4 次数据拷贝（DMA 从磁盘拷贝→内核→用户缓冲空间，用户缓冲空间→socket 内核→DMA 拷贝网络中）。其中 4 次数据拷贝中包括了 2 次 DMA 拷贝和 2 次 CPU 拷贝。通过零拷贝技术完成相同的操作，减少了在用户空间与内核空间交互，并且不需要 CPU 复制数据。

应用程序调用 mmap()，磁盘上的数据会通过 DMA 拷贝到内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，应用程序再调用 write()，操作系统直接将内核缓冲区的内容拷贝到 socket 缓冲区中，这　切都发生在内核态，最后，socket 缓冲区再把数据发到网卡去。

sendfile

sendfile 实现的零拷贝 I/O 只使用了 2 次用户空间与内核空间的上下文切换（发文件和收文件），以及 3 次数据的拷贝。其中 3 次数据拷贝中包括了 2 次 DMA 拷贝和 1 次 CPU 拷贝。

高级I/O函数之sendfile函数

1. 系统调用 sendfile() 通过 DMA 把硬盘数据拷贝到 kernel buffer，然后数据被 kernel 直接拷贝到另外一个与 socket 相关的 kernel buffer。这里没有 用户态和核心态 之间的切换，在内核中直接完成了从一个 buffer 到另一个 buffer 的拷贝。
2. DMA 把数据从 kernel buffer 直接拷贝给协议栈，没有切换，也不需要数据从用户态和核心态，因为数据就在 kernel 里。

感觉就像把 mmap 及其后续 write 过程封装了一下

IO 行为

NIO进阶篇：Page Cache、零拷贝、顺序读写、堆外内存

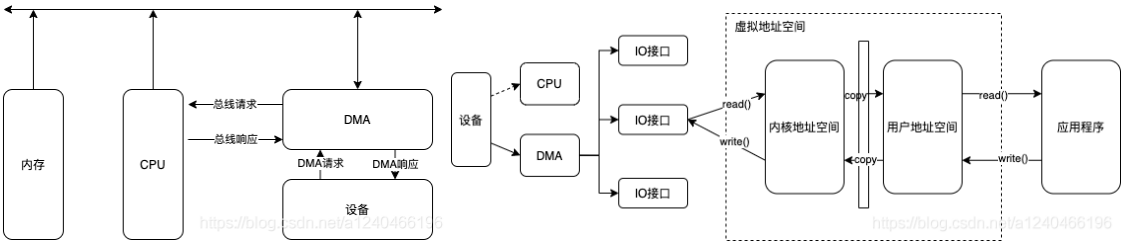
DMA

DMA 的出现就是为了解决批量数据的输入/输出问题。DMA 是指外部设备不通过 CPU 而直接

与系统内存交换数据的接口技术。类比显卡，也是从 CPU 中剥离出来的功能。将这些特殊的模块进行剥离，使得 CPU 可以更加专注于计算工作。

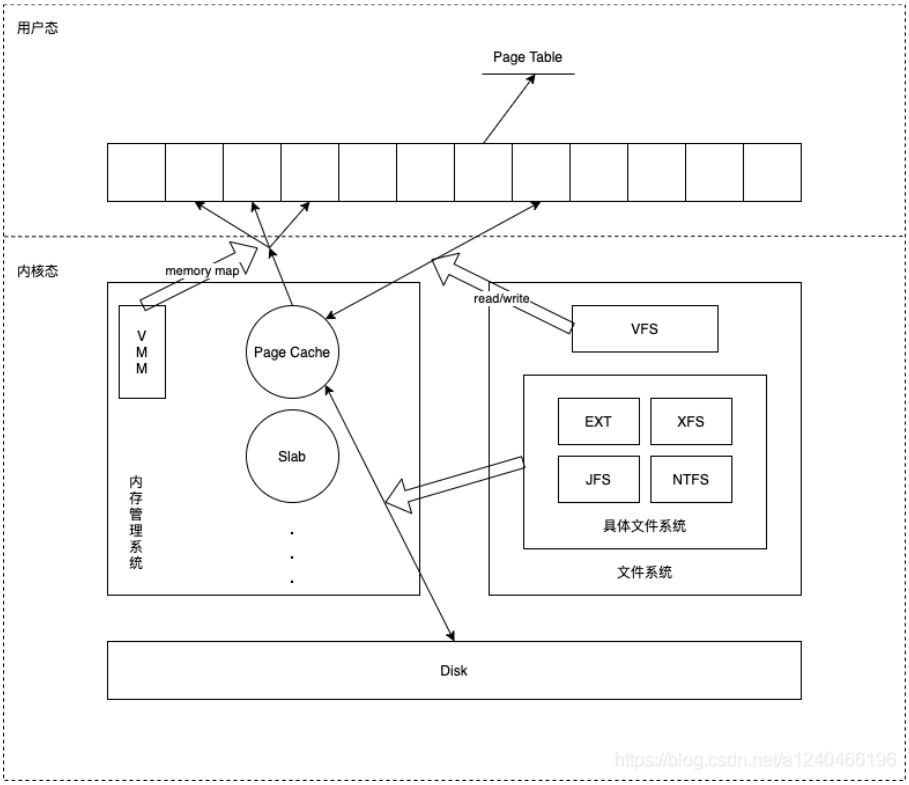
通常系统总线是由 CPU 管理的，在 DMA 方式时，就希望 CPU 把这些总线让出来而由 DMA 控制器接管，控制传送的字节数，判断 DMA 是否结束，以及发出 DMA 结束信号。因此 DMA 控制器必须有以下功能:

1. 能向 CPU 发出系统保持(HOLD)信号，提出总线接管请求；
2. 当 CPU 发出允许接管信号后，对总线的控制由 DMA 接管;
3. 能对存储器寻址及能修改地址指针，实现对内存的读写；
4. 能决定本次 DMA 传送的字节数，判断 DMA 传送是否借宿。
5. 发出 DMA 结束信号，使 CPU 恢复正常工作状态。



Page Cache

对于存储设备上的数据，操作系统向应用程序提供的逻辑概念就是"文件"。应用程序要存储或访问数据时，只需读或者写"文件"的一维地址空间即可，而这个地址空间与存储设备上存储块之间的对应关系则由操作系统维护。说白了，文件就是基于内核态 Page Cache 的一层抽象。

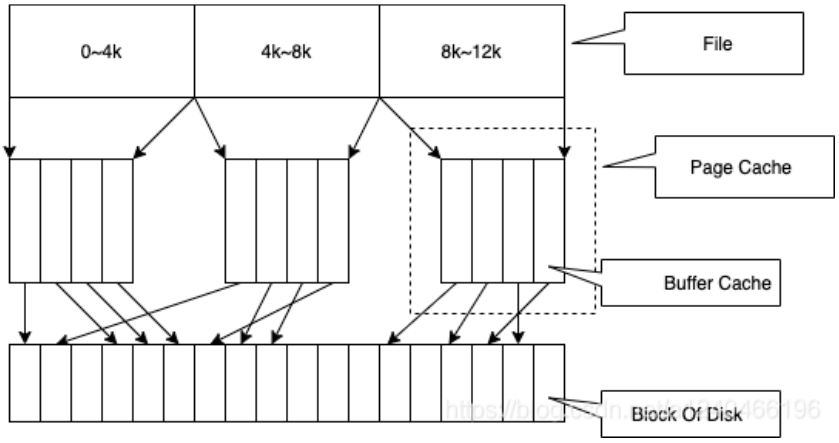


图中描述了 Linux 操作系统中文件 Cache 管理与内存管理以及文件系统的关系示意图。从图中可以看到，在 Linux 中，具体文件系统，如 ext2/ext3、jfs、ntfs 等，负责在文件 Cache 和存储设备之间交换数据，位于具体文件系统之上的虚拟文件系统 VFS 负责在应用程序和文件 Cache 之间通过 read/write 等接口交换数据，而内存管理系统负责文件 Cache 的分配和回收，同时虚拟内存管理系统(VMM)则允许应用程序和文件 Cache 之间通过 memory map 的方式交换数据。可见，在 Linux 系统中，文件 Cache 是内存管理系统、文件系统以及应用程序之间的一个联系枢纽。

Page cache 和 Buffer cache

每一个 Page Cache 包含若干 Buffer Cache。

1. 内存管理系统与 Page Cache 交互，负责维护每项 Page Cache 的分配和回收，同时在使用 memory map 方式访问时负责建立映射；
2. VFS 与 Page Cache 交互，负责 Page Cache 与用户空间的数据交换，即文件读写；
3. 具体文件系统则一般只与 Buffer Cache 交互，它们负责在外围存储设备和 Buffer Cache 之间交换数据。



普通拷贝与零拷贝

【linux】图文并茂|彻底搞懂零拷贝（Zero-Copy）技术

Linux 内核中与 Page Cache 操作相关的 API 有很多，按其使用方式可以分成两类：一类是以拷贝方式操作的相关接口， 如 read/write/sendfile 等；另一类是以地址映射方式操作的相关接口，如 mmap。其中 sendfile 和 mmap 都是零拷贝的实现方案。

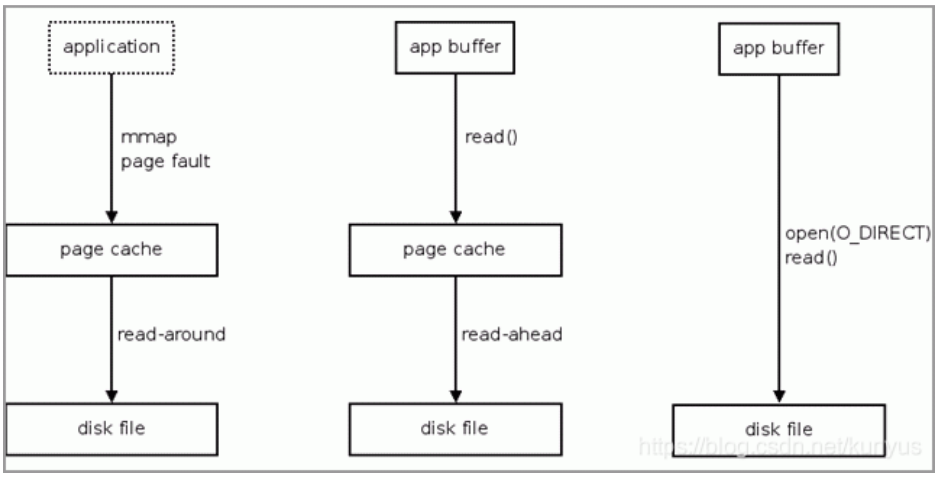
文件预读

Linux内核的文件预读机制详细详解

- read-around 算法适用于那些以 mmap 方式访问的程序代码和数据，它们具有很强的局域性(locality of reference)特征。当有缺页事件发生时，它以当前页面为中心，往前往后预取

共计 128KB 页面。

- read-ahead 算法主要针对 read()系统调用，它们一般都具有很好的顺序特性。但是随机和非典型的读取模式也大量存在，因而 read-ahead 算法必须具有很好的智能和适应



read-ahead

Linux 内核中文件预读算法的具体过程是这样的：

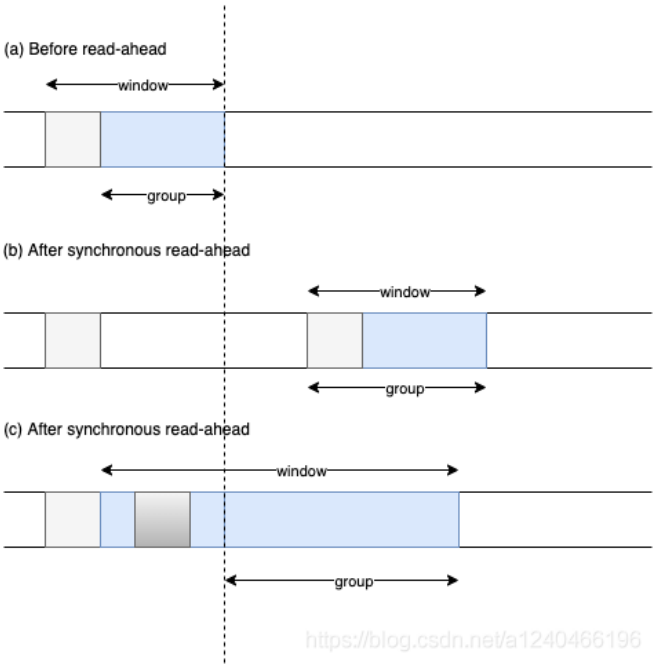
- 对于每个文件的第一个读请求，系统读入所请求的页面并读入紧随其后的少数几个页面(不少于一个页面，通常是三个页面)，这时的预读称为同步预读。
- 对于第二次读请求，
  - 如果所读页面不在 Cache 中，即不在前次预读的 group 中，则表明文件访问不是顺序访问，系统继续采用同步预读
  - 如果所读页面在 Cache 中，则表明前次预读命中，操作系统把预读 group 扩大一倍，并让底层文件系统读入 group 中剩下尚不在 Cache 中的文件数据块，这时的预读称为异步预读。

无论第二次读请求是否命中，系统都要更新当前预读 group 的大小。

此外，系统中定义了一个 window，它包括前一次预读的 group 和本次预读的 group。任何接下来的读请求都会处于两种情况之一：

- 第一种情况是所请求的页面处于预读 window 中，这时继续进行异步预读并更新相应的 window 和 group；
- 第二种情况是所请求的页面处于预读 window 之外，这时系统就要进行同步预读并重置相应的 window 和 group。

如下是 Linux 内核预读机制的一个示意图，其中 a 是某次读操作之前的情况，b 是读操作所请求页面不在 window 中的情况，而 c 是读操作所请求页面在 window 中的情况。



图中 group 指一次读入 page cached 的集合；window 包括前一次预读的 group 和本次预读的 group；浅灰色代表要用户想要查找的 page cache，深灰色代表命中的 page。

MMIO 与 PMIO

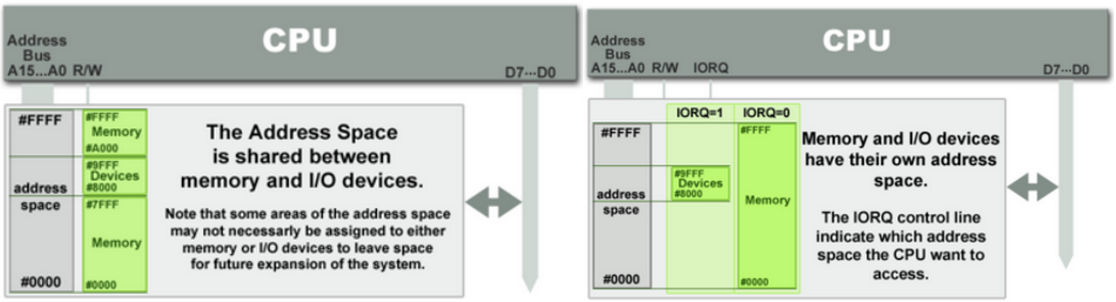
浅谈内存映射I/O(MMIO)与端口映射I/O(PMIO)的区别

概念

在计算机中，内存映射 I/O（MMIO）和端口映射 I/O(PMIO)是两种互为补充的 I/O 方法，在 CPU 和外部设备之间。另一种方法是使用专用的 I/O 处理器，通常为大型机上的通道，它们执行自己特有的指令。

在 MMIO 中，内存和 I/O 设备共享同一个地址空间。 MMIO 是应用得最为广泛的一种 IO 方法，它使用相同的地址总线来处理内存和 I/O 设备，I/O 设备的内存和寄存器被映射到与之相关联的地址。当 CPU 访问某个内存地址时，它可能是物理内存，也可以是某个 I/O 设备的内存。因此，用于访问内存的 CPU 指令也可来访问 I/O 设备。每个 I/O 设备监视 CPU 的地址总线，一旦 CPU 访问分配给它的地址，它就做出响应，将数据总线连接到需要访问的设备硬件寄存器。为了容纳 I/O 设备，CPU 必须预留给 I/O 一个地址区域，该地址区域不能给物理内存使用。（知道了这个之后就能理解为什么 cxl 的缓存一致性是 pcie 的一个升级版本了）

在 PMIO 中，内存和 I/O 设备有各自的地址空间。 端口映射 I/O 通常使用一种特殊的 CPU 指令，专门执行 I/O 操作。在 Intel 的微处理器中，使用的指令是 IN 和 OUT。这些指令可以读/写 1,2,4 个字节(例如：outb, outw, outl)从/到 IO 设备上。I/O 设备有一个与内存不同的地址空间，为了实现地址空间的隔离，要么在 CPU 物理接口上增加一个 I/O 引脚，要么增加一条专用的 I/O 总线。由于 I/O 地址空间与内存地址空间是隔离的，所以有时将 PMIO 称为被隔离的 IO(Isolated I/O)。



对比

- 在 MMIO 中，IO 设备和内存共享同一个地址总线，因此它们的地址空间是相同的; 而在 PMIO 中，IO 设备和内存的地址空间是隔离的。
- 在 MMIO 中，无论是访问内存还是访问 IO 设备，都使用相同的指令； 而在 PMIO 中，CPU 使用特殊的指令访问 IO 设备，在 Intel 微处理器中，使用的指令是 IN 和 OUT。

注意： 内存映射(MMIO 和 PMIO)作为一种 CPU 对 I/O 设备(CPU-to-device)的通信方法，并不影响 DMA(直接内存访问), 因为 DMA 是一种绕过 CPU 的内存对设备(memory-to-device)的通信方法。

内存映射IO (MMIO) 简介

1. 前者不占用 CPU 的物理地址空间，后者占有（这是对 x86 架构说的，一些架构，如 IA64，port I/O 占用物理地址空间）。
2. 前者是顺序访问。也就是说在一条 I/O 指令完成前，下一条指令不会执行。例如通过 Port I/O 对设备发起了操作，造成了设备寄存器状态变化，这个变化在下一条指令执行前生效。uncache 的 MMIO 通过 uncahce memory 的特性保证顺序性。
3. 使用方式不同。由于 port I/O 有独立的 64K I/O 地址空间，但 CPU 的地址线只有一套，所以必须区分地址属于物理地址空间还是 I/O 地址空间。

如何实现 MMIO

在 Linux 中， 内核使用 ioremap()将 IO 设备的物理内存地址映射到内核空间的虚拟地址上；用户空间程序使用 mmap(2)系统调用将 IO 设备的物理内存地址映射到用户空间的虚拟内存地址上，一旦映射完成，用户空间的一段内存就与 IO 设备的内存关联起来，当用户访问用户空间的这段内存地址范围时，实际上会转化为对 IO 设备的访问。

mmap 和 MMIO

（个人理解）感觉上 mmap 好像和 memory map IO 很接近，其实是完全不同的概念。mmap 是一个系统调用，目的是把一段内存映射到用户虚拟地址空间中，mmio 是一种 io 方式，后者通过前者实现。我们普通的 mmap 一个文件，相当于把文件通过 dma 读到内核的一段地址空间中，然后用户 mmap 这段地址空间访问。mmio 则不考虑文件系统，就如上一节所说，通过 ioremap 直接把设备映射到内核一段地址空间中，用户 mmap 这段地址空间访问。

内存屏障

内存屏障（Memory Barrier）究竟是个什么鬼？

缓存一致性协议导致的性能损失

如上图 CPU 0 执行了一次写操作，但是此时 CPU 0 的 local cache 中没有这个数据。于是 CPU 0 发送了一个 Invalidate 消息，其他所有的 CPU 在收到这个 Invalidate 消息之后，需要将自己 CPU local cache 中的该数据从 cache 中清除，并且发送消息 acknowledge 告知 CPU 0。CPU 0 在收到所有 CPU 发送的 ack 消息后会将数据写入到自己的 local cache 中。这里就产生了性能问题：当 CPU 0 在等待其他 CPU 的 ack 消息时是处于停滞的（stall）状态，大部分的时间都是在等待消息。为了提高性能就引入的 Store Buffer。

为了减少性能损失增加的 store buffer

store buffer 的目的是让 CPU 不再操作之前进行漫长的等待时间，而是将数据先写入到 store buffer 中，CPU 无需等待可以继续执行其他指令，等到 CPU 收到了 ack 消息后，再从 store buffer 中将数据写入到 local cache 中。

（也就是说通过 store buffer，某次写操作可以只完成一半，这就带来了两个问题：一是同一个数据可能同时存在在 store buffer 和 cache 中，该读哪个；二是 store buffer 中的数据不归缓存一致性协议管，会带来多核访问问题）

问题 1：单机，store forwarding

导致这个问题是因为 CPU 对内存进行操作的时候，顺序和程序代码指令顺序不一致。在写操作执行之前就先执行了读操作。另一个原因是在同一个 CPU 中同一个数据存在不一致的情况，在 store buffer 中是最新的数据，在 cache line 中是旧的数据。为了解决在同一个 CPU 的 store buffer 和 cache 之间数据不一致的问题，引入了 Store Forwarding。store forwarding 就是当 CPU 执行读操作时，会从 store buffer 和 cache 中读取数据，如果 store buffer 中有数据会使用 store buffer 中的数据，这样就解决了同一个 CPU 中数据不一致的问题。

问题 2：多核，内存序

```
a = 0 (on CPU1) , b = 0 (on CPU0);
void fun1() {   // on CPU0
  a = 1;   // local miss
  b = 1;   // local hit
}

void fun2() {  // on CPU1
  while (b == 0) continue;  // local miss
  assert(a == 1);    // local hit
}
```

产生问题的原因是 CPU 0 对 a 的写操作还没有执行完，但是 CPU 1 对 a 的读操作已经执行了。毕竟 CPU 并不知道哪些变量有相关性，这些变量是如何相关的。不过 CPU 设计者可以间接提供一些工具让软件工程师来控制这些相关性。这些工具就是 memory barrier 指令。要想程序正常运行，必须增加一些 memory barrier 的操作。

我觉得本质上的原因是 CPU 0 对 a 的操作还没有完成就执行了对 b 的操作，~~由于 store buffer 和 cache 的独立性，缓存一致性协议并不知道 a 的值处于一个 pending 的状态~~ ，同时在系统总线上信号被响应的顺序也是不确定的。在发出层面我们有 cpu0 发出关于 a 的信号，早于 b 被修改，早于响应 cpu1 对 b 的响应，但执行层面不一定是这个顺序，cpu0 最早发出的 a 信号，反而晚于 cpu1 最后发出的 b 信号被目标接收到。