Virtual Memory (IV)

swap, system, mmap
(OSTEP Sec21，CSAPP e3 9-7、9-8)

Outline

- Swap
- Practical System Examples
- Memory Mapping

我们讲 page fault 时，大家有没有想过磁盘上的 page 是从哪来的?

Swap

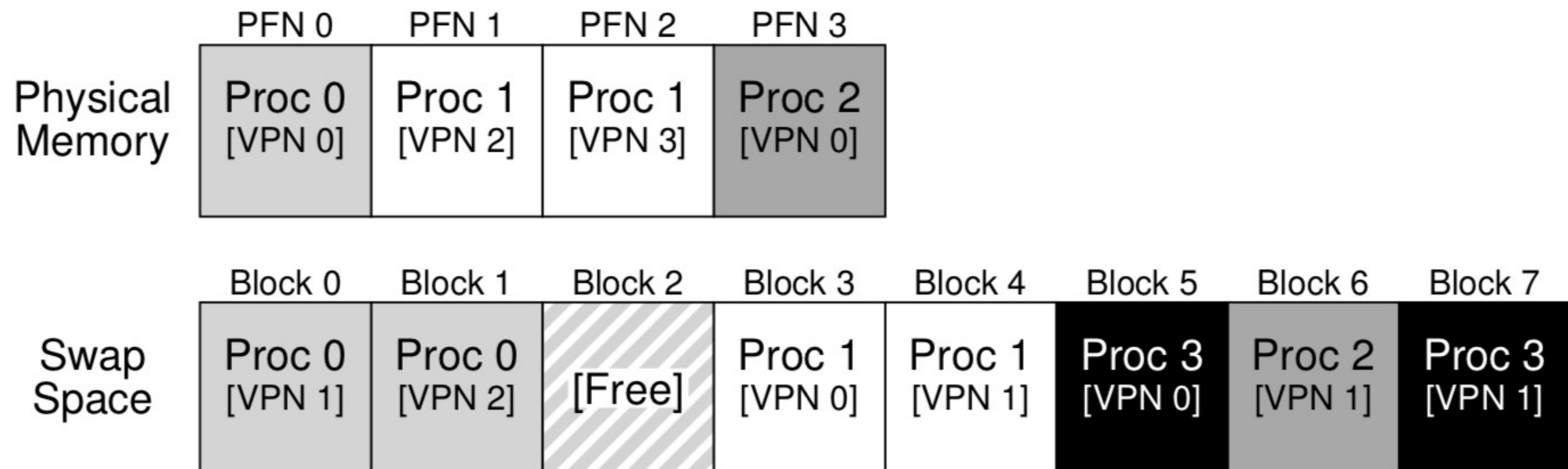
- OS 中所有进程的虚存空间总和可能大于物理内存
  - 进程 Vm 空间大的好处：
    - Data structure 不用考虑 memory 空间的限制
    - 不过实际上完全不考虑也是不负责任的，例如外排算法（当数据量非常大时）
  - 不能保证每个 VPN 都对应一个 PPN
  - 资源不够时，只能将部分内存中的内容交换到磁盘上

Swap

- Swap space
  - 磁盘上需要保留一定空间，专门用来存放交换出来的内存中的内容
  - Swap space 大小直接决定 Vm 空间的大小
  - 一般假设 swap 空间足够

Swap

- Swap space example
  - 4-page physical memory
  - 8-page swap space



Page-Fault Control Flow (Hardware)

- 内存访问流程
  - 要访问的 VA 翻译成 PA，在此过程中可以确定要访问的 VA 是否在物理内存 RAM 中
    - PTE 中有标记 page 是否在 RAM 的标记位
  - 不在 RAM 中则触发 Page Fault Exception

Page-Fault Control Flow (Software)

- page fault handler 负责将 page 读入 RAM，更新 PTE，再返回并重新执行访存指令
- 对 Page-Fault 的处理由软件 (handler) 完成
  - 因为需要进行 disk-io，比较慢，没有必要用快速硬件
  - 需要了解 swap space，与 disk 打交道处理 I/O 请求，还有很多细节，这些对于 hardware 来说过于复杂

Page-Fault Control Flow (Software)

- 如果此时物理内存满了，需要将一个内存中的 page 换出到 swap space
  - cache replacement policy （之后讲到）
- EvictPage() 执行该换出操作

```
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
```

实际上 Linux 不会等到物理内存完全满了才进行 swap (swap daemon)

When Replacements Really Occur

- **high watermark** (HWM)
- **low watermark** (LWM)
  - 当 free page 数量少于 LWM 时，一个后台线程负责淘汰 page，直到 free page 数量达到 HWM
- 后台操作的好处:
  - to increase efficiency
  - to allow for grouping of operations
    - 磁盘 I/O 粒度更大，性能更好

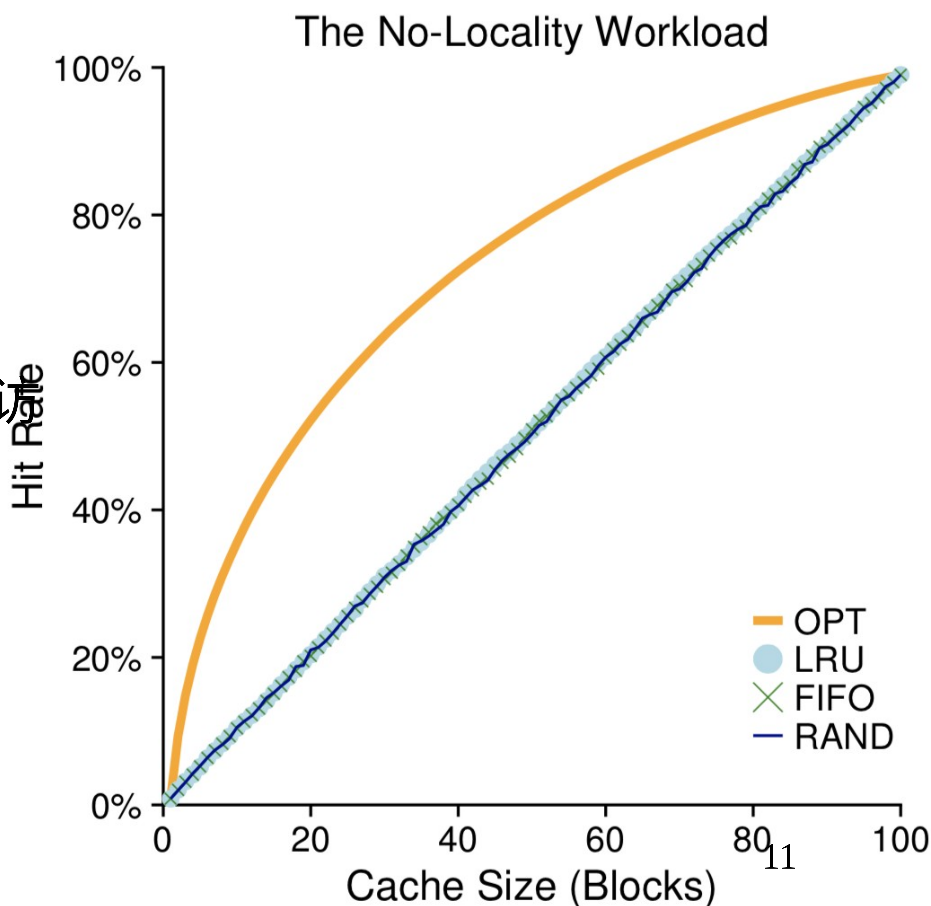
后台批量处理的方式在日常生活中也很多

Replacement Policy

- OPT
- FIFO
- Random
- LRU

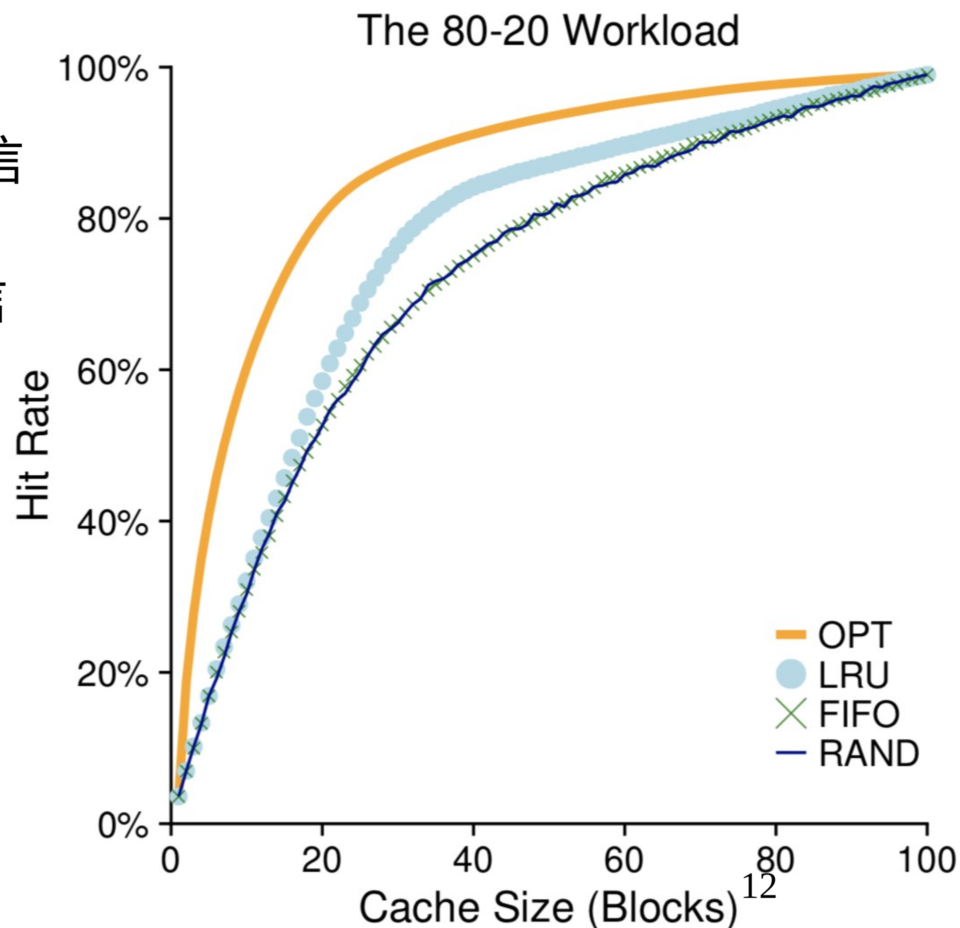
Replacement Policy

- **The No-Locality Workload**
  - The hit rate exactly determined by the size of the cache.
  - Optimal performs noticeably better than the realistic policies (知道未来访问信息)



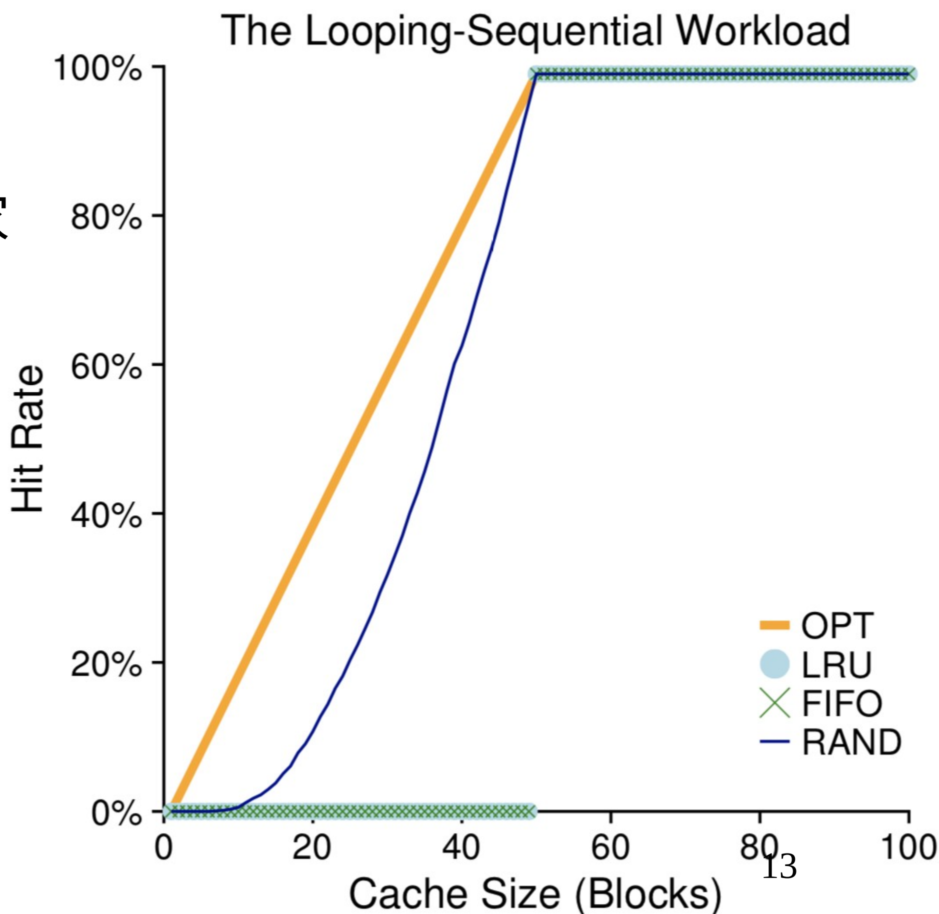
Replacement Policy

- **The 80-20 Workload**
  - 80% of the references are made to 20% of the pages
  - LRU does better (利用历史信息找到 hot pages)
  - OPT 最好，说明 LRU 的历史信息不是完美的



Replacement Policy

- **The Looping-Sequential Workload**
  - 循环遍历一个子集，例如 page 1->50 循环遍历 2 遍
  - Cache size 大于 50 时，大家命中率都是 100%
  - Cache size 小于 50 时，FIFO 和 LRU 很差，因为留下最近刚访问的数据在这里是最差的策略（越老的数据越可能被访问）
  - Random 反而会比较好



Implementing Historical Algorithms

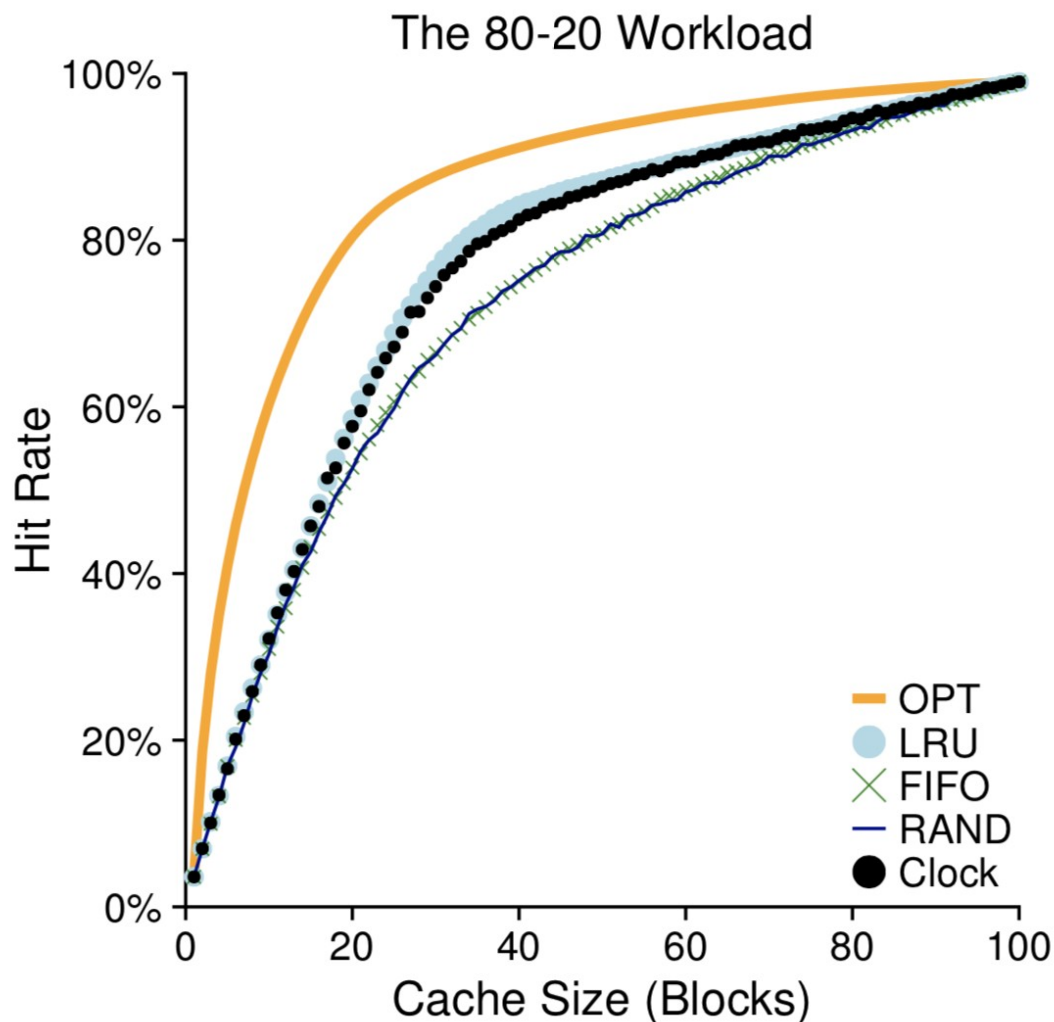
- 实现 LRU 需要很多工作:
  - 每次 page access ，需要更新一个数据结构（链表 / 队列等），将访问的页移动到最前面
  - 而 FIFO ，则不需要每次访问都更新数据结构；只是在 replacement 时修改即可（ replacement 数量比 page access 要少很多）
  - Ps. LFU 需要记录访问次数，这个开销更大，一般系统中都不会使用 LFU 这种复杂度的方法
  - 应用层软件系统中通常使用 LRU 或同等复杂度的策略，底层系统（操作系统、数据库）可能还嫌 LRU 复杂度高

Approximating LRU

- Clock 算法
  - 用 circular list 管理所有的 page，有一个 clock hand 指向当前某个 page
  - 用一个 use bit 来近似 LRU （可以用 hardware 实现）
    - 被访问的 page, use bit 设为 1；
    - 替换时， clock hand 指向的 Page P 如果 use bit=0，则替换；否则将 P 的 use bit 设为 0，继续查看下一个 page P+1
  - 如果一个 page 很久没访问，那么 use bit=0；如果一个 page 频繁被访问，那么 use bit=1
  - 不用执行 LRU 的链表 / 队列操作

Approximating LRU

- Clock 算法效果
  - 接近于 LRU，复杂度更低



Approximating LRU

- Clock 算法变种
  - 增加一个 dirty bit，区分是 clean page 还是 dirty page
  - 多个 clock hands
  - 1 use bit -> multiple use bits
    - 每次 --，直到 0 才替换
    - 相当于分级别，记录一定访问次数信息，区分不同数据的 hotness 程度

Prefetching

- OS 管理物理内存资源不仅仅是通过 replacement
- 还包括 prefetching (read_ahead)
  - 开销比较大
  - 需要比较确定（例如顺序访问）
  - 有很好的加速效果

Practical System Examples

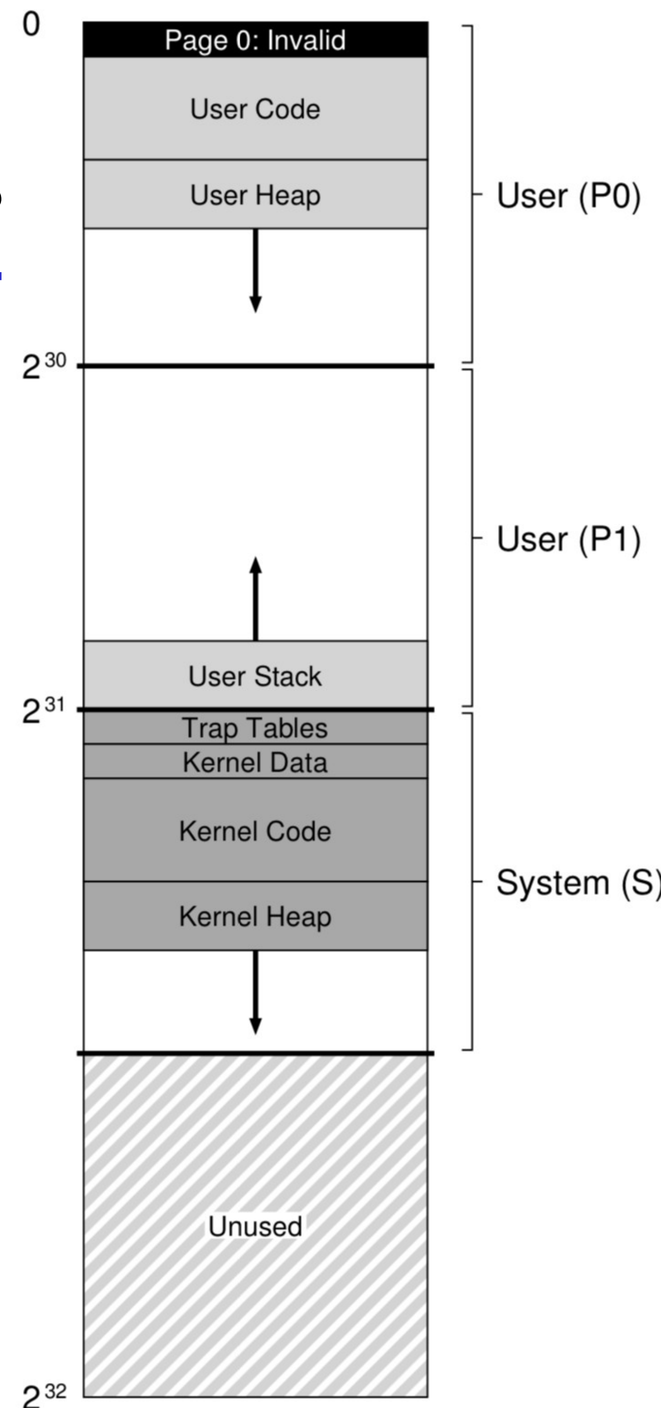
VAX/VMS, Intel i7, Linux

The VAX/VMS Virtual Memory System

- VAX-11
  - 1970's, **Digital Equipment Corporation** (**DEC**)
    - DEC 由于一系列决策失误，以及 PC 的兴起而失败
  - OS 叫做 VAX/VMS
    - Dave Culter 是主要作者之一，后来领导 MS 的 Windows NT 项目
    - 运行在一系列不同档次（差异极大）的计算机上
    - How to avoid "the curse of generality"?

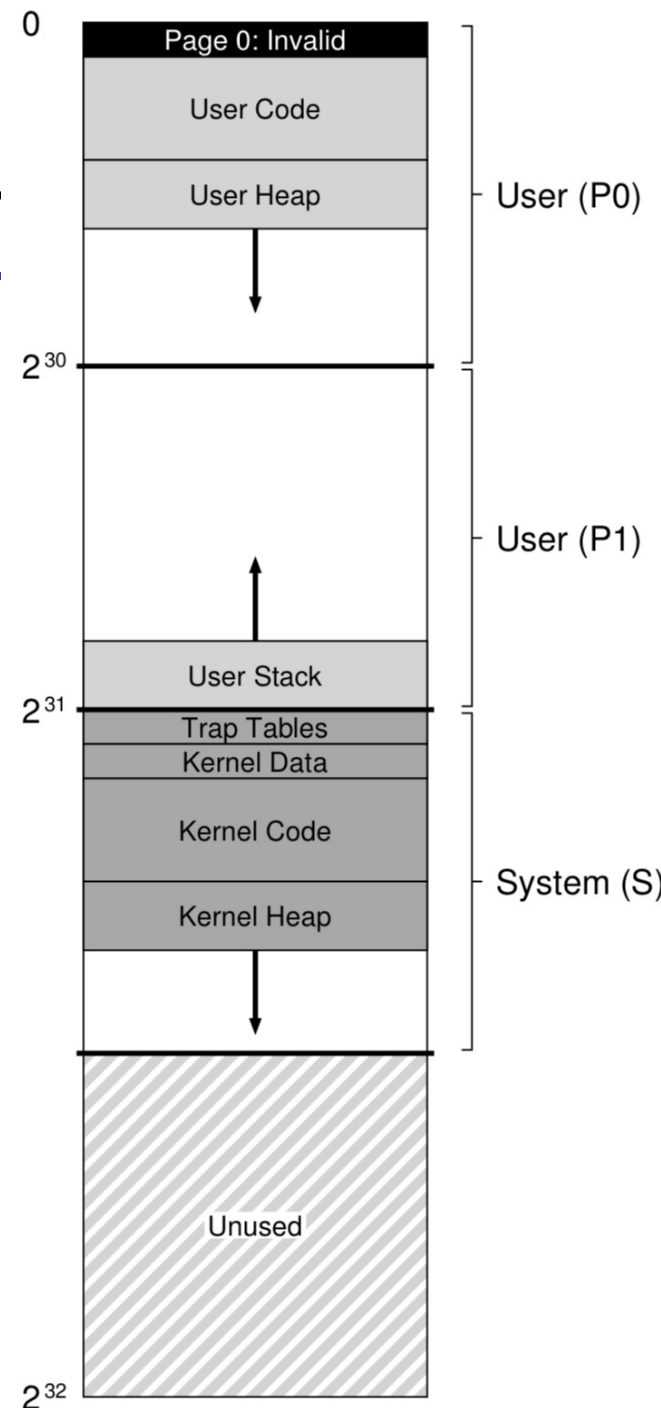
The VAX/VMS Virtual Memory Sys

- VMS Virtual Address Space
  - 32-bit VA space per process
  - 512-byte page
  - VPN: 23-bit, offset: 9-bit
  - VPN 的开头两位标记 segment
  - 低位的一半地址空间为用户态程序使用
    - P0: 包含 code, heap
    - P1: 包含 stack
  - 高位的一半为 kernel (system) 使用



The VAX/VMS Virtual Memory Sys

- Page 较小，如何减小 page table 空间?
  - #1. P0 和 P1 各有一个 page table，采用前面的 hybrid （段页式）方案， heap 和 stack 之间未使用的内存不记入 page table
  - #2. 把 P0 和 P1 的 page table 放到 kernel 的内存段
    - 挤占 kernel 内存资源
    - 如果物理内存满了， kernel 会把自己的一部分数据交换到 swap sapce



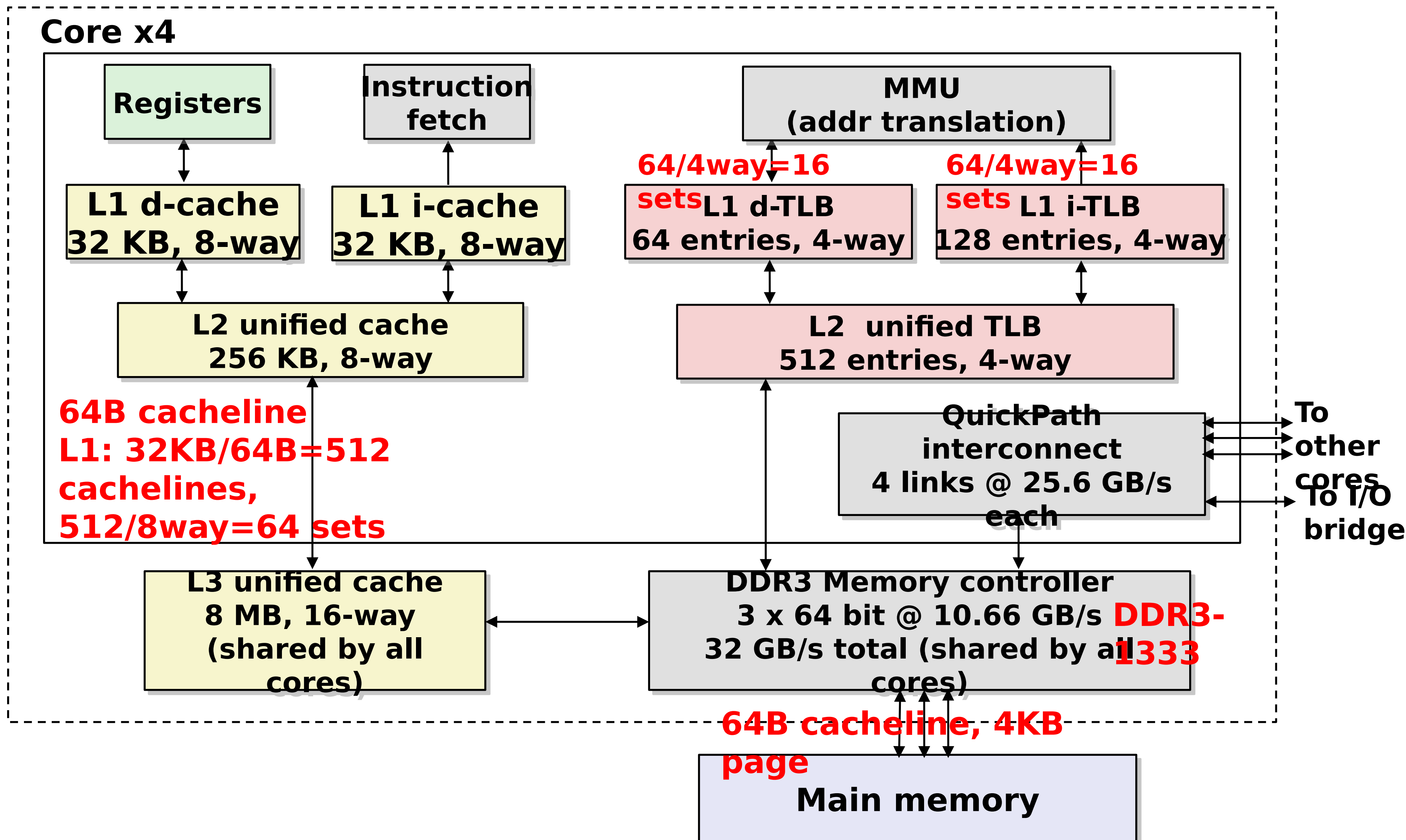
The VAX/VMS Virtual Memory System

- The page table entry (PTE) in VAX contains:
  - a valid bit
  - a protection field (4 bits)
  - a modify (or dirty) bit
  - a field reserved for OS use (5 bits)
  - and finally a physical frame number (PFN) to store the location of the page in physical memory
  - no **reference bit,** 不能用硬件来辅助 replacement

The VAX/VMS Virtual Memory System

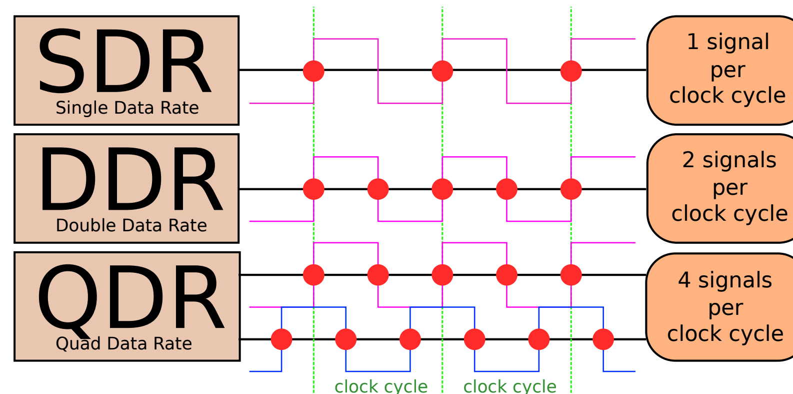
- Replacement scheme
  - **Segmented FIFO**
    - 每个进程占有物理内存页面存在上限： **resident set size** (**RSS**)
    - 每个进程内部按照 FIFO 规则替换
  - **second-chance lists**
    - 为了提升 FIFO 的命中率
    - 从 FIFO 中淘汰的数据放入 a global *clean-page free list* 或 *dirty-page list*
    - 如果另一个进程 Q 要申请 page，先从 clean list 中获取
    - 如果原来的进程 P 再次访问其中 page，则回到 FIFO，避免 disk io
    - Second-chance list 越大，效果越接近 LRU

Intel i7 Haswell (4 代)



DDR

- DDR: Double Data Rate



- DDR3-1333:
  - 667MHz memory clock
  - 1333MT/s transfer rate
  - (1333MT/s)*(64bit/T)=10.66GB/s

李雷买了一根 DDR5-5600 内存，它的带宽多大?
李雷的电脑支持 DDR5-4800 4 通道，它的内存带宽多大?

DDR

内存的带宽在不断提高，但内存的访问延迟并没有显著降低

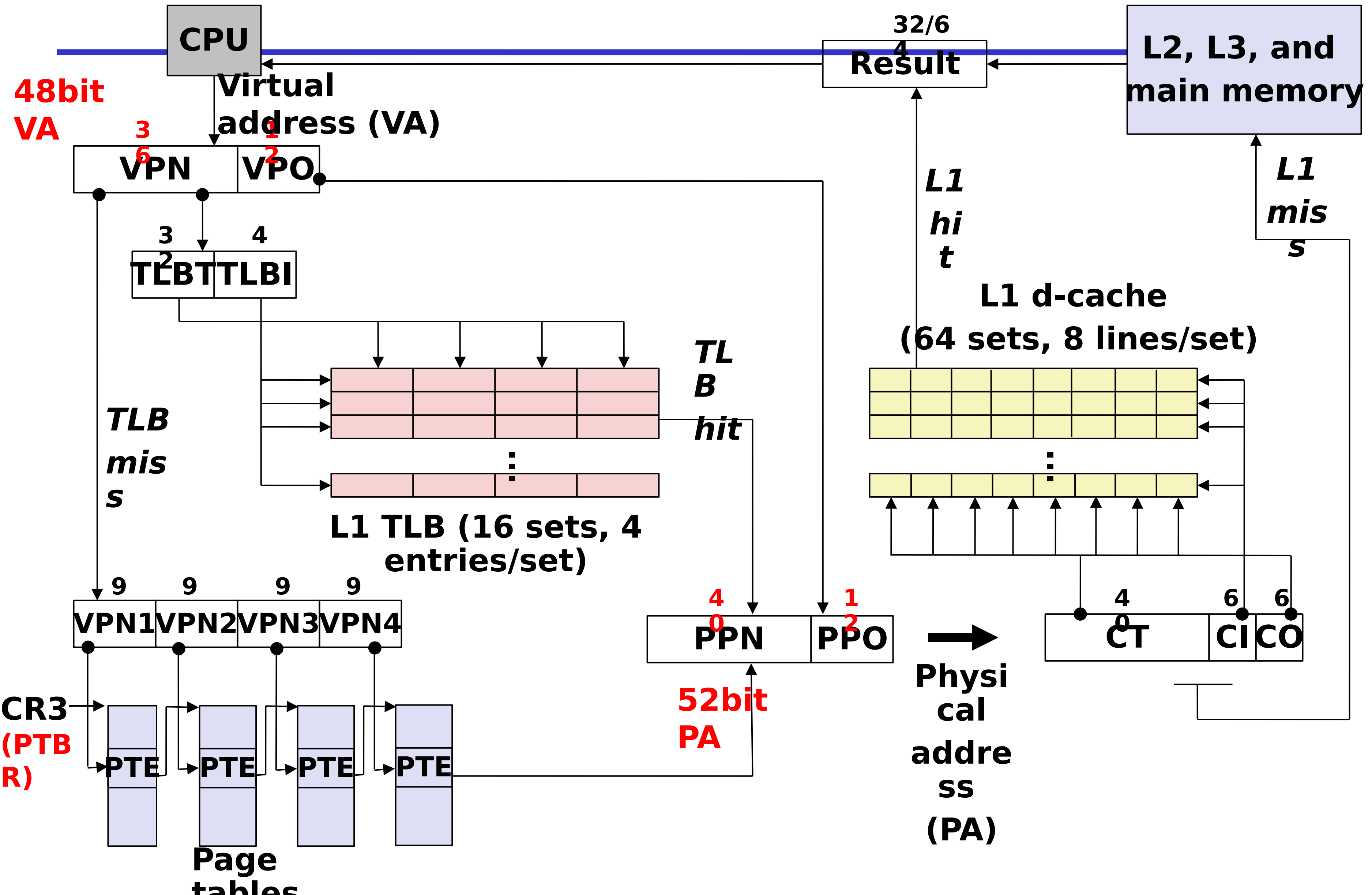
这意味着内存的 IOPS 并没有显著提高

也意味着内存在变得越来越像磁盘：
顺序访问性能和随机访问性能的差距在不断变大
我们在编写程序时，应该尽量提高顺序访问的占比

Review of Symbols

- 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 the virtual address (VA)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

End-to-end Core i7 Address Translation



Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS	P=0

Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

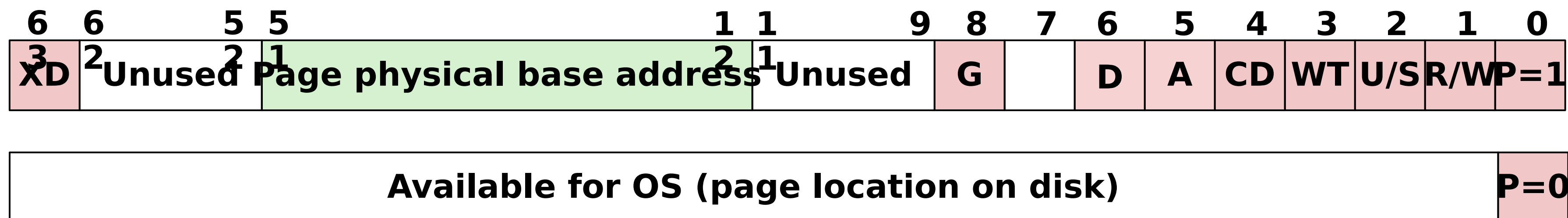
A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size: if bit set, we have 2 MB or 1 GB pages (bit can be set in Level 2 and 3 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 Level 4 Page Table Entries



Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

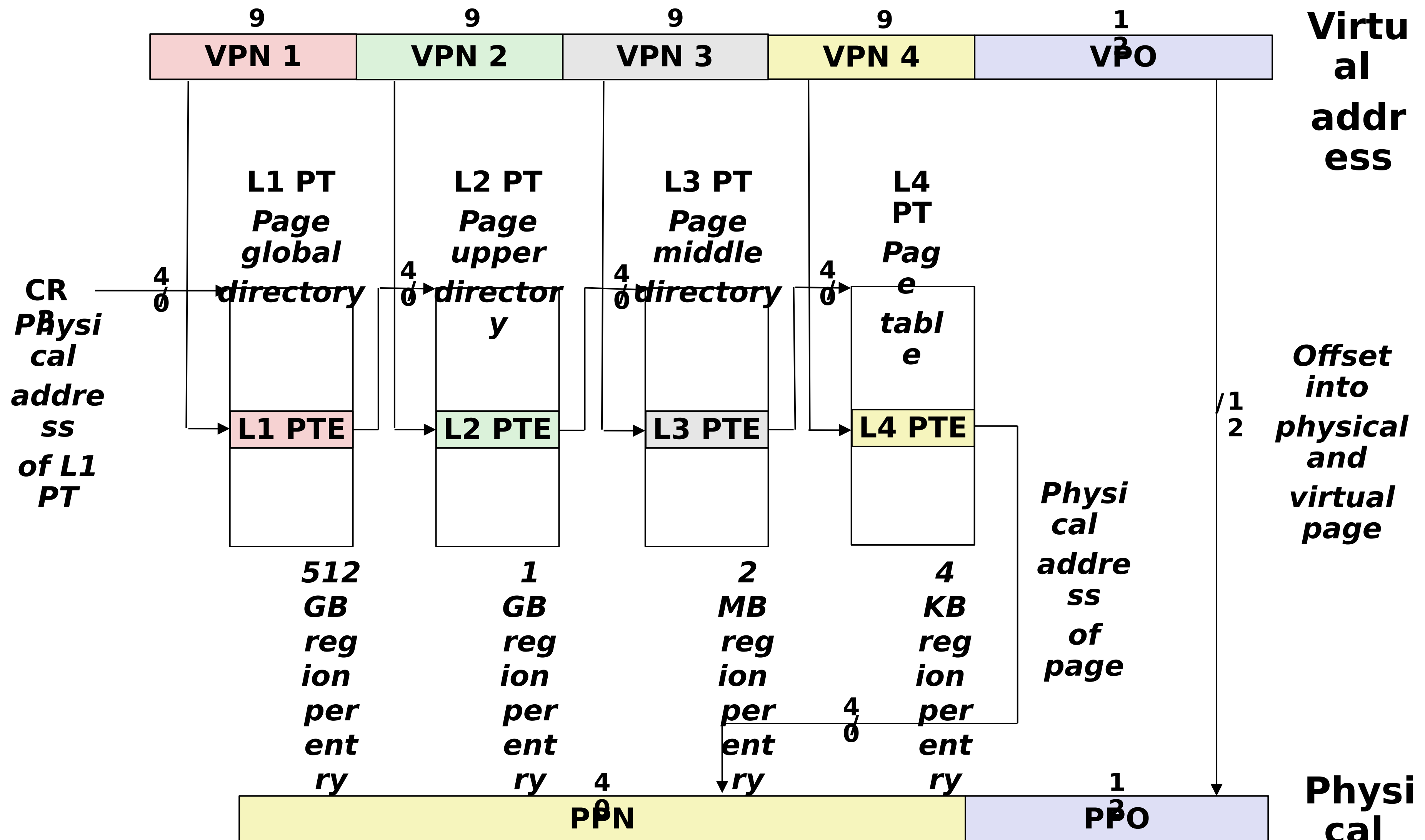
A: Reference bit (**set** by MMU on reads and writes, **cleared** by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Core i7 Page Table Translation

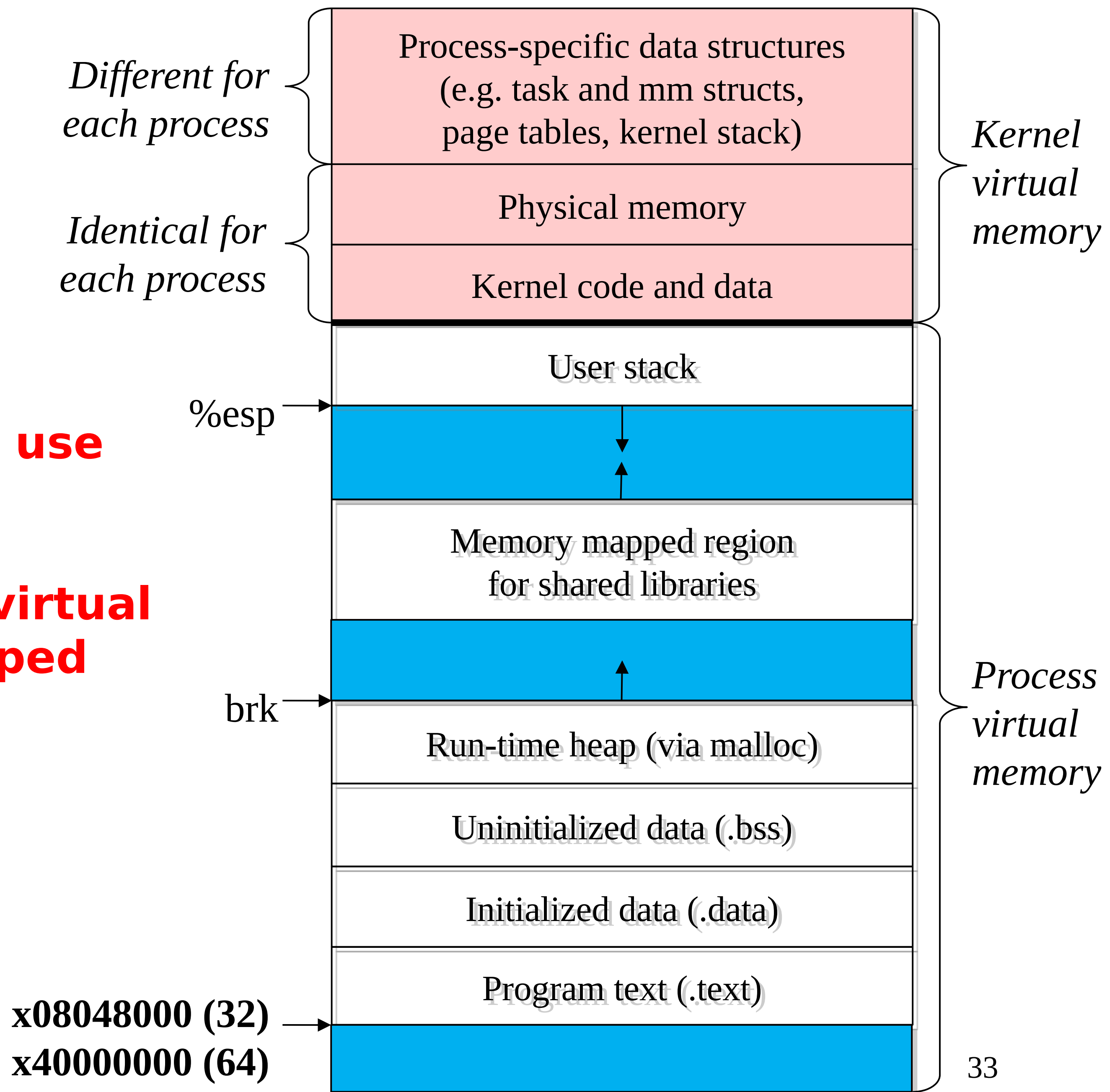


Linux Virtual Memory System

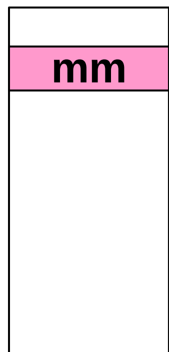
kernel does not use paging

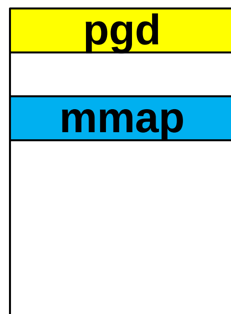
most of kernel virtual memory is mapped to contiguous physical pages

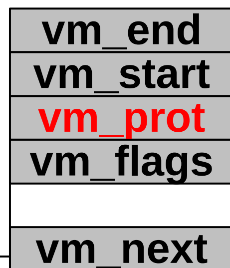
Why?

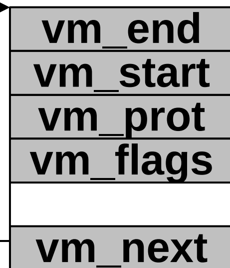


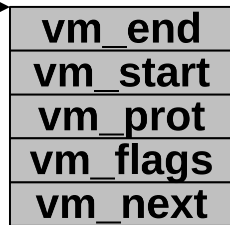
Linux organizes VM as a collection of "areas"

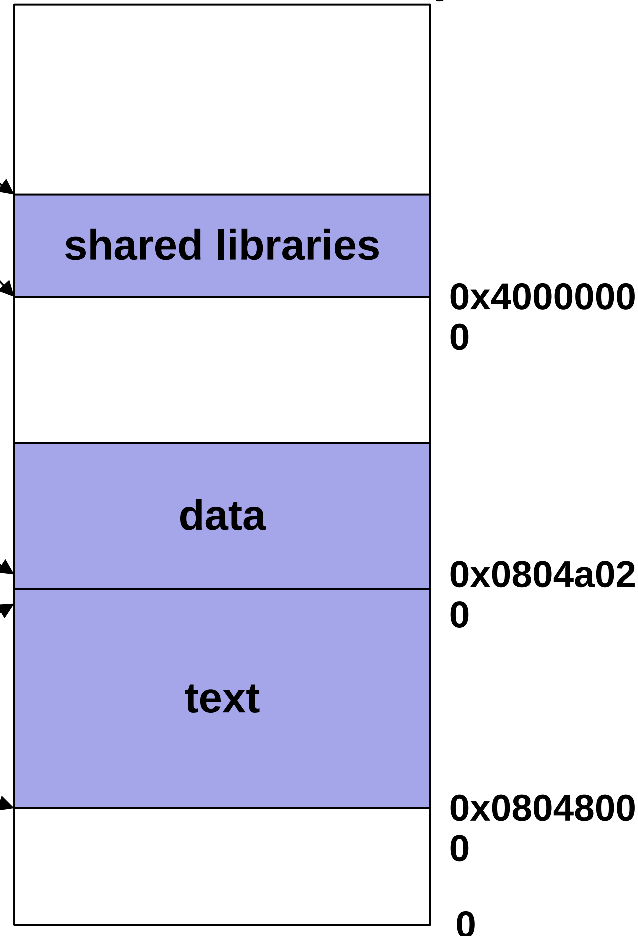






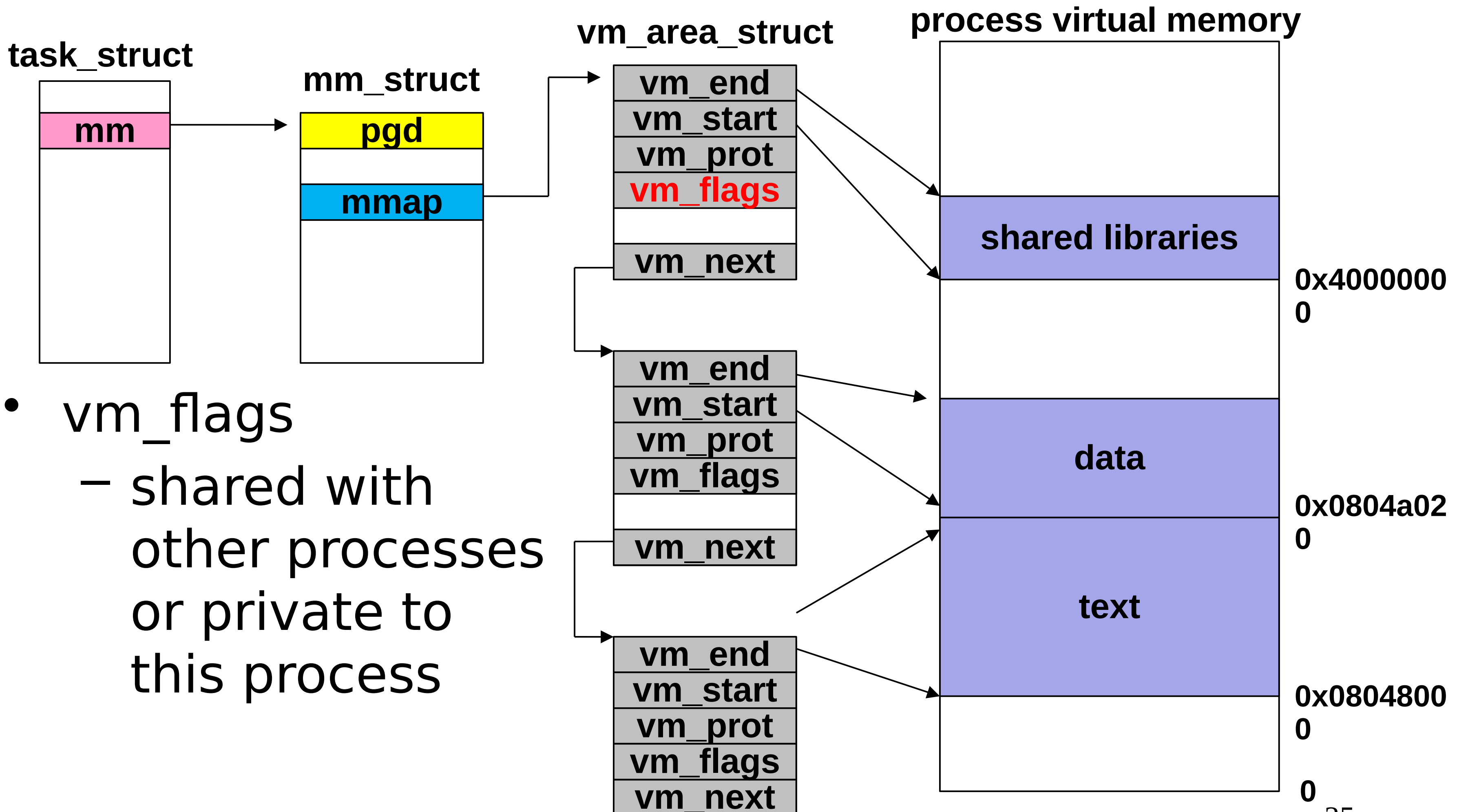






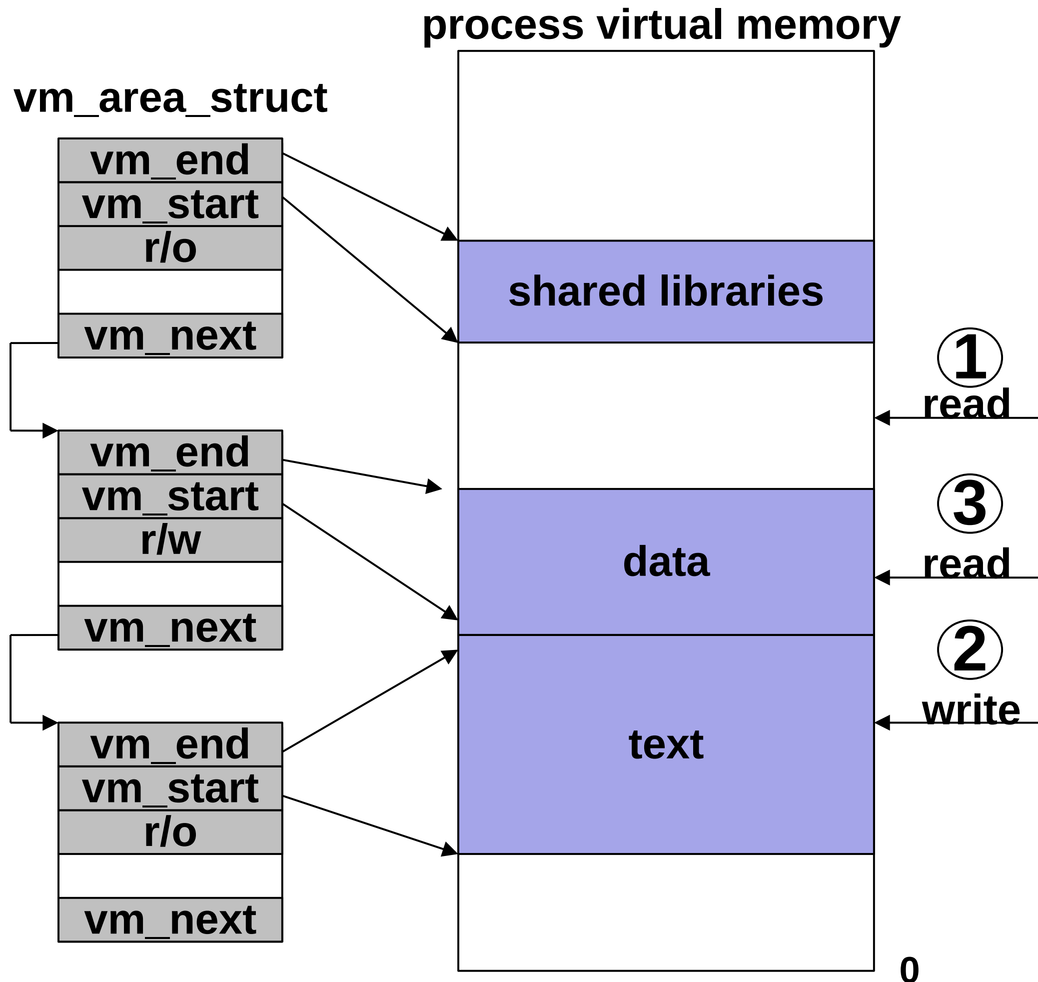
- pgd:
  - page directory address (will be loaded to CR3)
- vm_prot:
  - read/write permissions for this area

Linux organizes VM as a collection of "areas"



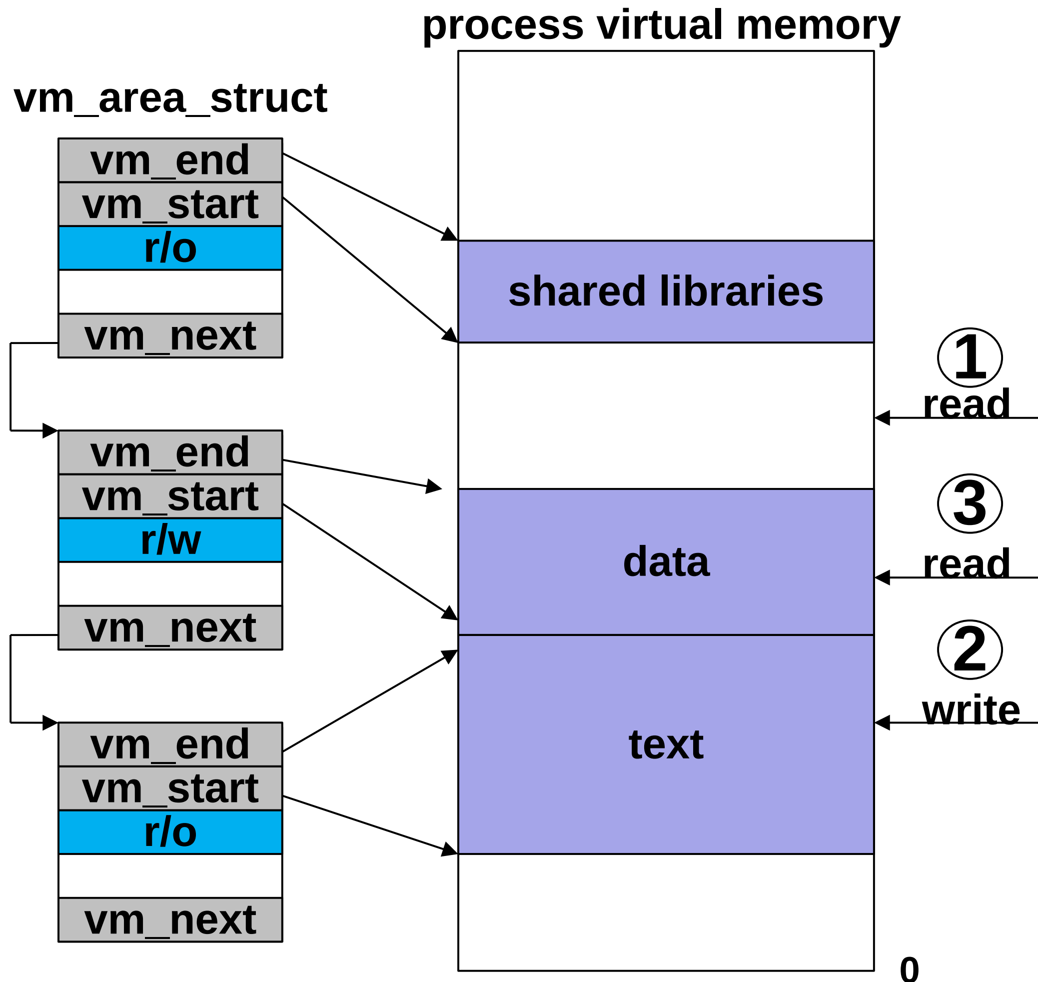
- vm_flags
  - shared with other processes or private to this process

Linux segmentation fault handling



- Is the VA legal?
  - i.e., is it in an area defined by a vm_area_struct?
  - if not then signal segmentation fault (e.g., (1))

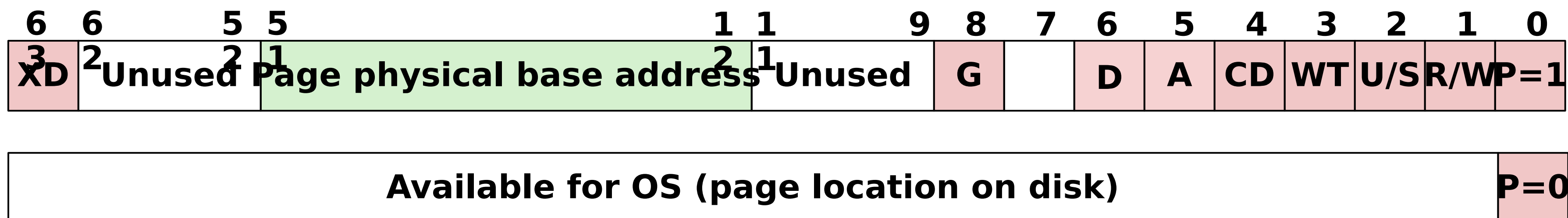
Linux segmentation fault handling



- Is the operation legal?
  - i.e., can the process read/write this area?
  - if not then signal protection exception (e.g., (2))
- If OK, handle fault
  - e.g., (3)

判定内存访问是否越界的方法（intel）

- 页表中的 valid 相当于 present，指是否已经加载到物理内存中，不用于判定越界
- 判定方法：在 P=0 触发 page-fault 的处理中，
  - 根据访问的是哪个段，找到对应的段描述符（专用寄存器），其中包括该段的长度信息，可以判定该次访问是否越界



Memory Mapping (mmap)

Memory mapping

- Linux 可以将内存中的 area 映射到一个文件对象

Memory mapping

- 两种文件对象：
  - regular file
    - 文件按照 page 划分，内存 area 中的 page 和文件中的 page 一一对应
    - 内存 page 的内容来自文件中对应的 page
  - anonymous file (i.e., nothing)
    - 不含任何数据内容
    - 首次访问该 area 中的某个 page 会触发 page fault 以分配 page 的空间

Demand Paging

- No virtual pages are copied into physical memory until they are referenced
  - known as "demand paging"
  - crucial for time and space efficiency

Memory mapping

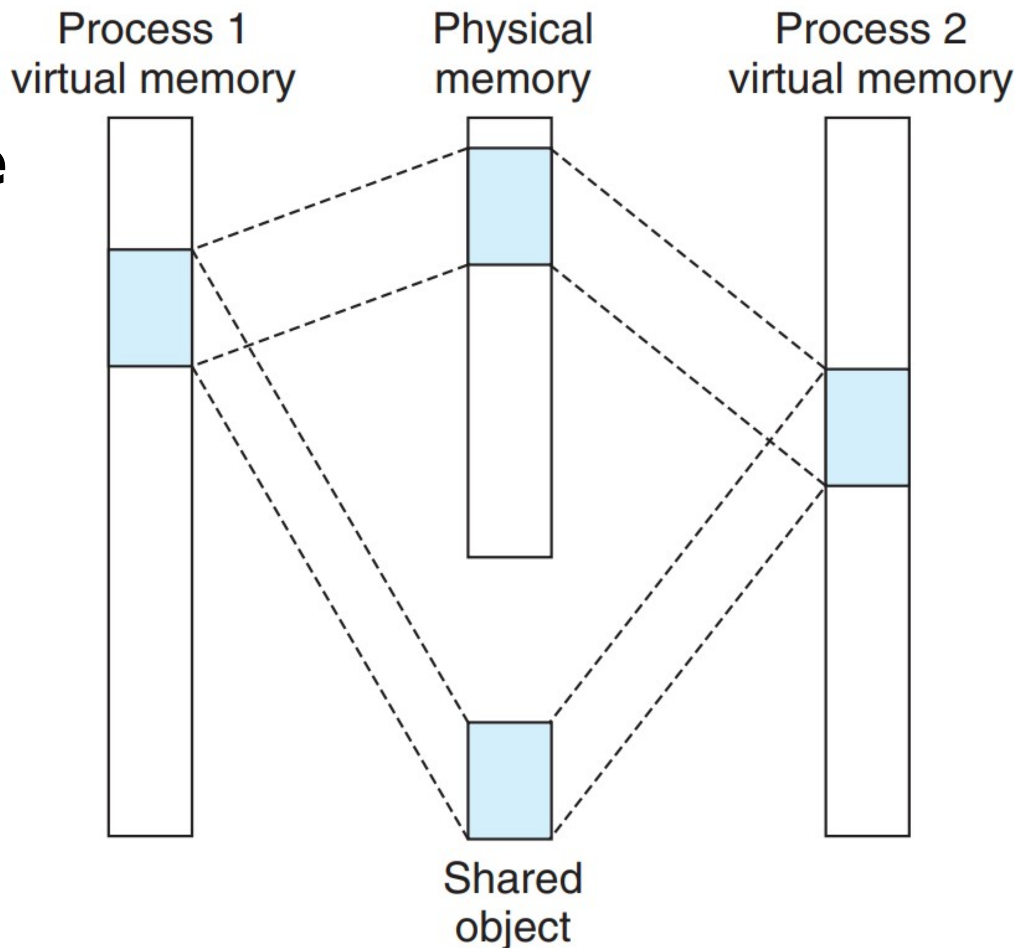
– Memory mapping 本质上是一种将 VM 系统和文件系统集成的方案
– 将 VM 的部分区域映射到文件系统中，可以很方便地实现一些操作：
  – code 和 data 装载
  – shared memory

Shared object

memory mapping 时，可以选择是否 share object

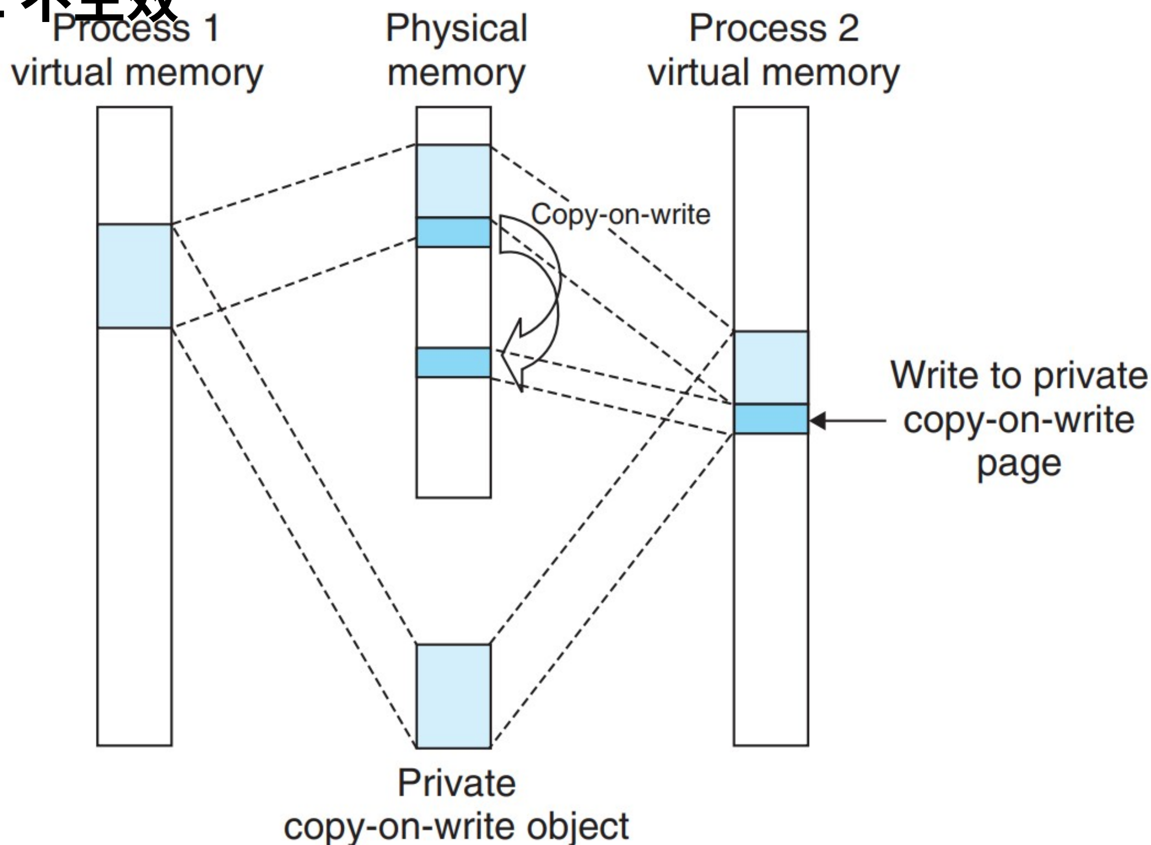
两个进程 map 同一个 shared object，则会共享同一段对应于 file object 的物理内存

两个进程对 mapped memory 的读写互相可见，修改对 object 生效



Private object

private object 类似，但对 mapped memory 的修改会触发 protection fault，并采用 copy-on-write (COW)，对 object 不生效



Memory files

- Linux 中的文件不一定在磁盘等外存中
- Linux kernel 可以将虚拟内存空间模拟成一个文件系统，如 tmpfs 、 ramfs
- 内存文件系统底层没有任何 I/O 设备
- 但内存文件系统中的 page 可能发生 swap
- 可以将内存文件作为 file object 映射到进程地址空间中

Fork() revisted

- **父进程调用 `fork` 时**:
  - kernel 复制父进程的 mm_struct, vm_area_struct, 和 page tables
  - 父子进程此时共享 physical pages

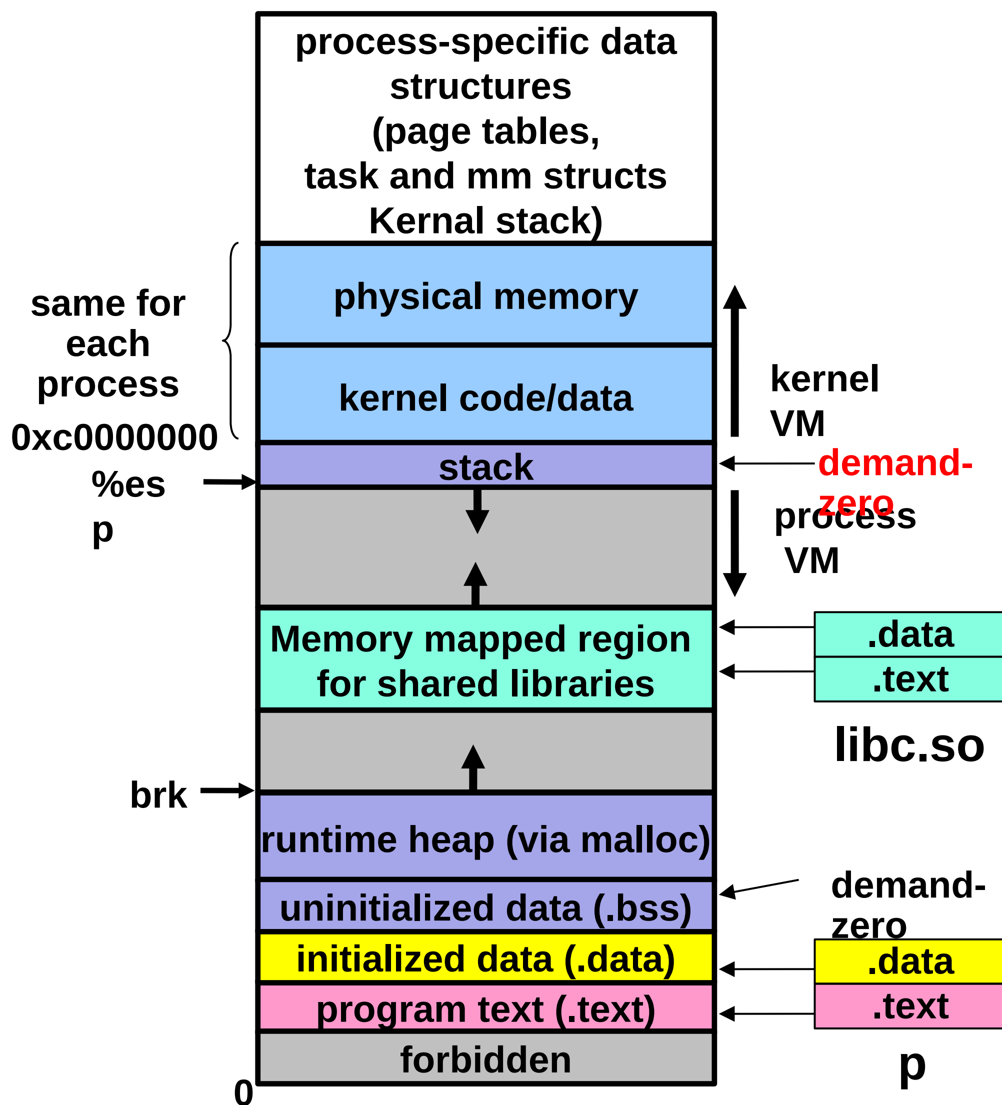
Fork() revisted

- Copy-On-Write
  - 将父子进程的 vm_area_struct 中各个 area 标记为 private “copy-on-write”.
  - 将父子进程的 writable area 中的 page 标记为 read-only
  - 之后父子进程中的写操作会出发 protection fault
    - fault handler 创建 page 的一个副本创建 page 的副本并为其设置写权限

Fork() revisted

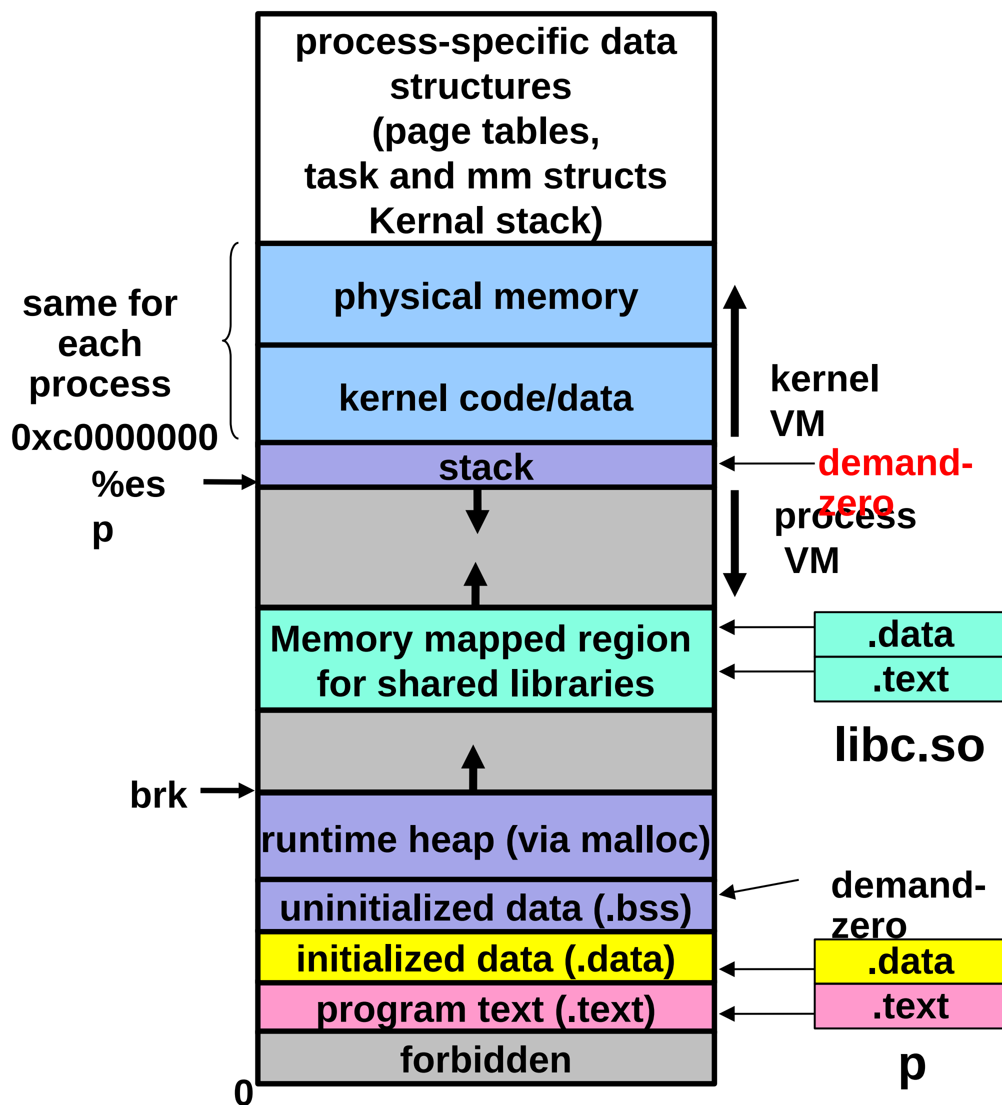
- Result:
  - copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

Execve() revisited



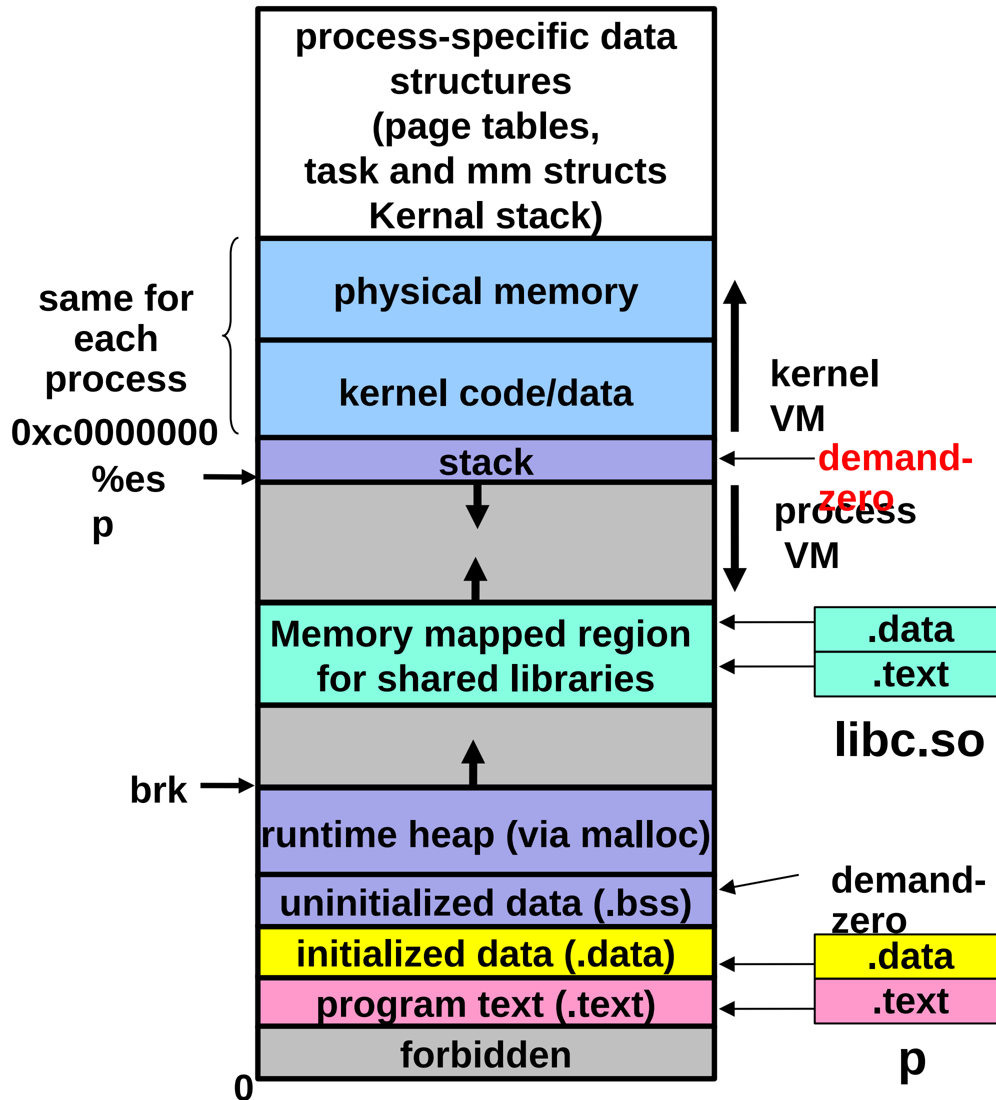
- Execve():
  - 删除 user space 的所有 vm_area_structs 和 PTE
    - stack, bss, data, text, shared libs.

Execve() revisited



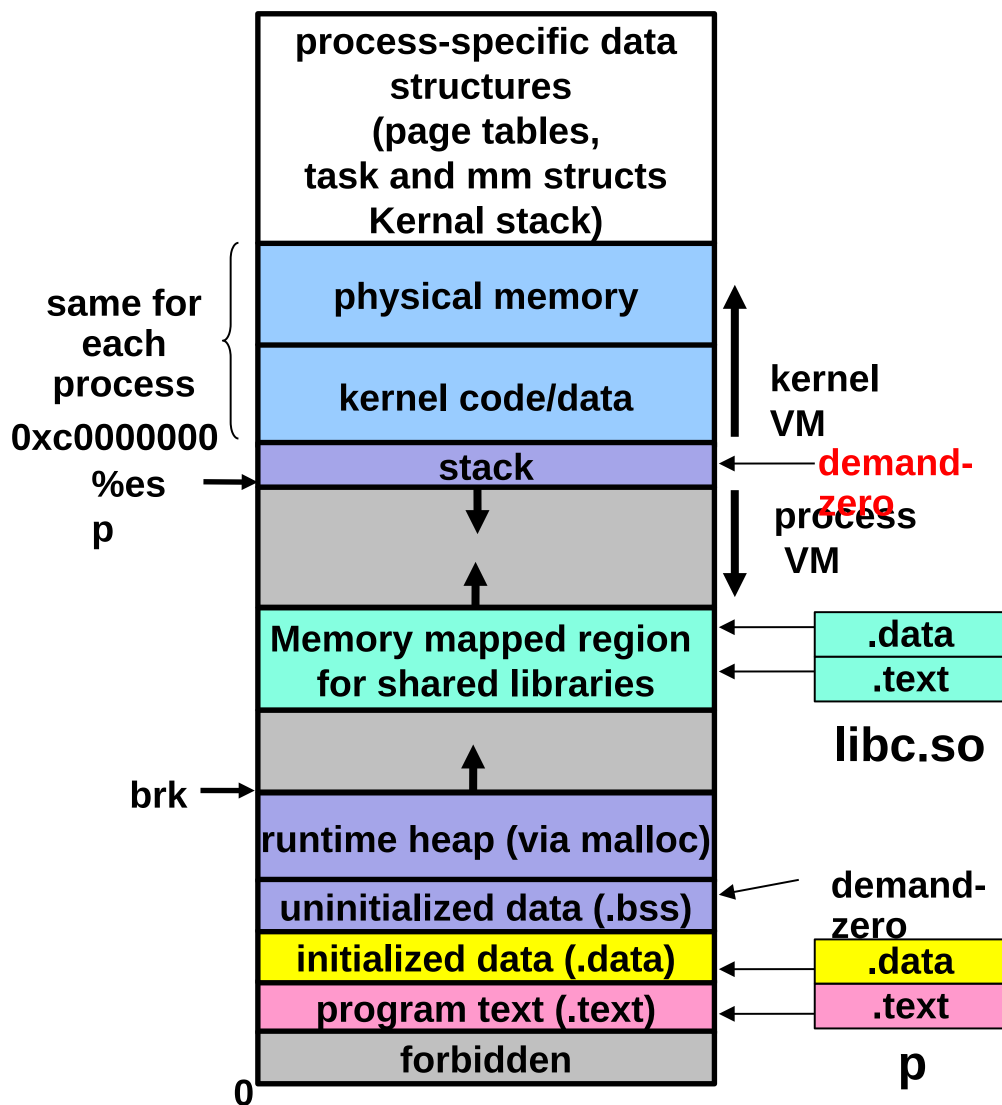
- Execve():
  - 为如下 area 创建新的 vm_area_structs 和 PTE
    - stack, bss, data, text, shared libs.

Execve() revisited



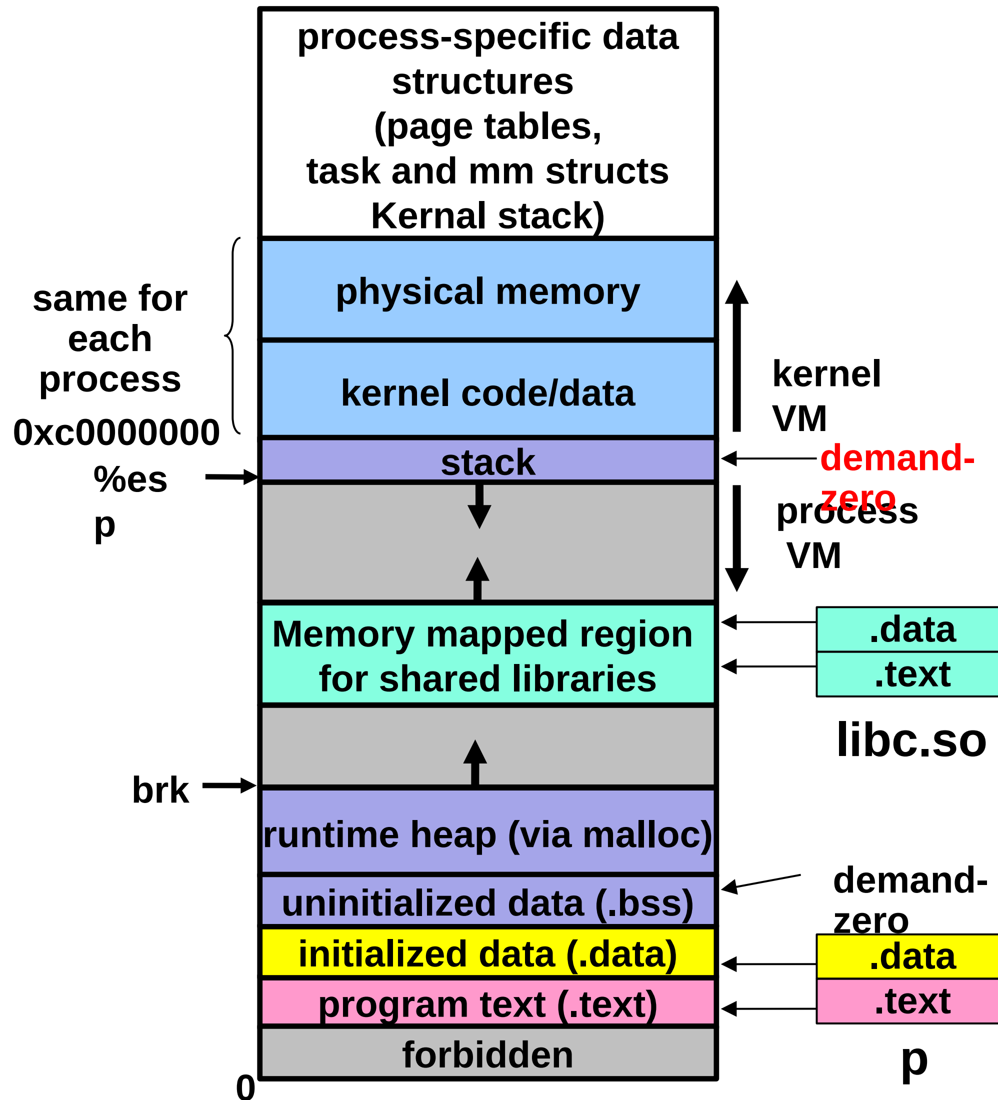
- Execve():
  - 为如下 area 创建新的 vm_area_structs 和 PTE
    - text and data backed by the .text 和 .data sections of the executable object file.
    - bss and stack initialized to zero.

Execve() revisited



- Execve():
  - 共享库以 shared object 映射到进程的地址空间中

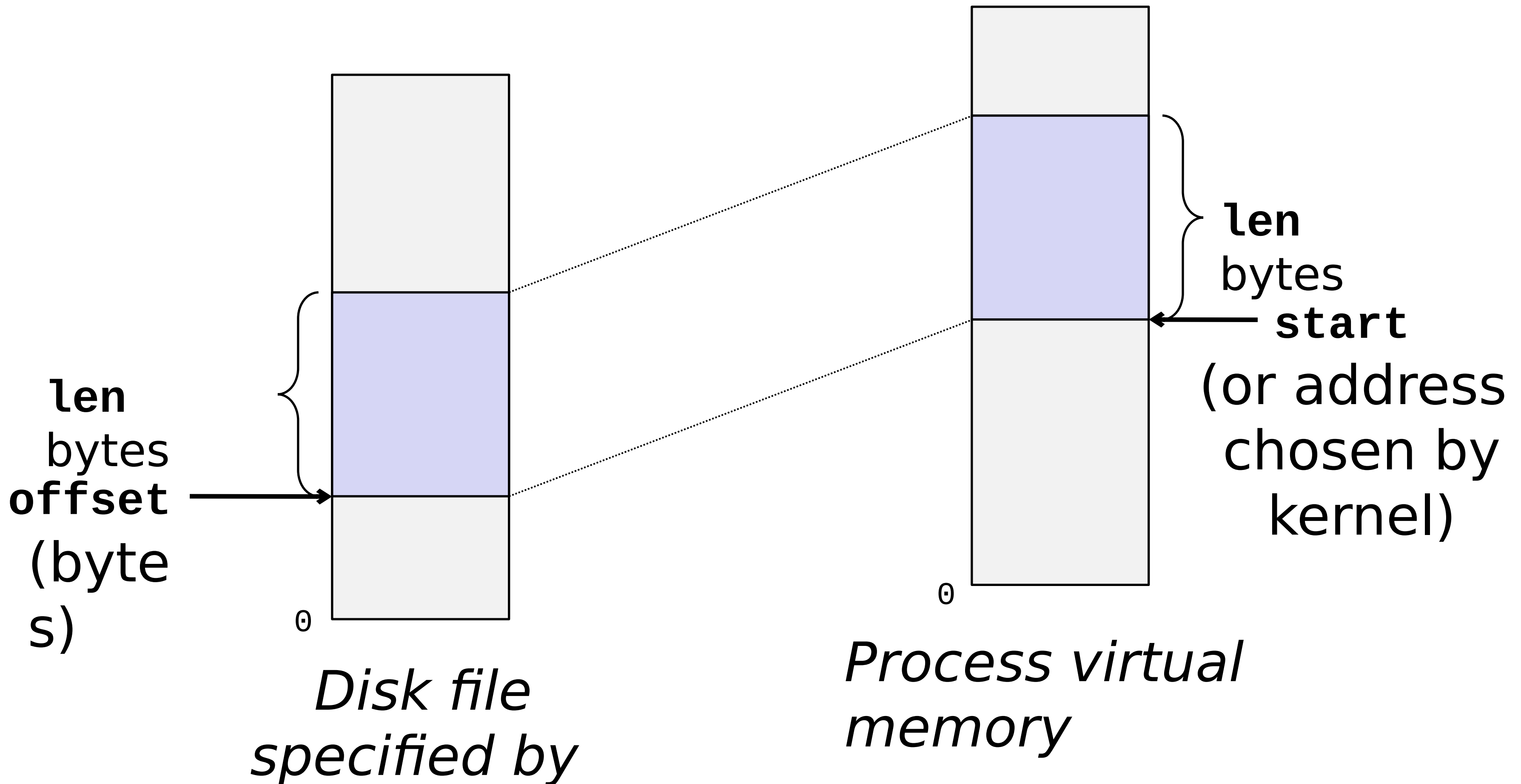
Execve() revisited



- Execve():
  - set PC to entry point in .text
    - Linux will do paging for code and data pages as needed.

User-level memory mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



User-level memory mapping

```
void *mmap(void *start, int len, int prot,

                int flags, int fd, int offset)
```

Map ***len*** bytes starting at offset ***offset*** of the file specified by file description ***fd***, preferably at address ***start*** (usually 0 for don't care).

- **prot**: **PROT_READ, PROT_WRITE**
- **flags**: **MAP_PRIVATE, MAP_SHARED**

Return a pointer to the mapped area

User-level memory mapping

- MAP_SHARED
  - 对应射区域的写入数据会写回文件内，而且允许其他映射该文件的进程共享
  - 但不保证立即写回文件，有一定延迟
  - 显式调用 msync()/munmap() 可以强制刷回文件
- MAP_PRIVATE
  - Create a private copy-on-write mapping
  - 写入内容不写回到磁盘，而是采用 copy on write
  - 对此区域作的任何修改都不会写回原来的文件内容。

User-level memory mapping

- MAP_PRIVATE
  - 对映射区域的修改对其他映射该文件的进程不可见，也不会反映到 file object 中
  - Page 与 file 的关系：只在加载时相关，后面完全无关
  - 常用于共享库的加载

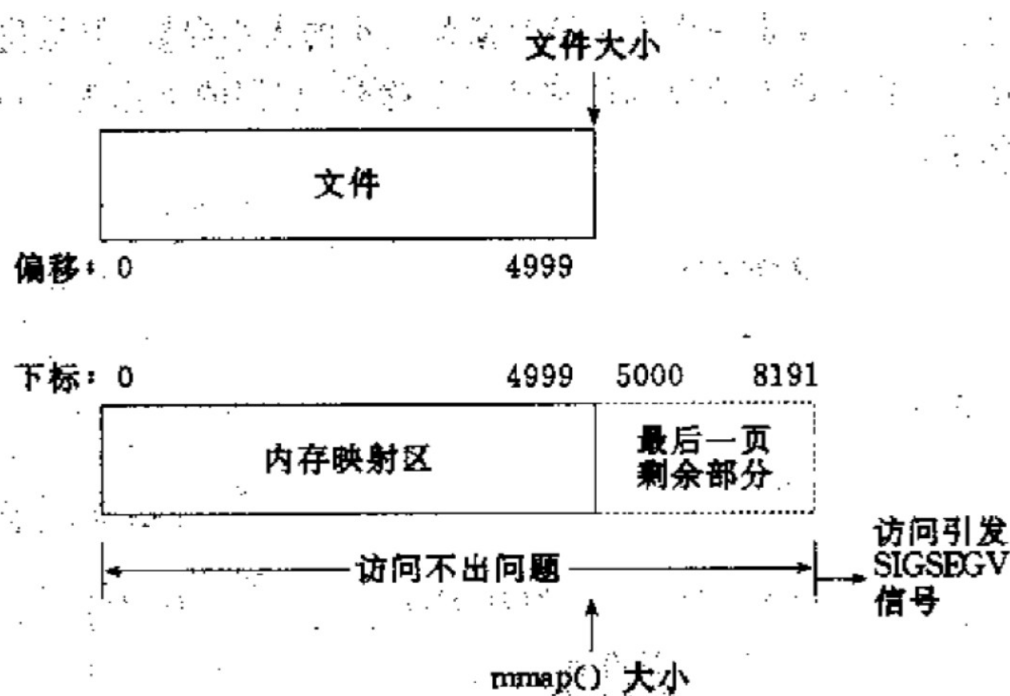
一个 file ，有的进程作为 private 映射，有的进程作为 shared 映射，会如何?

User-level memory mapping

- 使用 mmap 需要注意的关键点:
  - mmap 映射区域大小必须是物理页大小 (page_size) 的整倍数（32 位系统中通常是 4k 字节）；
  - 对于较大的文件， mmap 需要较大的一块连续虚存空间（对应需要访问的文件大小）；而正常 read/write 不需要一次申请这么大的内存 buffer；
  - 如果文件的大小一直在扩张，只要在映射区域范围内的数据，进程都可以合法得到，这和映射建立时文件的大小无关;
  - 映射建立之后，即使文件关闭，映射依然存在。因为映射的是磁盘的地址，不是文件本身，和文件描述符无关

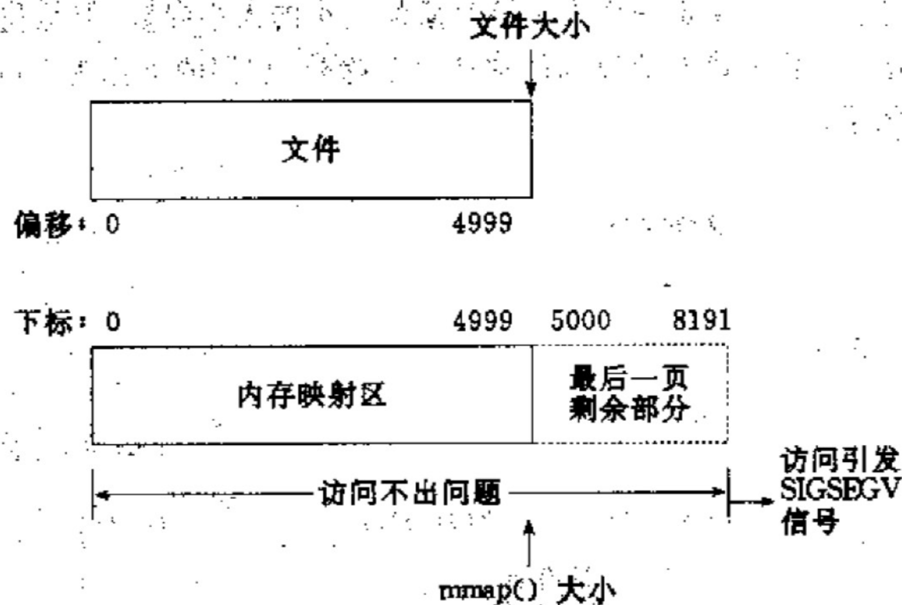
User-level memory mapping

- **例 1．一个文件的大小是 5000 字节， mmap 函数从一个文件的起始位置开始，映射 5000 字节到虚拟内存中**
  - 被映射的文件只有 5000 字节;
  - 5000~8



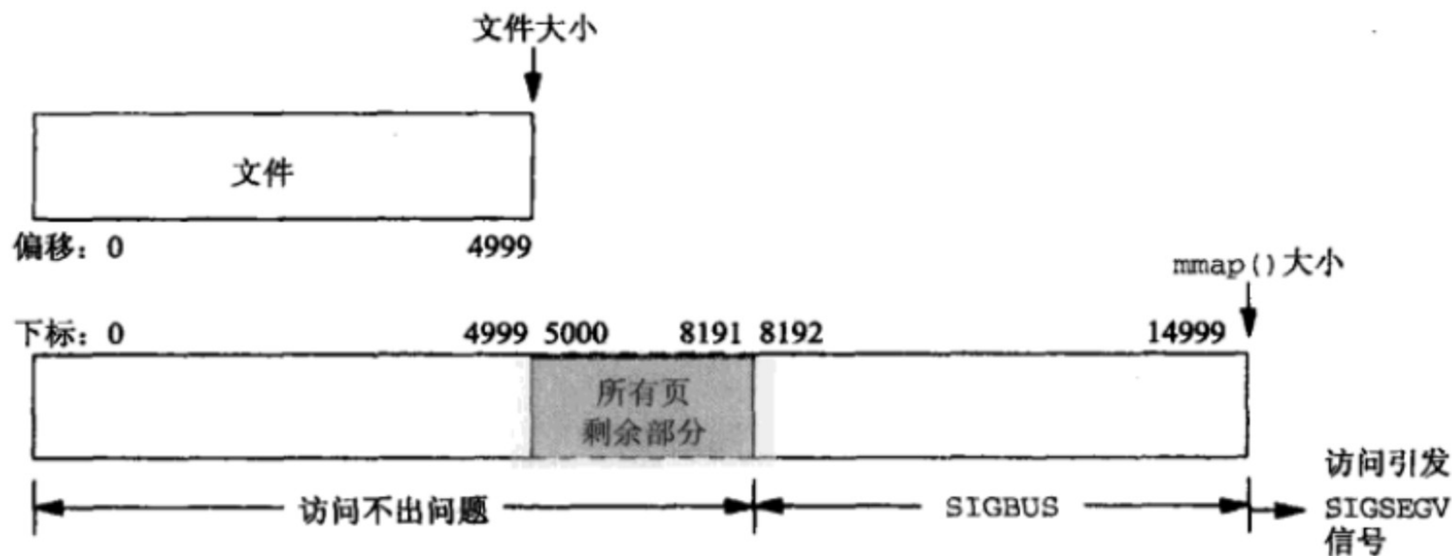
User-level memory mapping

- （1）读 / 写前 5000 个字节（0~4999），会返回操作文件内容
- （2）读字节 5000~8191 时，结果全为 0 。写 5000~8191 时，进程不会报错，但是所写的内容不会写入原文件中
- （3）读 / 写 8192 以外的磁盘部分，会返回一个 SIGSECV 错



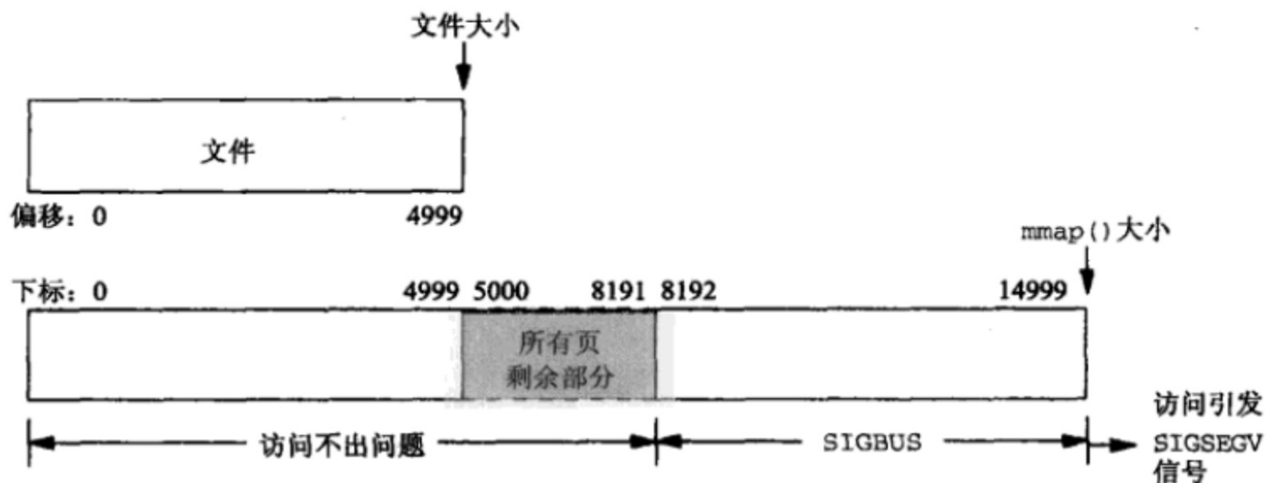
User-level memory mapping

- 例 2．一个文件的大小是 5000 字节， mmap 函数从一个文件的起始位置开始，映射 15000 字节到虚拟内存中，即映射大小超过了原始文件



User-level memory mapping

- （1）进程可以正常读 / 写被映射的前 5000 字节 (0~4999)，写操作的改动会在一定时间后反映在原文件中
- （2）对于 5000~8191 字节，进程可以进行读写过程，不会报错。但是内容在写入前均为 0，另外，写入后不会反映在文件中
- （3）对于 8192~14999 字节，进程不能对其进行读写，会报 SIGBUS 错误。
- （4）对于 15000 以外的字节，进程不能对其读写，会引发 SIGSE



SIGBUS 信号

- 产生原因：
  - 地址为未对齐，地址有效，但总线不能正常使用；
  - 访问 mmap region，超过了文件大小（上述情况）
- 默认行为：
  - Core dump

User-level memory mapping

- Example: fast file copy
  - Useful for applications like Web servers that need to quickly copy files
  - **`mmap`** allows file transfers without copying into kernel space
    - `VM->PM->file, page_fault_handler`

mmap() example: fast file copy

```
/*
 * mmapcopy - uses mmap to copy file fd to stdout
 */
void mmapcopy(int fd, int size)
{
  char *bufp;
  /* map the file to a new VM area */
  bufp = mmap(0, size, PROT_READ,
              MAP_PRIVATE, fd, 0);
  /* write the VM area to stdout */
  write(1, bufp, size);
  return ;
}
```

mmap() example: fast file copy

```
int main(int argc, char **argv)
{
  struct stat stat;
  /* check for required command line argument */
  if ( argc != 2 ) {
    printf(“usage: %s <filename>\n”, argv[0]);
    exit(0) ;
  }
  /* open the file and get its size*/
  fd = open(argv[1], O_RDONLY, 0);
  fstat(fd, &stat);
  mmapcopy(fd, stat.st_size);
}
```

对 mmap() 的理解

- Mmap() 的主要性能优势来自于减少 copies
  - vfs 的 read/write 系统调用中，文件内容的拷贝要多经历内核缓冲区这个阶段，所以比 mmap 多了一次内存拷贝
  - mmap 只有用户空间的内存拷贝 (这个阶段 read/write 也有)
  - mmap 也被称为 zero-copy 技术。

课堂练习

- 假设有一个输入文件 hello.txt，由字符串” Hello, world!\n” 组成，编写一个 C 程序，使用 mmap 将 hello.txt 的内容改为” Jello, world!\n” 。

```
1 #include "csapp.h"
2
3 /*
4  * mmapwrite - uses mmap to modify a disk file
5  */
6 void mmapwrite(int fd, int len)
7 {
8     char *bufp;
9
10    bufp = Mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
11    bufp[0] = 'J';
12 }
13
14 /* mmapwrite driver */
15 int main(int argc, char **argv)
16 {
17     int fd;
18     struct stat stat;
19
20     /* check for required command line argument */
21     if (argc != 2) {
22         printf("usage: %s <filename>\n", argv[0]);
23         exit(0);
24     }
25
26     /* open the input file and get its size */
27     fd = Open(argv[1], O_RDWR, 0);
28     fstat(fd, &stat);
29     mmapwrite(fd, stat.st_size);
30     exit(0);
31 }
```