- 超越物理内存:机制
  - 如何超越物理内存
    - 到目前为止,我们一直假定地址空间非常小,能放入物理内存(页表存于内存中).
      事实上,我们假设每个正在运行的进程的地址空间都能放入内存.
    - 现在假设我们需要支持许多同时运行的巨大地址空间.
      为了达到这个目的,需要在内存层级上再加一层(---> 硬盘)
      操作系统需要把当前没有在用的那部分地址空间找个地方存储起来
    - 为什么我们要为进程支持巨大的地址空间? 答案是方便和易用性
      - 有了巨大的地址空间,不必担心程序的数据结构是否有足够空间存储,
        只需自然地编写程序,根据需要分配内存.这是操作系统提供的一个强大的假象.
      - 一个反面例子是,一些早期系统使用"内存覆盖(memory overlays)",
        它需要程序员根据需要手动移入或移出内存中的代码或数据
      - 不仅是一个进程,增加交换空间让操作系统为多个并发运行的进程都提供巨大地址空间的假象.
        多道程序的出现,强烈要求能够换出一些页,因为早期的机器显然不能将所有进程需要的所有页都放在内存中.
      - 多道程序和易用性都需要操作系统支持比物理内存更大的地址空间.
        这是所有现代虚拟内存系统都会做的事情,也是现在我们要进一步学习的内容
  - 交换空间
    - 要做的第一件事情就是在硬盘上开辟一部分空间用于物理页的移入和移出,
      这样的空间称为交换空间(swap space)
    - 假设操作系统能够以页大小为单元读取或者写入交换空间,
      因为我们将内存中的页交换到其中, 并在需要的时候又交换回去,
      为了达到这个目的,操作系统需要记住给定页的硬盘地址(disk address)
    - 例子中,有一个4页的物理内存和一个8页的交换空间.
      进程 0,1,2 共享物理内存,并且3个都只有一部分有效页在内存中,剩下的在硬盘的交换空间中.
      进程 3的所有页都被交换到硬盘上, 因此很清楚它目前没有运行.
    - 需要注意的是, 交换空间不是唯一的硬盘交换目的地.
      例如运行一个二进制程序(如ls或者你自己编译的 main程序), 这个二进制程序的代码页最开始是在硬盘上.
      但程序运行的时候, 它们被加载到内存中(要么在程序开始运行时全部加载,要么在现代操作系统中按需要一页一页加载).
      但是, 如果系统需要在物理内存中腾出空间以满足其他需求, 则可以安全地重新使用这些代码页的内存空间,
      因为稍后它又可以重新从硬盘上的二进制文件加载.
  - 存在位
    - 现在我们在硬盘上有一些空间, 需要在系统中增加一些更高级的机制, 来支持从硬盘交换页.
      简单起见, 假设有一个硬件管理 TLB 的系统
    - 内存访问时,如果TLB 未命中,则硬件在内存中查找页表,如果页有效且存在于物理内存中,则硬件从 PTE中获得 PFN,将其插入TLB,并重试该指令,这次产生 TLB 命中
    - 如果希望允许页交换到硬盘,必须添加更多的机制.
      具体来说,当硬件在 PTE中查找时,可能发现页不在物理内存中.
      硬件操作系统判断是否在内存中的方法,是通过页表项中的一条新信息,即存在位(present bit)
    - 如果存在位设置为1,则表示该页存在于物理内存中.如果存在位设置为0,则页不在内存中,而在硬盘上.
      访问不在物理内存中的页,这种行为通常被称为 page fault
    - page fault 时,操作系统被唤起来处理 page fault.
      一段称为 page-fault handler 的代码会执行来处理
  - 页错误
    page fault
    - 在 TLB 未命中的时, 两种类型的系统:硬件管理 TLB和软件管理 TLB.
      不论在哪种系统中,如果页不存在,都由操作系统负责处理页错误.
      操作系统的 page-fault handler 确定要做什么. 几乎所有的系统都在软件中处理页错误
    - 如果一个页已被交换到硬盘, 在处理 page fault 的时候, 操作系统需要将该页交换到内存中.
      问题来了: 操作系统如何知道所需的页在哪儿? 在许多系统中, 页表是存储这些信息最自然的地方.
      因此, 操作系统可以用 PTE中的某些位来存储硬盘地址, 这些位通常用来存储像页的 PFN这样的数据.
      当操作系统接收到页错误时, 它会在 PTE中查找地址, 并将请求发送到硬盘, 将页读取到内存中.
    - 为什么硬件不能处理页错误
      - 从 TLB的经验中得知, 硬件设计者不愿意信任操作系统做所有事情. 那为什么他们相信操作系统来处理页错误呢?
      - 首先, 页错误导致的硬盘操作很慢.
        即使操作系统需要很长时间来处理故障, 执行大量的指令, 但相比于硬盘操作, 这些额外开销是很小的.
      - 其次, 为了能够处理 page fault,硬件必须了解交换空间,
        如何向硬盘发起 I/O 操作, 以及很多它当前所不知道的细节.
      - 因此, 由于性能和简单的原因,操作系统来处理页错误, 即使硬件人员也很开心.
    - 当硬盘 I/O 完成时,操作系统会更新页表,将此页标记为存在, 更新页表项的 PFN字段以记录新获取页的内存位置,并重试指令.
      下一次重新访问 TLB还是未命中,然而这次因为页在内存中,因此会将页表中的地址更新到 TLB中
      (也可以在处理页错误时更新 TLB以避免此步骤).
      最后的重试操作会在 TLB中找到转换映射, 从已转换的内存物理地址,获取所需的数据或指令

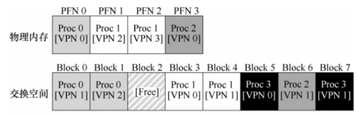

图 21.1　物理内存和交换空间

注意, 当 I/O在运行时,进程将处于阻塞(blocked)状态.
因此, 当页错误正常处理时,操作系统可以自由地运行其他可执行的进程.
因为 I/O 操作是昂贵的,一个进程进行 I/O 时会执行另一个进程,
这种交叠 (overlap) 是多道程序系统充分利用硬件的一种方式

内存满了怎么办

在上面描述的过程中, 我们假设有足够的空闲内存来存储交换空间换入(page in)的页.情况可能并非如此, 内存可能已满(或接近满了).
因此,操作系统可能希望先交换出(page out) 一个或多个页, 以便为操作系统即将交换入的新页留出空间.
选择哪些页被交换出或被替换(replace)的过程, 被称为页交换策略(page-replacement policy)

事实表明,人们在创建好页交换策略上投入了许多思考, 因为换出不合适的页会导致程序性能上的巨大损失,
也会导致程序以类似硬盘的速度运行而不是以类似内存的速度.
在现有的技术条件下, 这意味着程序可能会运行慢 10000～100000倍.
因此, 这样的策略是我们应该详细研究的. 现在我们只要知道有这样的策略存在, 建立在之前描述的机制之上.

页错误处理流程

如果有人问你：“当程序从内存中读取数据会发生什么？”
你应该对所有不同的可能性有了很好的概念.

图 21.2 展示了硬件在地址转换过程中所做的工作

当 TLB miss 发生时有 3种重要情景. 第一种情况, 该页存在(present) 且有效(valid)[18～21].
在这种情况下, TLB miss处理程序可以简单地从 PTE中获取 PFN,
然后重试指令(这次 TLB会命中)并继续之前的流程.

第二种情况[22～23 行], 页错误处理程序需要运行.
虽然这是进程可以访问的合法页(valid), 但它并不在物理内存中

第三种情况, 访问的是一个无效页, 可能由于程序中的错误[13～14行]
在这种情况下, PTE 中的其他位都不重要了.
硬件捕获这个非法访问, 操作系统陷阱处理程序运行, 可能会杀死非法进程.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else
16         if (CanAccess(PTE.ProtectBits) == False)
17             RaiseException(PROTECTION_FAULT)
18         else if (PTE.Present == True)
19             // assuming hardware-managed TLB
20             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21             RetryInstruction()
22         else if (PTE.Present == False)
23             RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

从图 21.3的软件控制流中,可以看到为了处理页错误,
操作系统大致做了什么

首先, 操作系统必须为将要换入的页找到一个物理帧,如果没有这样的物理帧,
我们将不得不等待交换算法运行, 并从内存中踢出一些页, 释放帧供这里使用.

在获得物理帧后, 处理程序发出 I/O请求从交换空间读取页

最后, 当这个慢操作完成时, 操作系统更新页表并重试指令.
重试将导致 TLB miss, 然后再一次重试时, TLB 命中.
此时硬件将能够访问所需的值

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

交换何时真正发生

到目前为止, 我们一直描述的是操作系统会等到内存已经完全满了以后才会执行交换流程,
然后才替换一个页为其他页腾出空间

这有点不切实际的, 因为操作系统可以更主动地预留一小部分空闲内存

为了保证有少量的空闲内存, 大多数操作系统会设置高水位线(High Watermark) 和低水位线（Low Watermark),
来帮助决定何时从内存中清除页

当操作系统发现有少于 LW 个页可用时, 后台负责释放内存的线程会开始运行, 直到有 HW个可用的物理页.
这个后台线程有时称为交换守护进程(swap daemon)或页守护进程 (page daemon), 它然后会很开心地进入休眠状态.

通过同时执行多个交换过程,可以进行一些性能优化. 例如许多系统会把多个要写入的页聚(cluster)或分组 (group),
同时写入到交换区间, 从而提高硬盘的效率. 这种合并操作减少了硬盘的寻道和旋转开销,从而显著提高了性能

为了配合后台的分页线程, 图 21.3中的控制流需要稍作修改.
交换算法需要先简单检查是否有空闲页, 而不是直接执行替换.
如果没有空闲页, 会通知后台分页线程按需要释放页.
当线程释放一定数目的页时,它会重新唤醒原来的线程,然后就可以把需要的页交换进内存, 继续它的工作.

小结

这个简短的一章中, 我们介绍了访问超出物理内存大小时的一些概念.
要做到这一点, 在页表结构中需要添加额外信息.

比如增加一个存在位或者其他类似机制, 告诉我们页是不是在内存中.
如果不存在, 则操作系统 page-fault handler 会运行处理 page fault, 从而将需要的页从硬盘读取到内存,
可能还需要先换出内存中的一些页, 为即将换入的页腾出空间.

很重要的是(并且令人惊讶的是), 这些行为对进程都是透明的.
对进程而言, 它只是访问自己私有的、连续的虚拟内存.
在后台, 物理页被放置在物理内存中的任意(非连续)位置,有时它们甚至不在内存中, 需要从硬盘取回

虽然我们希望在一般情况下内存访问速度很快, 但在某些情况下, 它需要多个硬盘操作的时间.
像执行单条指令这样简单的事情, 在最坏的情况下, 可能需要很多毫秒才能完成.