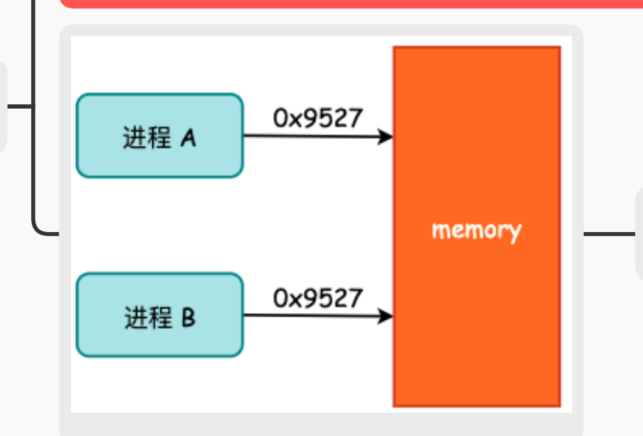
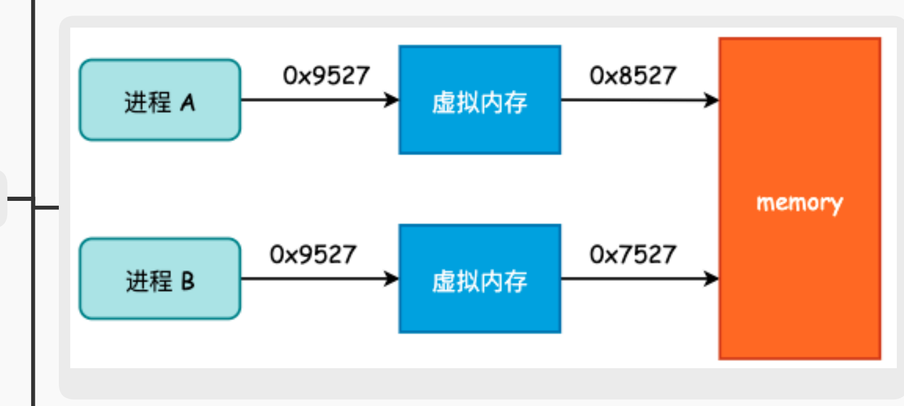
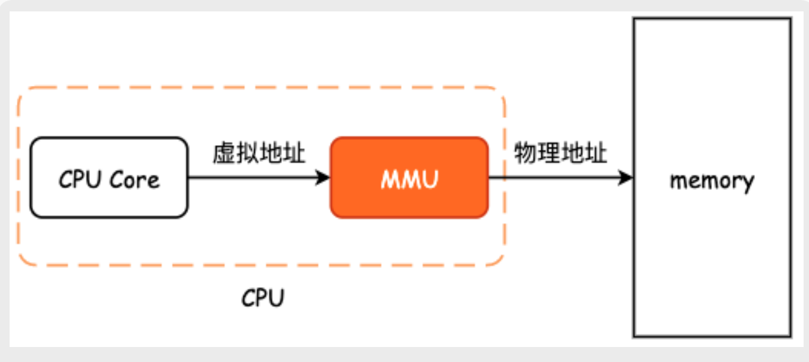
Linux 虚拟内存管理

物理地址与虚拟地址

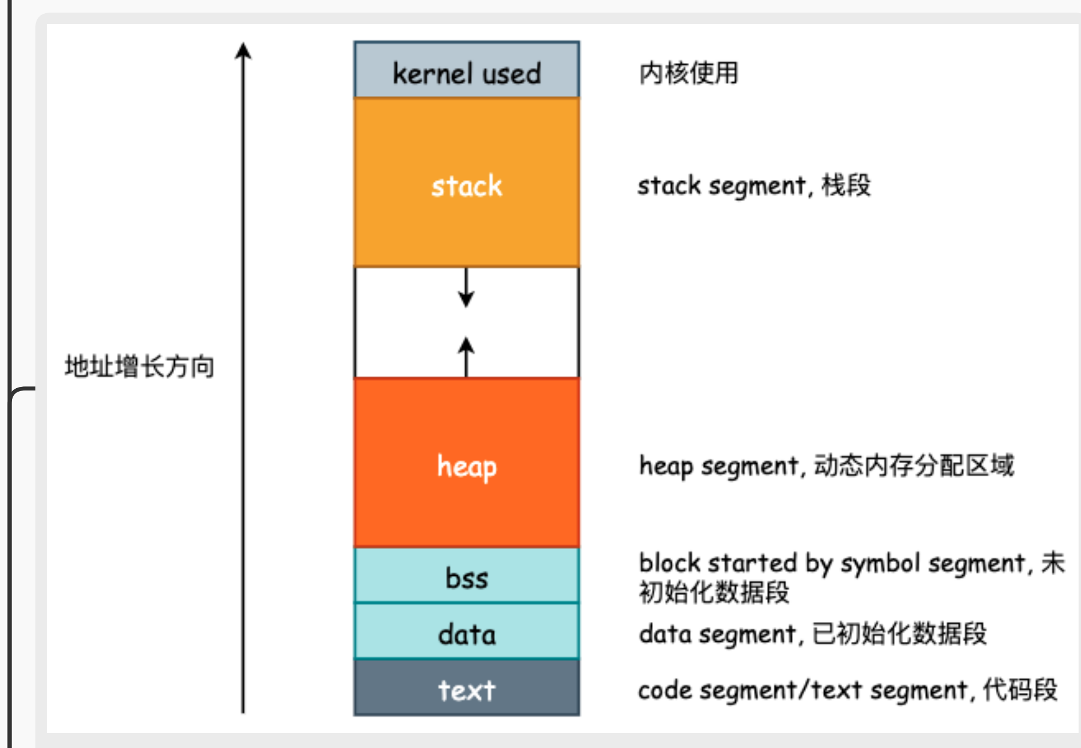
- 为什么进程不能直接使用物理地址？
  - 如果进程直接使用物理地址的话，那么同时运行多个相同的进程是不可能实现的
  - 

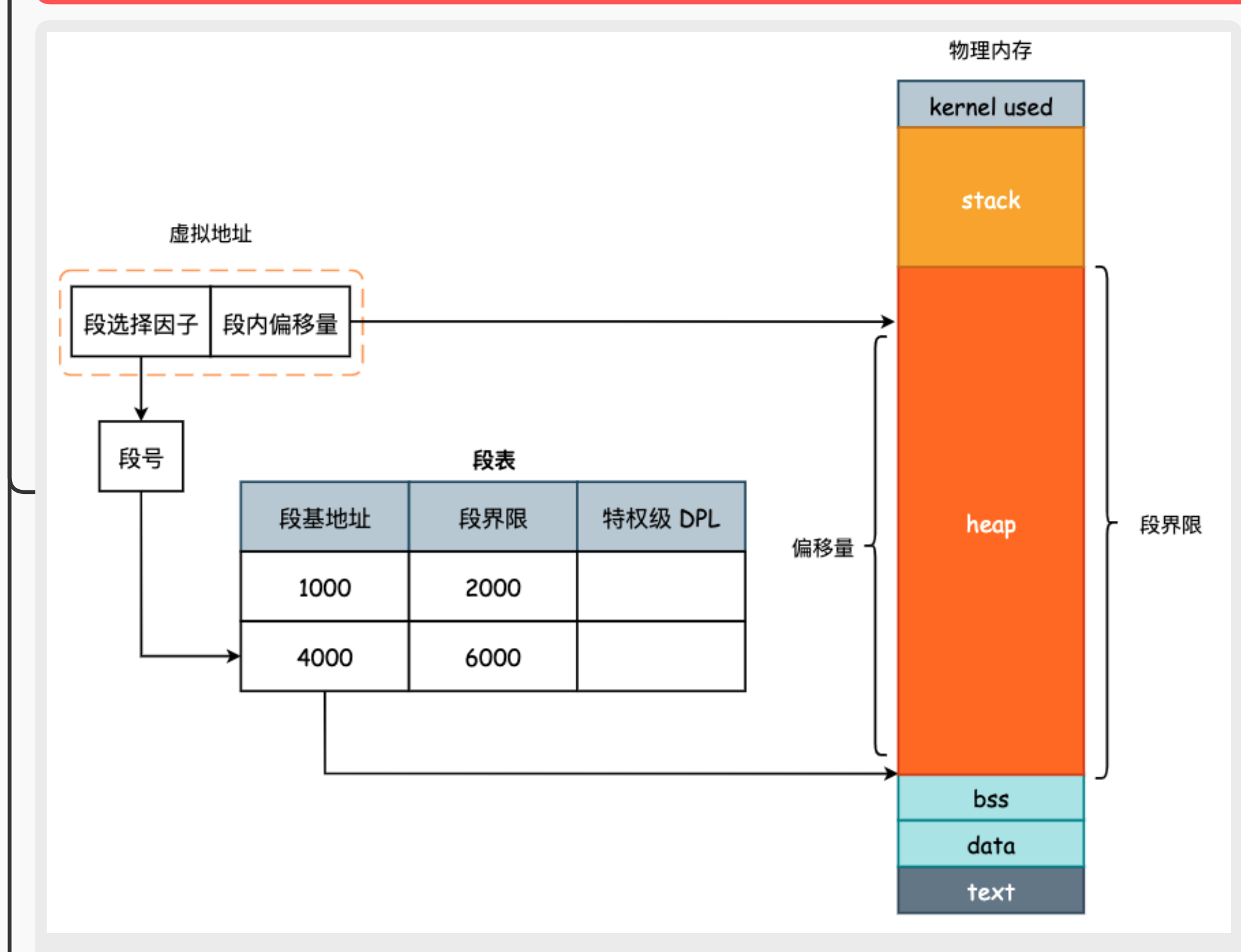
    - 若进程直接操作物理内存，那么当进程 A 运行时向 0x9527 中写入的数据就会被同样的进程 B 所覆盖掉，从而导致程序崩溃
- 虚拟内存
  - 因此，为了解决上面的问题，OS 引入了「虚拟内存」 的概念。为每一个进程都分配一个虚拟的进程地址空间，进程想在里面怎么玩就怎么玩，反正这个 「虚拟内存」机制能够保证两个进程不会写入同一块物理内存
  - 

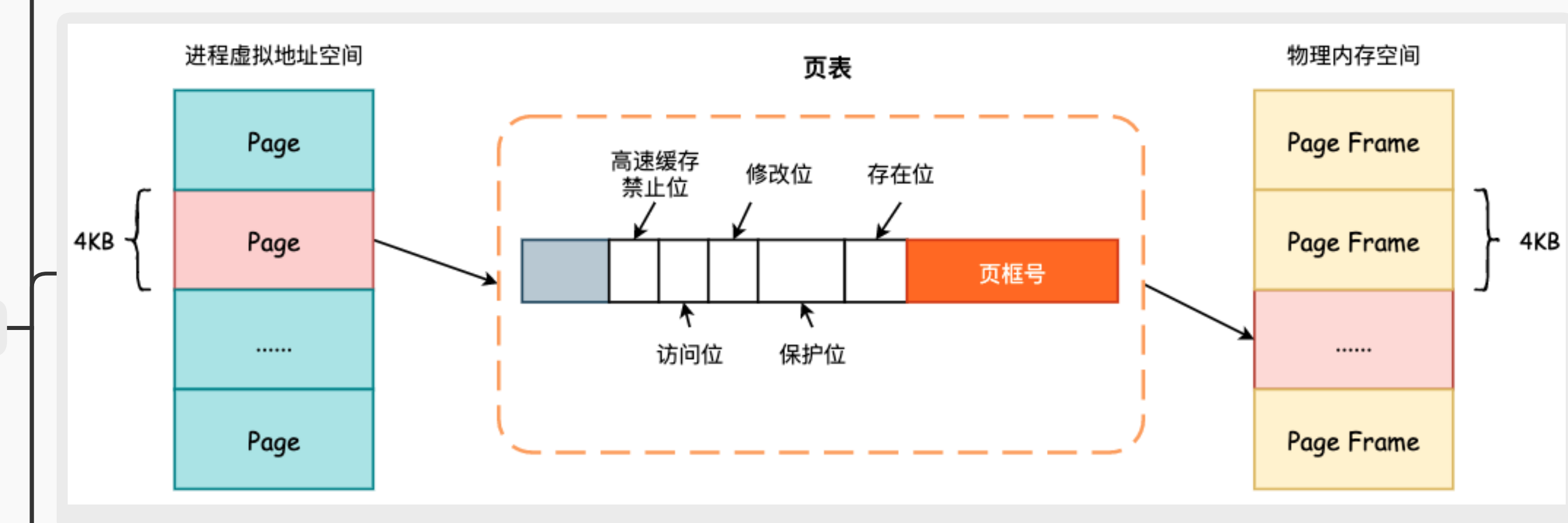
    - 也就是说，虚拟内存机制做的事情就是将进程的虚拟内存和物理内存映射起来，并且要保证在映射时不会出现不同的虚拟内存对应同一块物理内存
  - 在有了虚拟内存以后，那么同时运行多个相同的进程便可以实现，不会再出现多个相同的进程相互"打架"，相互覆盖掉同一块物理内存区域了
- MMU
  - 在有了虚拟内存机制以后，那么我们需要一个计算单元来帮助 OS 将虚拟地址转换成物理地址，从而进行内存的访问，这个计算单元就是 MMU，Memory Management Unit
  - 

    - 在这里，只需要先将 MMU 看成是一个转换器即可，也就是把虚拟内存转换成物理内存的一个黑盒
- 那么现在问题来了，操作系统该怎么去管理虚拟内存和物理内存之间的映射关系呢？

虚拟内存映射

- 分段
  - 分段机制是早期操作系统使用的虚拟内存机制，其本质就是对内存进行分类
    1. 将代码放到文本段，用于保存程序的运行指令，可以被多个进程共享，以节省内存空间
    2. 将程序中已经初始化了全局变量和静态变量放在初始化数据段，将未初始化的全局变量和静态变量放在未初始化数据段，也就是 BSS 段
    3. 将程序在运行时动态分配的内存放在堆段，函数调用所需内存则放在栈段，由多个栈帧所组成
  - 分段机制如果用比喻来说明的话，其实就是分类+整理。厨具、调料应该放在厨房，衣服、袜子应该放在衣柜，阿杜应该被放在车底。分类整理后 🏠 就会看着很舒服，并且想要找什么物品也能够很快地找到
  - 

    - 左侧为分段式进程虚拟内存空间的内存布局，需要增长的内存区域其实只有两个，也就是栈段和堆段
    - 其中栈段向下增长，也就是每当有新的函数调用，栈帧将往下放。而堆段是向上增长的
  - 当有了分段虚拟内存以后，我们就可以把虚拟内存空间中的一个又一个的段映射到物理内存上了。分段式虚拟内存地址由两部分组成，一个是段选择因子，另一个则是段内偏移量
  - 段选择因子其中包含了段号，而段号其实就是段表的一个索引，我们能够通过段号迅速的找到段表中的某一条记录。段内偏移量则是从物理内存段开始的偏移量
  - 

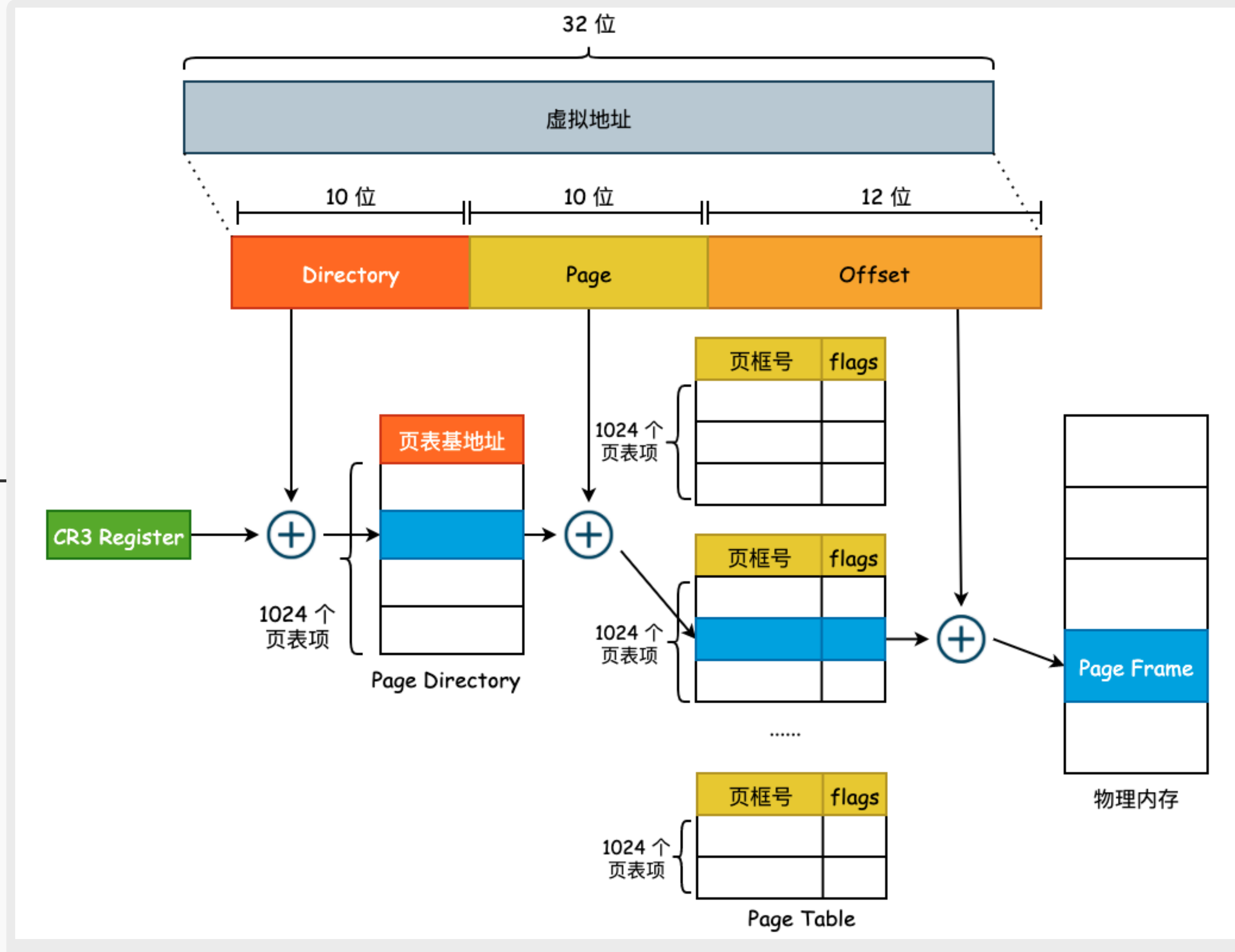
    - 也就是说，虚拟内存和物理内存之间的映射，是一段一段的，每一个段的大小并不一定相同
  - 分段虚拟内存有两个主要的缺陷
    - 非常容易产生内存碎片
    - 内存交换的效率较低
    - 产生原因就在于分段式的虚拟内存管理粒度太大
- 分页
  - 既然分段式虚拟内存管理方式是因为内存划分的粒度太大了，那么将划分粒度进一步地降低，不就可以解决内存碎片和内存交换效率低下的问题了吗？
  - 也就是分页式虚拟内存管理。分页式虚拟内存机制将虚拟进程空间以及物理内存空间切割成一块一块固定尺寸大小的页，每一页的大小都是 4KB，并且使用页表来建立虚拟内存和物理内存之间的映射，就如同段表所做的事情一样
  - 

    - Page — 进程实际使用的虚拟内存页，大小通常为 4KB
    - Page Frame — 即页框，为了将虚拟内存页和物理内存页进行区分，所以将物理内存的一页称之为页框
  - 分页内存管理有 2 个比较重要的概念: Page 和 Page Frame。前者表示进程虚拟地址空间使用的页，而 Page Frame 通常翻译成页框，用于表示物理内存中的内存单元。Page 和 Page Frame 的大小通常是相等的，在 Linux 操作系统下，其大小为 4KB
  - 由于每一页只有 4 KB，那么物理页在内存和硬盘上的换入、换出效率将得到提升：只需要换出少部分的数据页即可，而不需要一大段一大段内存的进行换入、换出
  - 😕 在页表中并没有记录下虚拟内存的 Page 号，而是只有一个页框号，那么虚拟内存地址怎么找到这一个页表项？
    - 关键点就在于页表中的页表项是连续的，那么我们就可以使用页号来作为页表的索引
    - 这其实也是为什么在一级页表的设计下，页表中必须保存所有虚拟内存页的原因

缺页中断处理

- 正是因为有了虚拟内存，OS 实际上可以运行总量超过物理内存的进程，这是因为 OS 将不常用的页面置换到了硬盘中，分段处理方式也是如此。也就是说，虚拟地中空间通常都会大于物理内存
- 那么当进程访问到不存在内存映射的页面，或者是当前物理页已经被置换到硬盘时，将会引发缺页异常
- 处理流程
  1. 陷入内核，保存程序计数器、堆栈指针、寄存器内数据等程序运行所需的重要信息
  2. 确定所需的虚拟内存地址，并校验其是否有效。若地址有效并且没有保护错误发生，则检查是否有空闲页框（Page Frame），若没有空闲页框，则调用页面置换算法淘汰一些页框
  3. 若此时页框为"脏页"，即页表项的修改位被置位时，那么需要将该页面刷新至硬盘，此时将发生上下文切换，缺页中断处理程序将被挂起，直到I/O 进程执行结束
  4. 载入进程所需的页框，因为此时仍然是一个 I/O 操作，所以也可能会出现上下文的切换，使得其它可运行的进程先运行。当页面载入完毕后，更新页表，恢复产生缺页中断之前的CPU现场
  - 也就是说，缺页中断程序在处理时，可能会有多次硬盘 I/O
- 另外值得一提的是，在执行 I/O 操作时，大部分的工作都交由 DMA 完成，CPU 可能仅仅只负责进行拷贝

优化分页机制

- 尽管分页机制带来了内存管理上性能的提升，但是同样会导致出现巨大的页表
  - 假设当前有 32G 内存，那么将会有有 32 * 1024 * 1024 / 4 = 8388608 个页，如果每一页都需要建立映射，那么最少占用内存空间为 8388608 * 8 / 1024 / 1024 / 1024 = 62.5M（假设页表项的大小为 64 位，也就是 8 字节）
  - 单个进程的 32G 进程空间需要 62.5M 的页表项存储，那么 100 个进程就需要 6.25G 的页表存储空间，这显然是一个非常严峻的问题
- 

- 那么此时，就可以使用多级页表的方式来优化页表使用空间。上图表示 32 位地址采用二级页表的方案，将虚拟地址分成 3 部分: 目录表、页表以及页内偏移量，其中页内偏移量不管使用多少层级的页表，都会存在。并且由于 Page 的大小通常为 4K，所以 Offset 基本固定为 12 位
- 二级分页一共包含 1 个目录表和 1024 个页表，不管是目录表还是页表，均包含 1024 个页表项
- 当我们采用二级分页以后，进程没有使用过的虚拟页面便可以不进行映射，或者说仅仅只映射 1 个页表，不需要再为每一个虚拟内存页都建立页表项。这是因为一级页表相当于二级页表的一个索引，当进程需要 1024~2048 的虚拟内存页时，我们只需要在第二个页表中建立页表项即可。当然，在二级分页中，目录表是必须要完全建立的，因为页表中的页表项必须连续
- 在 Linux 操作系统中，使用了四级分页策略，将虚拟地址划分成了 5 段，包括全局目录、上级目录、中间目录、页表以及页内偏移量，简而言之，就是索引的索引的索引，这里我们可以用跳跃表来理解多级分页机制