【ICS】虚拟内存 - Virtual Memory

- Why VM?
- What VM?
- How VM?

1. Why VM?

一个系统中的进程于其他进程共享CPU和主存资源；但是共享主存会带来问题：

- 如果太多的进程，需要太多的资源，那可能有的进程根本就跑不起来了（分到的内存不够用！）
- 如果一个进程写到另一个进程可以使用的内存，那说不定直接Failure了。

因此现代操作系统（绝大部分稍微复杂一点点的计算机）都提供了虚拟内存；它是对主存的一种抽象概念。它干了三件事情：

1. 它将memory看作是disk上一段地址空间的cache；memory中只保存活动区域，并且根据需要在disk和memory中来回传输数据——更有效地利用物理内存；
2. 它为每个进程提供了一致的地址空间——从而简化内存管理；
3. 它保护了每个进程的地址空间不被其他人破坏。

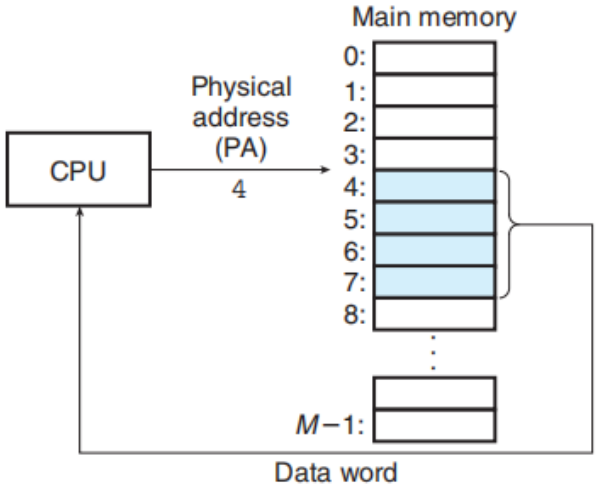
2. What VM?

2.1 物理寻址和虚拟寻址

> 地址有两种：第四章（处理器体系结构）和第六章（存储器层次结构）里面都是物理地址。我们主要讲的是cpu访存的时候，拿到物理地址就直接去访问了。
>
> 第三章（程序的机器级表示）、第五章（优化程序性能）、第七章（链接）就都是虚拟地址了（我们在汇编里面碰到的地址，全部都是虚地址）。

计算机系统的主存由连续的字节单元组成，每个字节都有唯一物理地址；CPU访问内存最简单的方式就是物理寻址（physical addressing）。

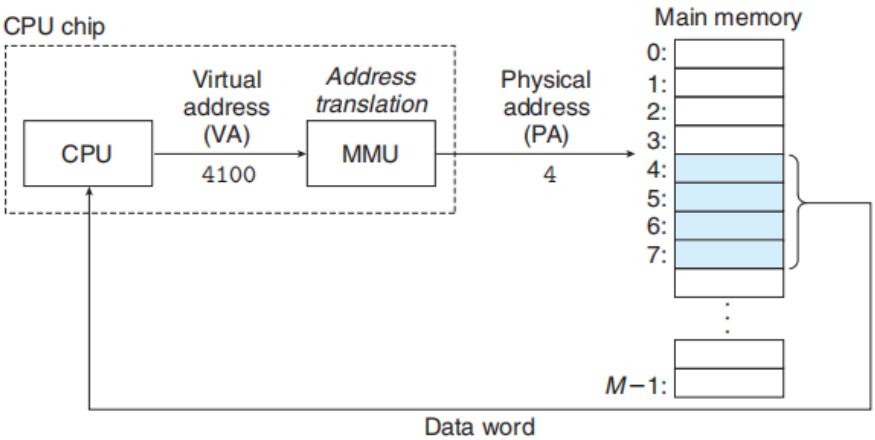
Figure 9.1
A system that uses physical addressing.



早期CPU使用物理寻址；然而现代处理器普遍采用虚拟寻址（virtual addressing）。CPU访问内存的时候，如果拿到的是虚拟地址，没有办法直接访问物理内存，就需要一个硬件——MMU，来做地址翻译的工作。

> 地址翻译不是os干的！
>
> - os：维护表：page table。
> - MMU：翻译时查表。

Figure 9.2
A system that uses virtual addressing.



2.2 地址空间

地址空间（address space）说穿了就是一个非负整数地址的有序的集合。

$$\{0, 1, 2, \ldots\}$$

便于讨论，我们假设这些整数都是连续的；换句话说，我们讨论的都是线性地址空间。

- 虚拟地址空间：

$$\{0, 1, 2, \ldots, N-1\}$$

    - 一个地址空间的大小是由表示最大地址所需要的位数来描述的。比如 `N=2**n` 个地址的虚拟地址空间就称为 `n` 位地址空间。现代系统通常支持32位或者64位虚拟地址空间。

- 物理地址空间：

$$\{0, 1, 2, \ldots, M-1\}$$

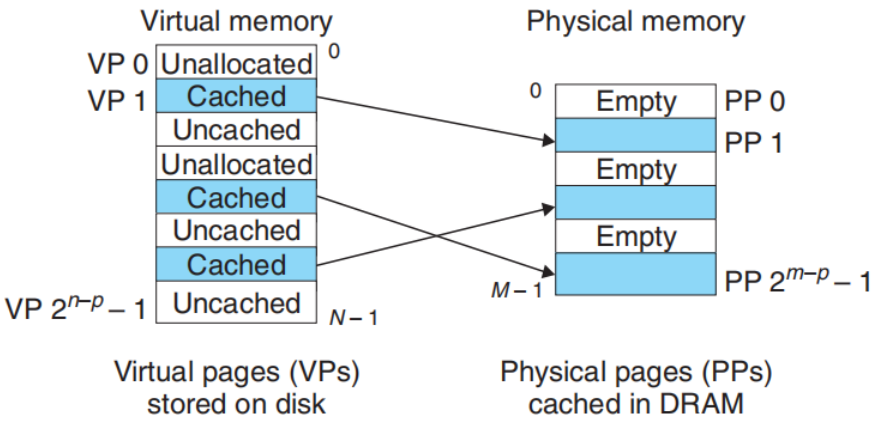
    - 对应于系统物理内存中的 `M` 个字节。 `M` 不一定是2的幂；但方便起见，我们假设它是。

> The concept of an address space is important because it makes a clean distinction between data objects (bytes) and their attributes (addresses). Once we recognize this distinction, then we can generalize and allow each data object to have multiple independent addresses, each chosen from a different address space. This is the basic idea of virtual memory. Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.
>
> —— CSAPP

3. How VM?

3.1 作为缓存的工具

Figure 9.3
How a VM system uses main memory as a cache.



虚拟内存就是disk上连续 `N` 个字节的单元组成的数组。其中的一部分被缓存到了memory上。磁盘上的数据被分割成块，作为disk和memory之间的传输单元。

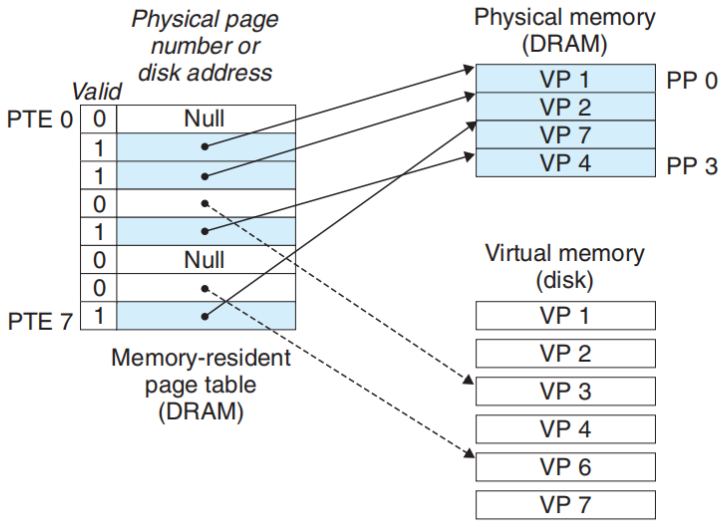
- 虚拟内存上的page：虚拟页（virtual page），大小 `P=2**p` 字节
- 物理内存上的page：物理页（physical page），大小 `P=2**p` 字节

注：

- 术语——DRAM：指虚拟内存系统的缓存，它在memory中缓存虚拟页。

3.1.1 Page Tables 页表

Figure 9.4
Page table.

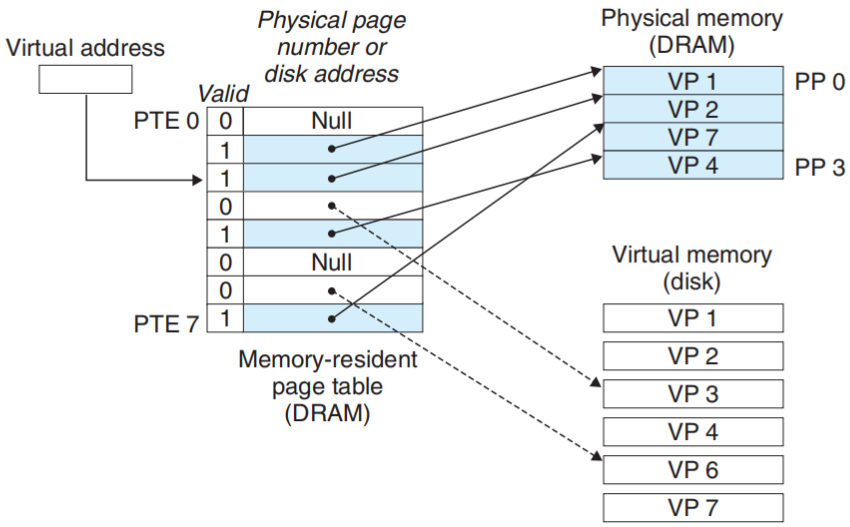


说穿了就是一种数据结构，里面有很多条目（entry），即（page table entry，PTE）。

- Valid bit：表示该page是否被缓存在DRAM中
- n位地址字段：
  - 如果valid == 1，那么指向物理内存中的page
  - 如果valid == 0，要么是null，该条目并不表示某个映射关系；要么该page不在物理内存当中，指向磁盘上的虚拟内存

3.1.2 Page Hit

Figure 9.5
VM page hit. The reference to a word in VP 2 is a hit.



3.1.3 Page Fault

Figure 9.6
VM page fault (before). The reference to a word in VP 3 is a miss and triggers a page fault.

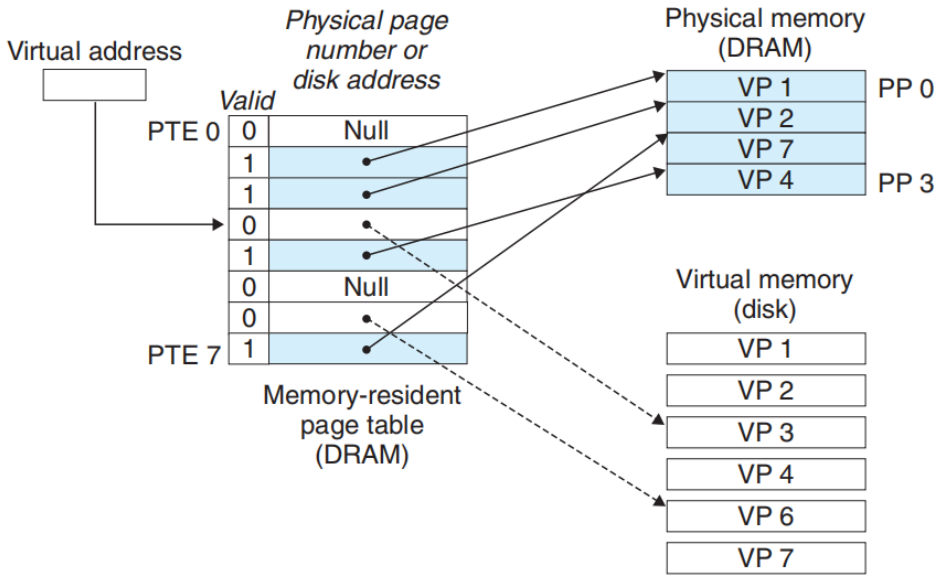
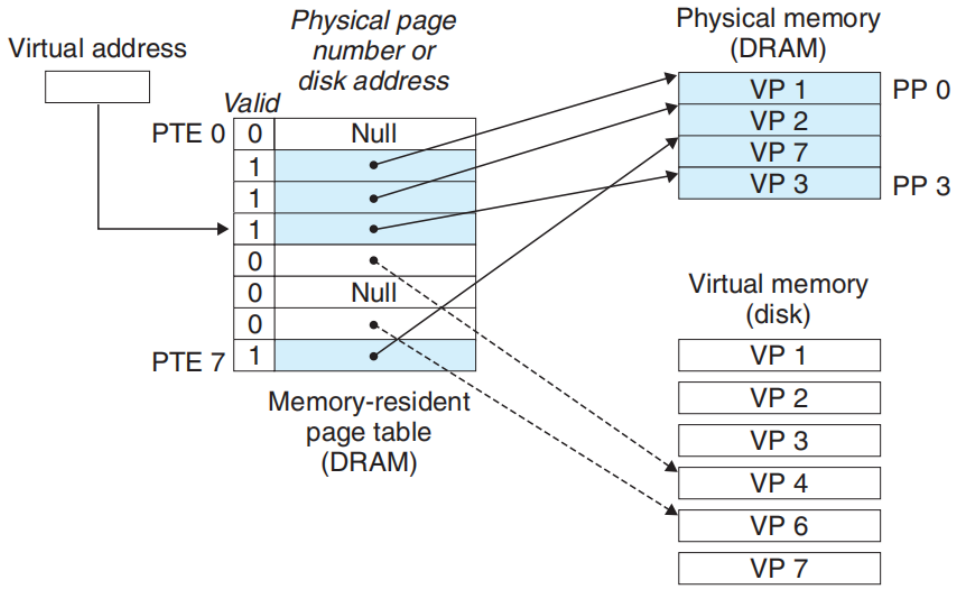


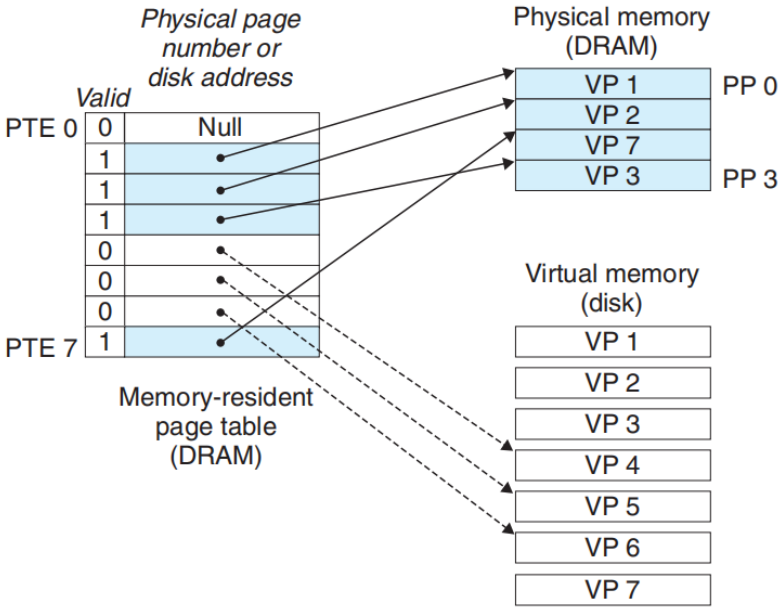
Figure 9.7
VM page fault (after). The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.



- 注意到这里的page fault是一个exception。当异常处理程序执行完毕返回以后，它会重新启动导致page fault的指令。当然现在就可以正常执行了。

3.1.4 Allocating Page

Figure 9.8
Allocating a new virtual page. The kernel allocates VP 5 on disk and points PTE 5 to this new location.



3.1.5 Locality again!

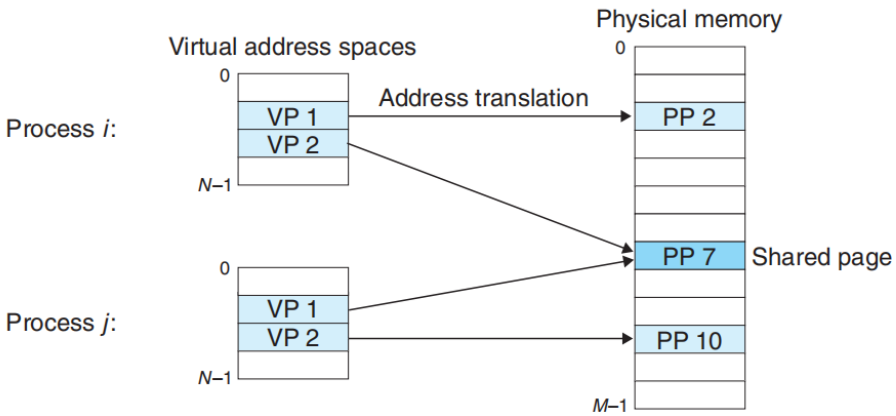
我们的第一反应可能是：这种设计听上去好XX啊！如果没有命中，会带来很大的punishment啊！

但是事实上，又是locality救了我们！

- 整个程序的运行过程中，引用的不同页面的总数可能超过物理内存的总大小，但是局部性保证了在任意时刻，程序趋向于在一个较小的active page的集合上工作！
- 也就是如果程序本身有好的time locality，虚拟内存就能工作得很好。当然如果你程序写的一泡污，运行起来慢得和爬一样，你就要考虑是不是发生了抖动——也就是页面不停地换进换出。

3.2 作为内存管理的工具

Figure 9.9
How VM provides processes with separate address spaces. The operating system maintains a separate page table for each process in the system.



到目前为止，我们都假设有一个单独的page table，来管理从虚拟地址到物理地址的映射。但是事实上，操作系统为每一个进程提供了一个独立的page table，因而也提供了一个独立的虚拟地址空间，如上图所示。注意，多个虚拟页面可以映射到同一个共享物理页面上。

> 按需页面调度和独立的虚拟地址空间的结合，对系统中内存的使用和管理造成了很深远的影响。比如，简化了**共享**。
>
> 一般而言，每个进程都有自己私有的代码、数据、堆和栈区域，不和其他进程共享。操作系统为每个进程创建页表，来做映射。
>
> 但是，事实上，有些情况进程需要共享代码和数据。比如每个进程需要调用相同的os kernal代码，每个C程序都要使用C标准库中的代码（比如printf）。如果每个进程都包括这些代码的单独的副本，就很糟糕。因此，os通过上图中共享物理页面的方式来做到这一点。

3.3 作为内存保护的工具

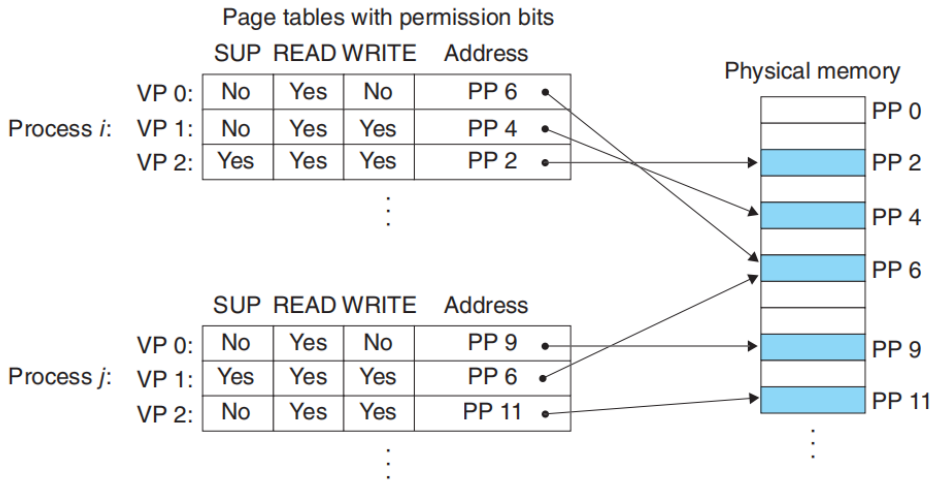
现代计算机系统需要为操作系统提供手段来控制对内存系统的访问：

- 一个user process应该被禁止修改对于它而言的read only code
- 一个user process应该被禁止读写kernal
- 一个user process应该被禁止修改和其他process共享的虚拟页，除非其他所有人都显式地允许了（通过调用进程间系统调用）

首先来说，虚拟内存为每个进程提供了独立的地址空间，也就有了私有内存。地址翻译机制可以帮我们做得更好！

因为每次读一个地址的时候，MMU都会去读一个PTE，所以我们只需要在PTE上加上一些额外的bit表示许可与否！

Figure 9.10
Using VM to provide page-level memory protection.



In this example, we have added three permission bits to each PTE. The SUP bit indicates whether processes must be running in kernel (supervisor) mode to access the page. Processes running in kernel mode can access any page, but processes running in user mode are only allowed to access pages for which SUP is 0. The READ and WRITE bits control read and write access to the page. For example, if process i is running in user mode, then it has permission to read VP 0 and to read or write VP 1. However, it is not allowed to access VP 2.

If an instruction violates these permissions, then the CPU triggers a general protection fault that transfers control to an exception handler in the kernel, which sends a SIGSEGV signal to the offending process. Linux shells typically report this exception as a "segmentation fault."

4. Address Translation

4.1 Basic Concepts

Symbol	Description
Basic parameters	
$N = 2^n$	Number of addresses in virtual address space
$M = 2^m$	Number of addresses in physical address space
$P = 2^p$	Page size (bytes)
Components of a virtual address (VA)	
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag
Components of a physical address (PA)	
PPO	Physical page offset (bytes)
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag

Figure 9.11 Summary of address translation symbols.

通常N>M，对x86来说，物理地址最48bit，虚拟地址最大52bit。

从逻辑上来说，地址翻译其实就是一个N元素的虚拟地址空间和一个M元素的物理地址空间中元素之间的映射：

Formally, address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS),

$$\text{MAP: VAS} \rightarrow \text{PAS} \cup \emptyset$$

where

$$\text{MAP}(A) = \begin{cases} A' & \text{if data at virtual addr. } A \text{ are present at physical addr. } A' \text{ in PAS} \\ \emptyset & \text{if data at virtual addr. } A \text{ are not present in physical memory} \end{cases}$$

当然因为N>M，又要求一一映射，因此映射出来可能是空的（因为在disk中）

4.2 Basic Idea

图 9-12 展示了 MMU 如何利用页表来实现这种映射。CPU 中的一个控制寄存器，页表基址寄存器（Page Table Base Register，PTBR）指向当前页表。n 位的虚拟地址包含两个部分：一个 p 位的虚拟页面偏移（Virtual Page Offset，VPO）和一个 $(n-p)$ 位的虚拟页号（Virtual

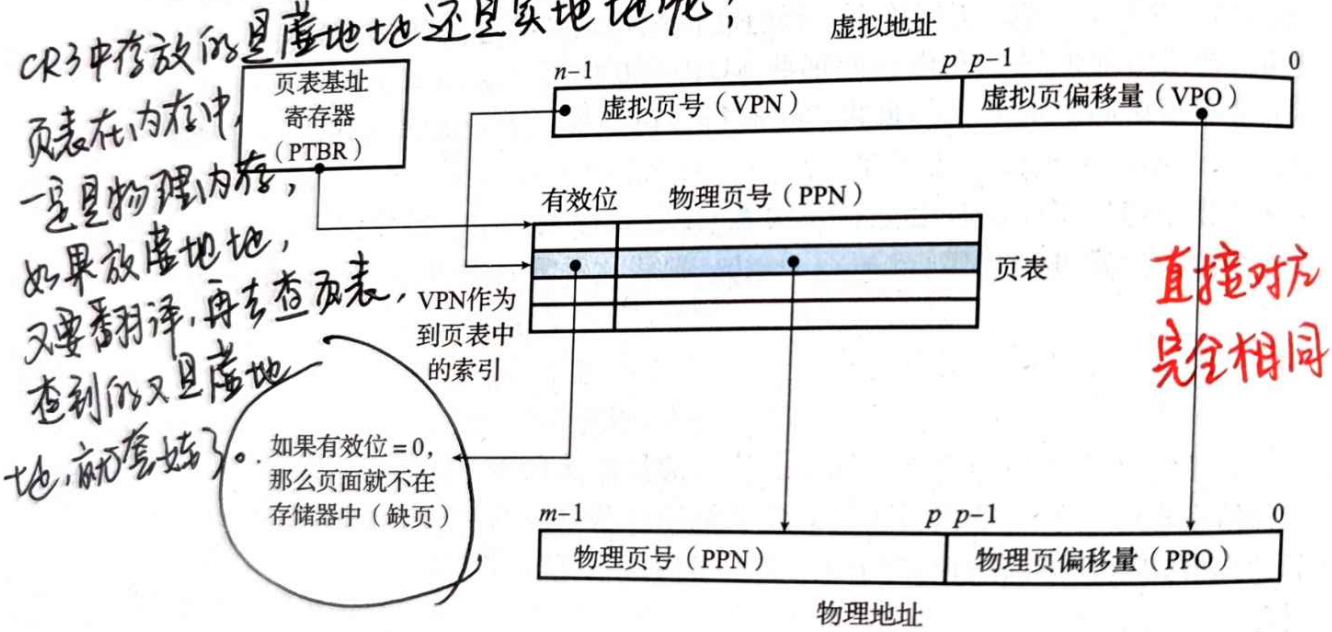


图 9-12　使用页表的地址翻译

Page Number，VPN）。MMU 利用 VPN 来选择适当的 PTE。例如，VPN 0 选择 PTE 0，VPN 1 选择 PTE 1，以此类推。将页表条目中物理页号（Physical Page Number，PPN）和虚拟地址中的 VPO 串联起来，就得到相应的物理地址。注意，因为物理和虚拟页面都是 P 字节的，所以物理页面偏移（Physical Page Offset，PPO）和 VPO 是相同的。

图 9-13a 展示了当页面命中时，CPU 硬件执行的步骤

Figure 9.13(a) shows the steps that the CPU hardware performs when there is a page hit.

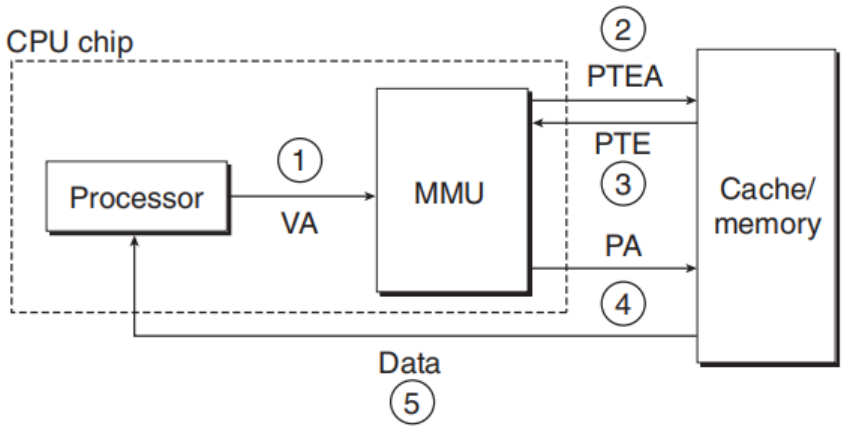
Step 1. The processor generates a virtual address and sends it to the MMU.

Step 2. The MMU generates the PTE address and requests it from the cache/main memory.

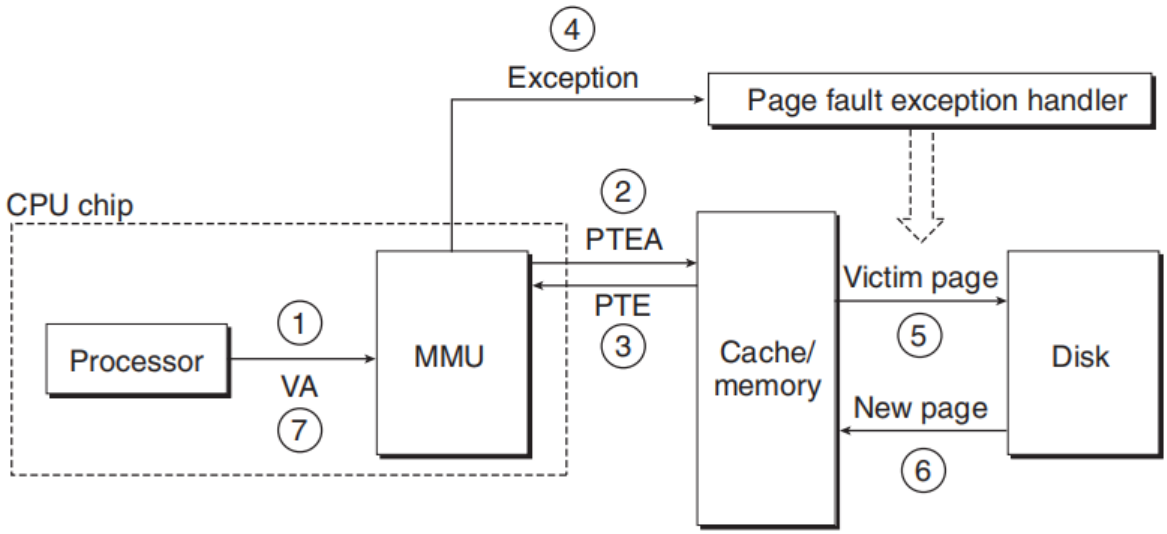
Step 3. The cache/main memory returns the PTE to the MMU.

Step 4. The MMU constructs the physical address and sends it to the cache/main memory.

Step 5. The cache/main memory returns the requested data word to the processor.



(a) Page hit



(b) Page fault

Figure 9.13 Operational view of page hits and page faults. VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

Unlike a page hit, which is handled entirely by hardware, handling a page fault requires cooperation between hardware and the operating system kernel (Figure 9.13(b)).

Steps 1 to 3. The same as steps 1 to 3 in Figure 9.13(a).

Step 4. The valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the operating system kernel.

Step 5. The fault handler identifies a victim page in physical memory, and if that page has been modified, pages it out to disk.

Step 6. The fault handler pages in the new page and updates the PTE in memory.

Step 7. The fault handler returns to the original process, causing the faulting instruction to be restarted. The CPU resends the offending virtual address to the MMU. Because the virtual page is now cached in physical memory, there is a hit, and after the MMU performs the steps in Figure 9.13(a), the main memory returns the requested word to the processor.

我们可以把整个流程再稍微扩展一下：现代计算机系统中，普遍引入了高速缓存。如果结合SRAM告诉缓存，那么整个流程应该如下：

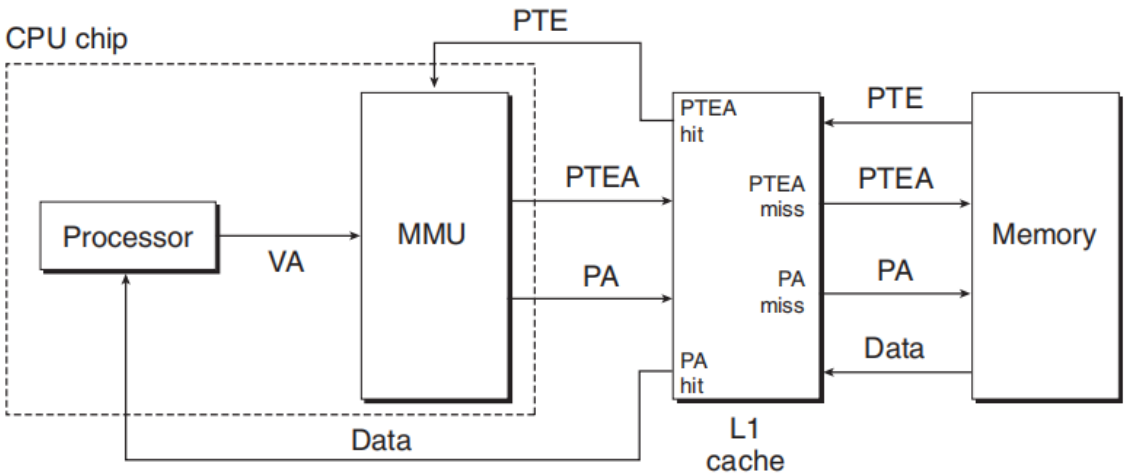


Figure 9.14 Integrating VM with a physically addressed cache. VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

> 题外话：
>
> 臧斌宇：CPU cache使用的是物理地址还是虚拟地址？
>
> Student：物理地址？

臧斌宇：再想想？

怎么说呢……应该是物理地址结合虚拟地址（或者物理地址单独也可以，但绝对不能是虚拟地址——这样大家每个人虚拟地址空间都一样，就出事情了）

CSAPP上面是这样写的：

9.6.1 Integrating Caches and VM

In any system that uses both virtual memory and SRAM caches, there is the issue of whether to use virtual or physical addresses to access the SRAM cache. Although a detailed discussion of the trade-offs is beyond our scope here, most systems opt for physical addressing. With physical addressing, it is straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from the same virtual pages. Further, the cache does not have to deal with protection issues, because access rights are checked as part of the address translation process.

Figure 9.14 shows how a physically addressed cache might be integrated with virtual memory. The main idea is that the address translation occurs before the cache lookup. Notice that page table entries can be cached, just like any other data words.

“使用物理寻址，多个进程在高速缓存中同时拥有存储块成为可能。”

4.3 TLB（翻译后备缓冲器）加速地址翻译

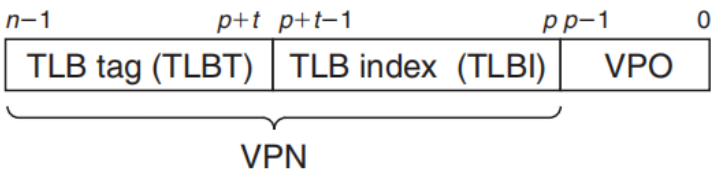
MMU拿到虚地址，先去找TLB【CPU中】。

TLB是page table之子集；

Page table：一个进程有一个，因此发生进程切换后，需要清空原先的TLB。

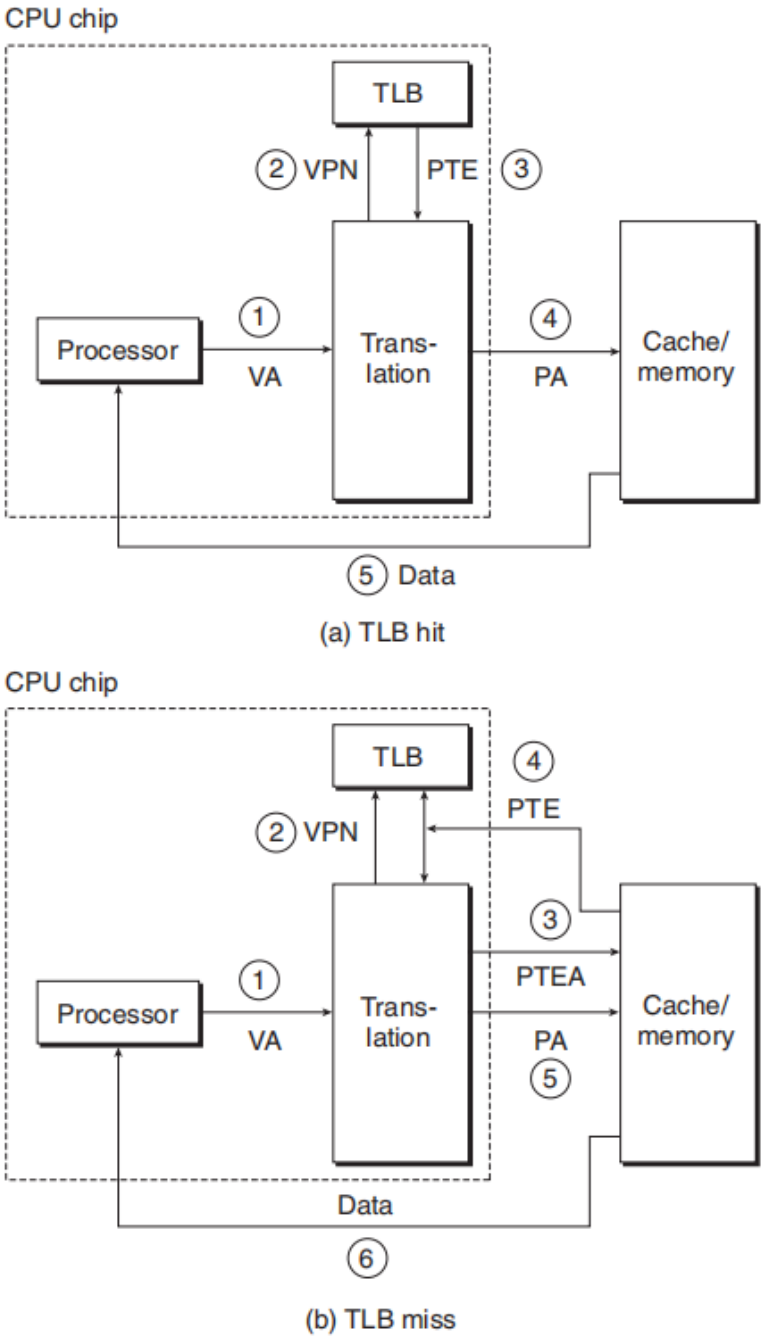
TLB长这样：

Figure 9.15
Components of a virtual address that are used to access the TLB.



整个查询的流程如下：

Figure 9.16
Operational view of a TLB hit and miss.



4.4 多级页表

如何压缩页表？分级啦。

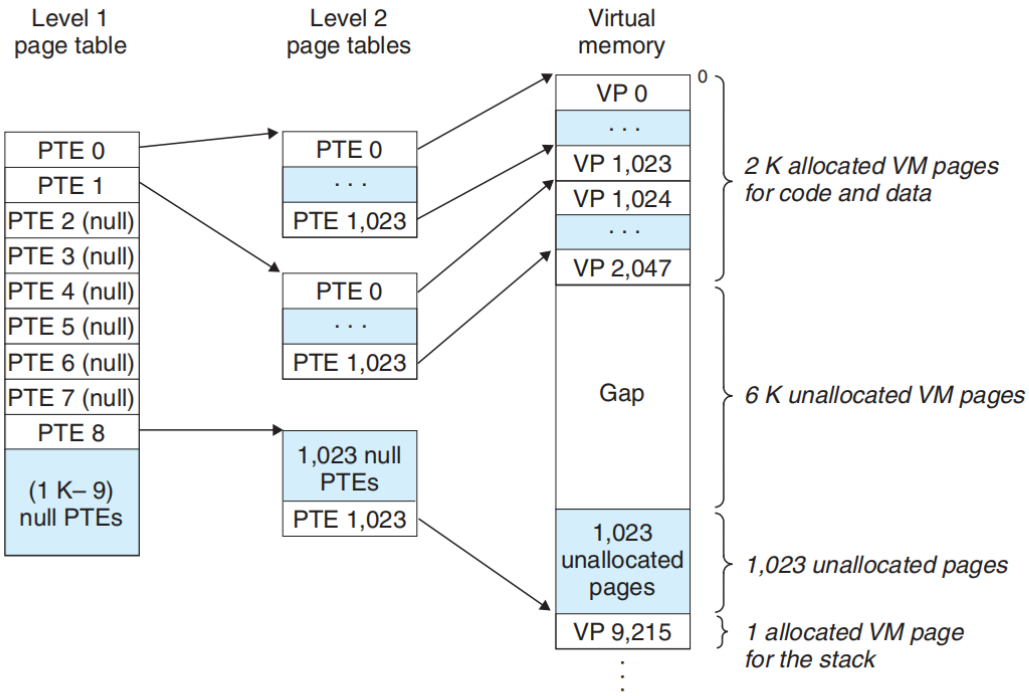


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

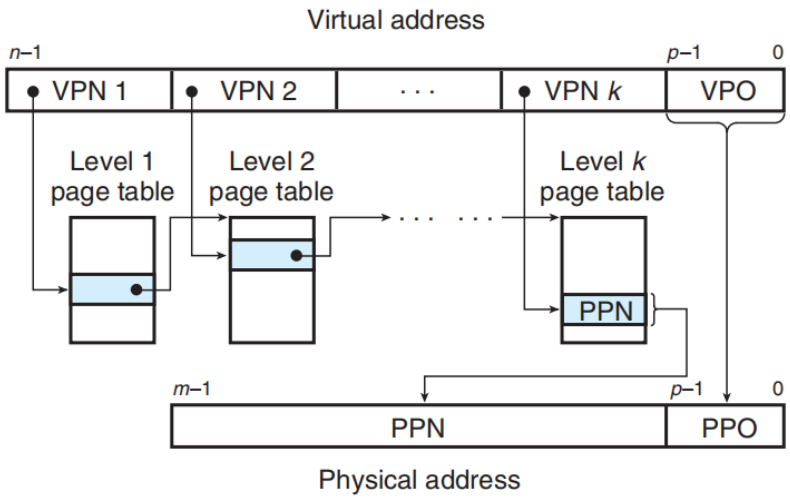


Figure 9.18 **Address translation with a k-level page table.**

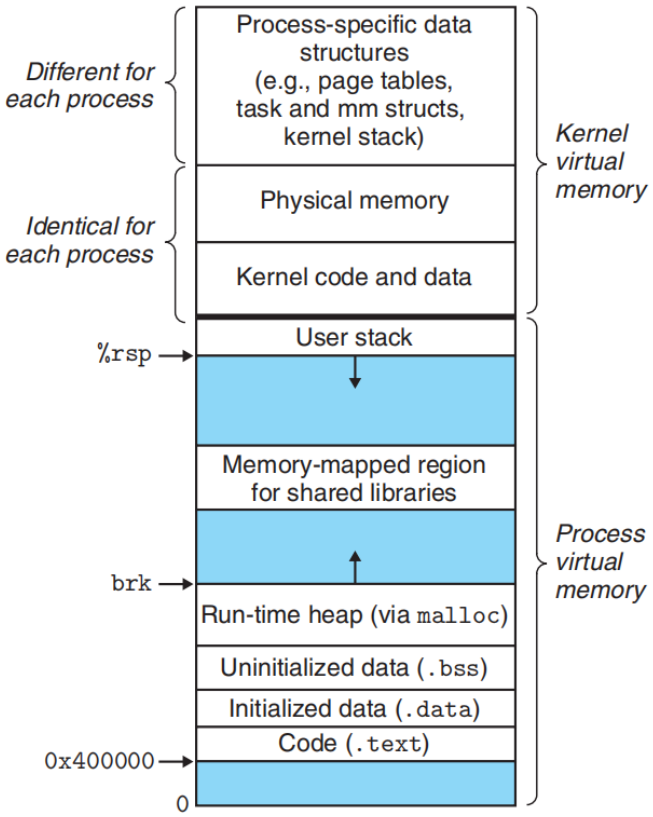
5. Case Study

5.1 Intel Core i7

TODO

5.2 Linux Memory System

Figure 9.26
The virtual memory of a Linux process.



Linux为每个进程维护了一个单独的虚拟地址空间——这幅图我们已经看见过好多遍了。这部分虚拟内存位于用户栈之上。

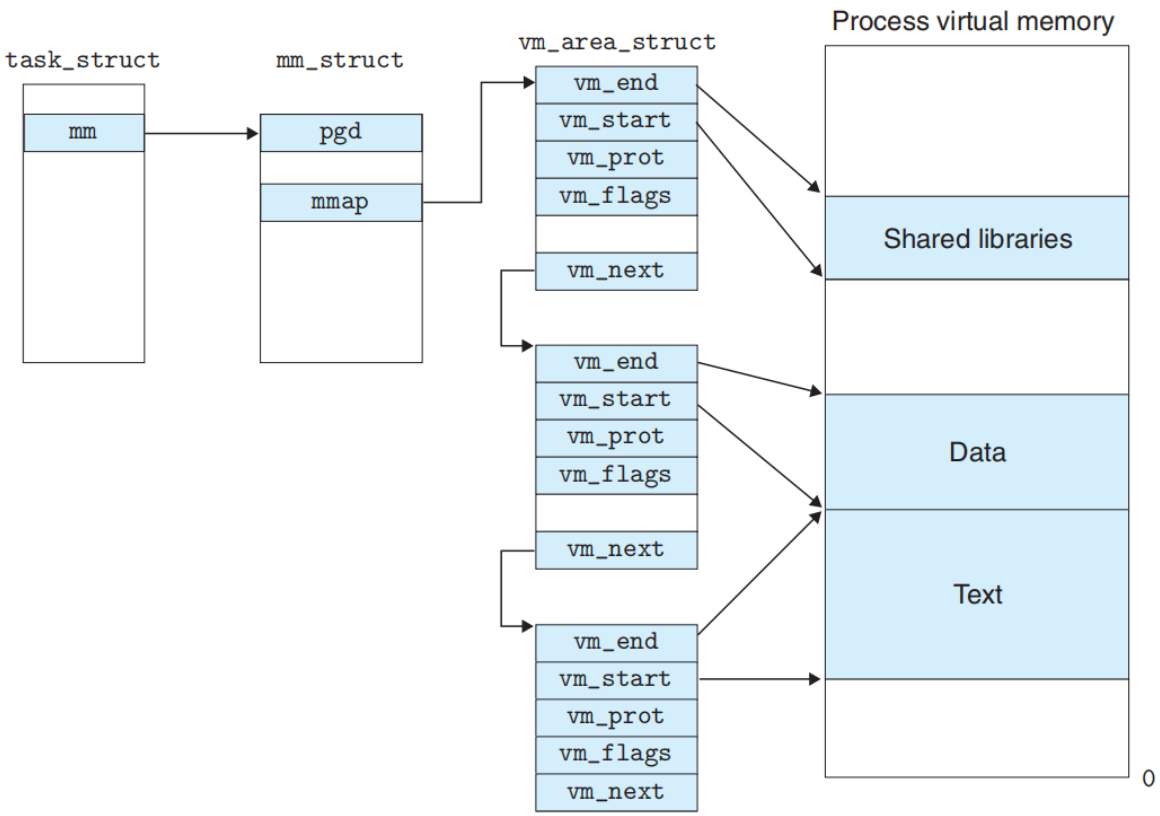


Figure 9.27 **How Linux organizes virtual memory.**

`fvm_start`. Points to the beginning of the area.

`vm_end`. Points to the end of the area.

`vm_prot`. Describes the read/write permissions for all of the pages contained in the area.

`vm_flags`. Describes (among other things) whether the pages in the area are shared with other processes or private to this process.

`vm_next`. Points to the next area struct in the list.

上图是Linux内核中记录一个进程虚拟内存区域的数据结构。Linux kernal为每个进程维护了一个单独的任务结构`task_struct`（里面有很多很多东西，比如PID，指向用户栈的指针，程序计数器……），其中一个条目指向了`mm_struct`，它描述了虚拟内存的当前状态。

- `pgd`：指向第一级页表的基地址（当被运行时，该值会被存放在CR3）中
- `mmap`：指向一个`vm_areas_struct`的链表；链表中每一项都描述了当前虚拟地址空间中的一个区域。

🍎Linux是如何处理Page Fault的?

- Linux试图翻译虚拟地址A时，触发Page Fault。于是导致控制转移到Linux Kernal中的handler;
- 检查A是否合法?
  - 搜索链表，搜索A是否在某个`vm_start`和`vm_end`之间;
  - 若不然，handler触发段错误，终止进程。
- 检查试图进行的内存访问是否合法?
  - 该进程是否有读/写/执行这个区域内页面的权限?
  - 若不然，触发保护异常，终止进程。
- 到这一步，Linux kernal终于知道这个地址确实是合法的了！它选择一个牺牲页面（若有必要），如果该页面被修改过，就将其先换出。然后换入新的页面，更新Page Table。
- handler返回，将控制权重新交给导致Page Fault的程序，之前导致缺页的指令再次执行，MMU再次翻译A——这一次，MMU成功了：）

6. Take Away Message

虚拟内存是对主存的一个抽象。支持虚拟内存的处理器通过使用一种叫做虚拟寻址的间接形式来引用主存。处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件和软件紧密合作。专门的硬件通过使用页表来翻译虚拟地址，而页表的内容是由操作系统提供的。

虚拟内存提供三个重要的功能。第一，它在主存中自动缓存最近使用的存放磁盘上的虚拟地址空间的内容。虚拟内存缓存中的块叫做页。对磁盘上页的引用会触发缺页，缺页将控制转移到操作系统中的一个缺页处理程序。缺页处理程序将页面从磁盘复制到主存缓存，如果必要，将写回被驱逐的页。第二，虚拟内存简化了内存管理，进而又简化了链接、在进程间共享数据、进程的内存分配以及程序加载。最后，虚拟内存通过在每条页表条目中加入保护位，从而了简化了内存保护。

地址翻译的过程必须和系统中所有的硬件缓存的操作集成在一起。大多数页表条目位于 L1 高速缓存中，但是一个称为 TLB 的页表条目的片上高速缓存，通常会消除访问在 L1 上的页表条目的开销。

现代系统通过将虚拟内存片和磁盘上的文件片关联起来，来初始化虚拟内存片，这个过程称为内存映射。内存映射为共享数据、创建新的进程以及加载程序提供了一种高效的机制。应用可以使用 mmap 函数来手工地创建和删除虚拟地址空间的区域。然而，大多数程序依赖于动态内存分配器，例如 malloc，它管理虚拟地址空间区域内一个称为堆的区域。动态内存分配器是一个感觉像系统级程序的应用级程序，它直接操作内存，而无需类型系统的很多帮助。分配器有两种类型。显式分配器要求应用显式地释放它们的内存块。隐式分配器(垃圾收集器)自动释放任何未使用的和不可达的块。

对于 C 程序员来说，管理和使用虚拟内存是一件困难和容易出错的任务。常见的错误示例包括：间接引用坏指针，读取未初始化的内存，允许栈缓冲区溢出，假设指针和它们指向的对象大小相同，引用指针而不是它所指向的对象，误解指针运算，引用不存在的变量，以及引起内存泄漏。