Virtual Memory
Beyond Physical Memory

Liu yufeng
Fx_yfliu@163.com
Hunan University

- Thus far, we’ve assumed that an address space is unrealistically small and fits into physical memory.
- However, to support large address spaces, the OS will **need a place to stash away portions of address spaces** that currently aren’t in great demand.
- In general, such a location should have **more capacity than memory**. In modern systems, this role is usually served by a **hard disk drive**.

Swap Space

- The first thing we will need to do is to **reserve some space** on the disk for moving pages back and forth.
- In operating systems, we generally refer to such space as **swap space**.
- Thus, we will simply assume that the OS can read from and write to the swap space, **in page-sized units**.
- The OS will need to remember the **disk address** of a given page.

A large VM when only a small physical memory is available

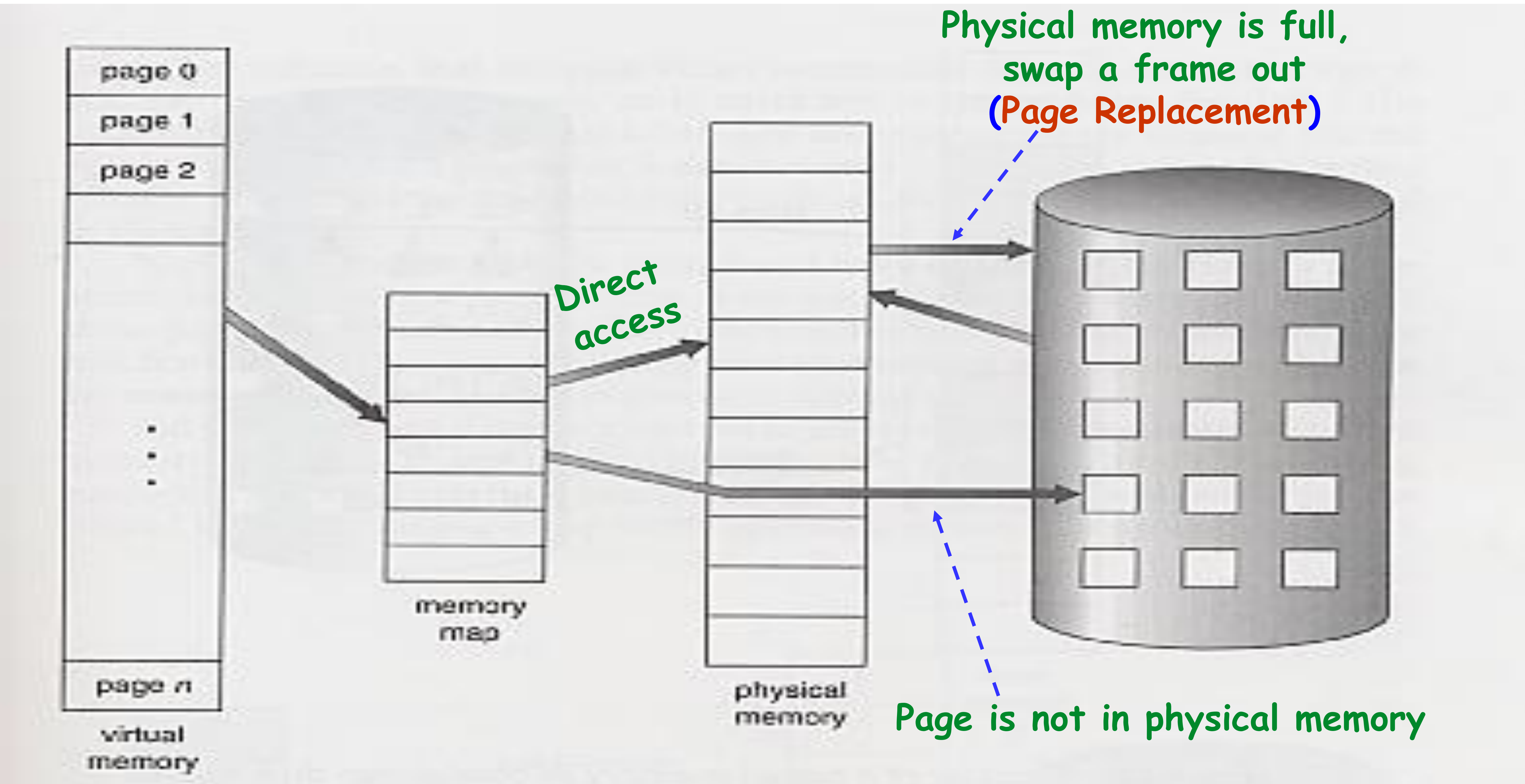
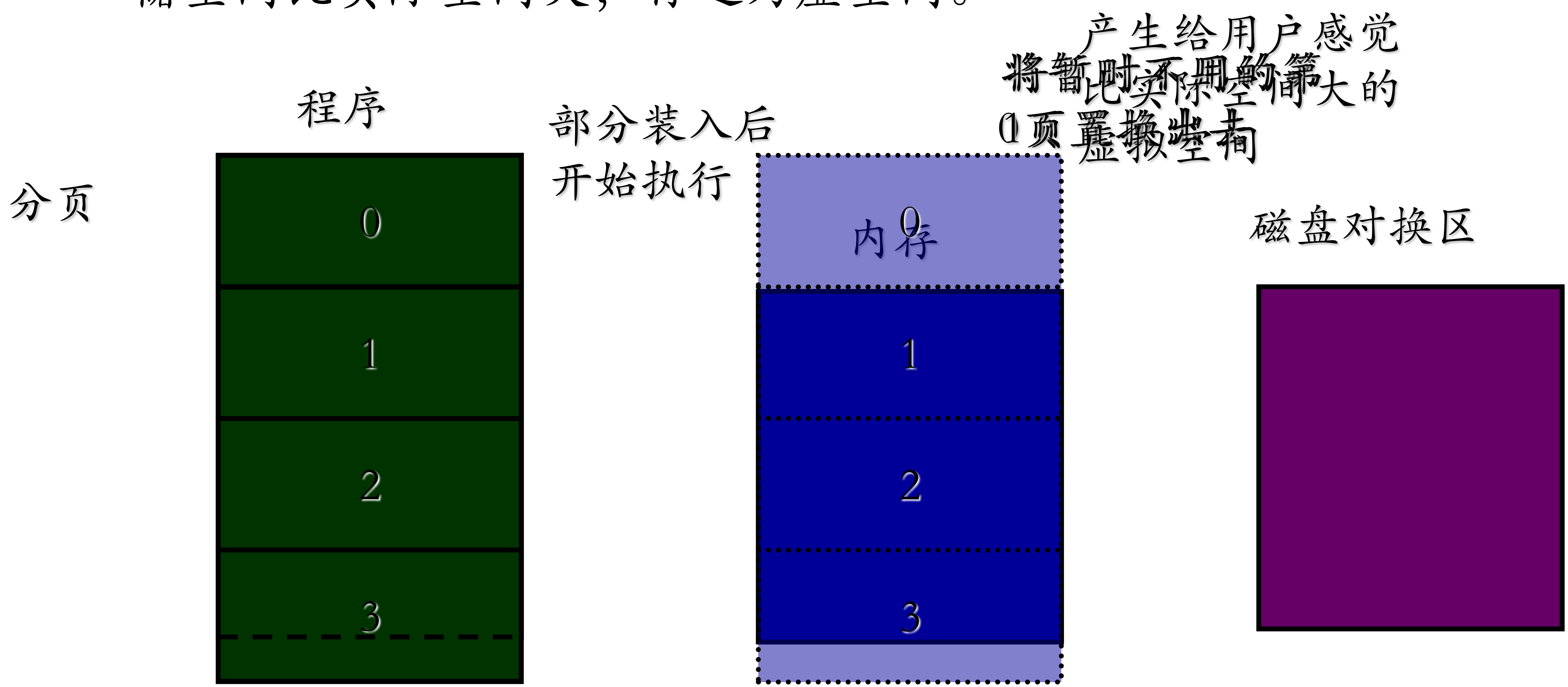


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

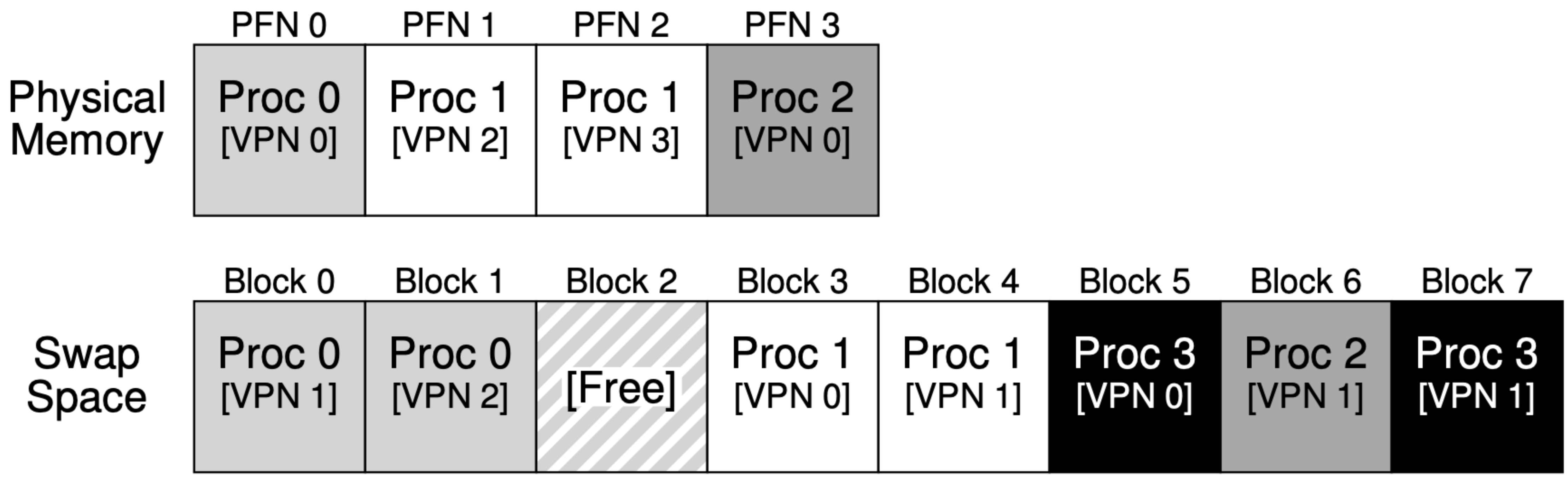
虚拟存储器的定义

- 虚拟存储器的定义
  - 虚拟存储指仅把作业的一部分装入内存便可运行的存储管理系统，通过作业各部分的动态调入和置换，用户所感觉的存储空间比实际空间大，称之为虚空间。



虚空间大小

- 虚空间大小
  - 虚空间的逻辑大小 ＝ 可寻址范围
  - 虚空间的实际大小 ＝ 内存＋外存对换区
- 例：32位操作系统的可寻址范围是2^{32}=4GByte，Windows98系列系统。
- 例：在window系统盘根目录下，有兑换文件——外存对换区。如XP系统的pagefile.sys文件
- 例：Linux中的swap交换分区



- 本例中，一个 包含4 页的物理内存和一个 8 页的交换空间。
- 3 个进程（进程 0、进程 1 和进程 2）主动共享物理内存。但 3 个中的每一个，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。
- 第 4 个进程（进程 3）的所有页都被交换到硬盘上。
- 有一块交换空间是空闲的。

Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk.
    - When the hardware looks in the PTE, it may find that the page is not present in physical memory.
    - The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

The Page Fault

- If a page is not present and has been swapped to disk, the OS will need to swap the page into memory.
- Thus, a question arises: **how will the OS know where to find the desired page**?
- In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as **the PFN of the page for a disk address**.
- When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

存在位与外存地址

- 存在位
- 对页表进行扩充
  - 让系统了解页面装入状态

页面如果不在内存中，记录页面在交换区的位置

页号	块号	存取控制	存在位	引用位	修改位	外存地址

  - 存在位：
    - » 为0　不在内存中
    - » 为1　在内存中

页面是否被访问过

页面是否被修改过

补充：交换术语及其他

对于不同的机器和操作系统，虚拟内存系统的术语可能会有点令人困惑和不同。例如，页错误（page fault）一般是指对页表引用时产生某种错误：这可能包括在这里讨论的错误类型，即页不存在的错误，但有时指的是内存非法访问。事实上，我们将这种完全合法的访问（页被映射到进程的虚拟地址空间，但此时不在物理内存中）称为“错误”是很奇怪的。实际上，它应该被称为“页未命中（page miss）”。但是通常，当人们说一个程序“页错误”时，意味着它正在访问的虚拟地址空间的一部分，被操作系统交换到了硬盘上。

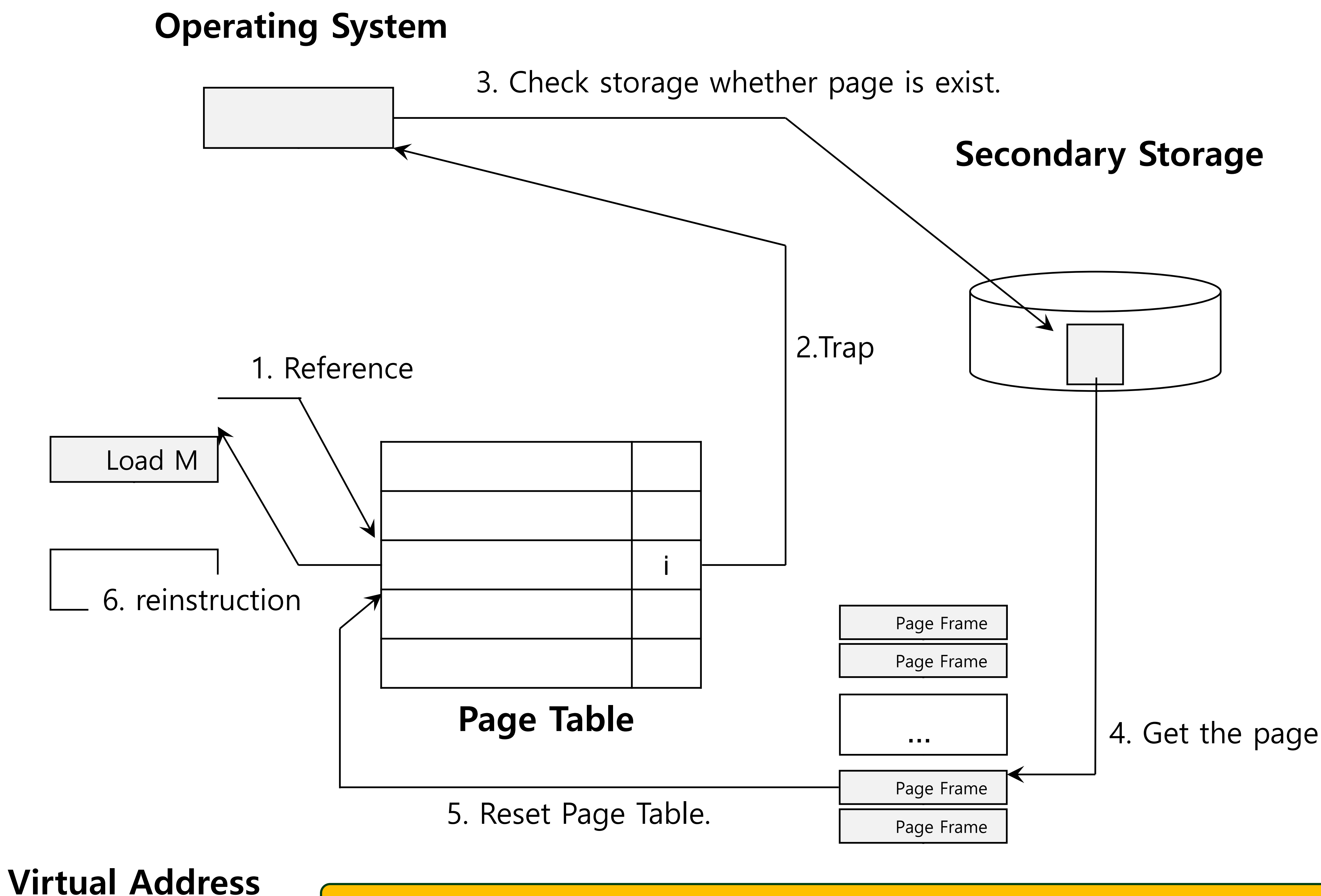
我们怀疑这种行为之所以被称为“错误”，是因为操作系统中的处理机制。当一些不寻常的事情发生的时候，即硬件不知道如何处理的时候，硬件只是简单地把控制权交给操作系统，希望操作系统能够解决。在这种情况下，进程想要访问的页不在内存中。硬件唯一能做的就是触发异常，操作系统从开始接管。由于这与进程执行非法操作处理流程一样，所以我们把这个活动称为“错误”，这也许并不奇怪。

The Page Fault

- Accessing page that is not in physical memory.
  - If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault.

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page Fault Control Flow – Hardware

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
else RaiseException(PROTECTION_FAULT)
```

Page Fault Control Flow – Hardware

```
else // TLB Miss
PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)    //还记得PTE的有效位吗？虚拟内存中的无效页，无需分配物理内存
        RaiseException(SEGMENTATION_FAULT)
else
if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else if (PTE.Present == True) //数据页在内存
// assuming hardware-managed TLB
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
else if (PTE.Present == False) // //数据页不在内存
        RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

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

- 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 未命中，然后再一次重试时，TLB 命中，此时硬件将能够访问所需 的值

What If Memory Is Full ?

- The OS like to page out pages to make room for the new pages the OS is about to bring in.
  - The process of picking a page to kick out, or replace is known as page-replacement policy

When Replacements Really Occur

- 操作系统不会等到内存已经完全满了以后才执行交换流程，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，HW）和低水位线（Low Watermark，LW），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）
- 通过同时执行多个交换过程，我们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集（cluster）或分组（group），同时写入到交换区间，从而提高硬盘的效率[LL82]。我们稍后在讨论硬盘时将会看到，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。

What If Memory Is Full?

- We **assumed there is plenty of free memory** in which to page in a page from swap space.
- Memory **may be full** (or close to it). Thus, the OS might like to first page out one or more pages to make room for the new page(s).
- The process of picking a page to kick out, or replace is known as the **page replacement policy**.

关键问题：如何决定踢出哪个页

操作系统如何决定从内存中踢出哪一页（或哪几页）？这个决定由系统的替换策略做出，替换策略通常会遵循一些通用的原则（下面将会讨论），但也会包括一些调整，以避免特殊情况下的行为。

Cache Management

- The main memory can be viewed as a **cache** for virtual memory pages in the system.
- A replacement policy for this cache is to **minimize** the number of **cache misses**.
- Alternately, one can view our goal as **maximizing** the number of **cache hits**.

- Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT 平均内存访问时间)** for a program.

$$AMAT = (Hit_{\%} \cdot T_M) + (Miss_{\%} \cdot T_D)$$

TM表示访问内存的成本

TD表示访问磁盘的成本

Hit表示在缓存中找到数据的 概率（命中）

Miss表示在缓存中找不到数据的概率（未命中）。

- Assuming the cost of accessing memory (T_M) is around **100 nanoseconds**, and the cost of accessing disk (T_D) is about **10 milliseconds**, and miss rate is **10%**, hit rate is **90%**.

- We have the following AMAT: 0.9 x 100ns + 0.1 x 10ms, which is 90ns + 1ms, or 1.00009 ms, or about 1 millisecond.

- If our hit rate had instead been **99.9%**, the result is quite different: AMAT is 10.1 µs, or **roughly 100 times faster**. As the hit rate approaches 100%, AMAT approaches 100 nanoseconds.

The Optimal Replacement Policy

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed furthest in the future
  - Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect（由于这是面向未来的算法，因此不可能真正实现，只能作为一个参照标准）

在引用最远将来会访问的页之前，你肯定会引用其他页！

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

前 3 个访问是未命中，因为缓存开始是空的。这种未命中有时也称作冷启动未命中（cold-start miss，或强制未命中，compulsory miss）。

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

- Unfortunately, **the future is not generally known**; you **can’t** build the optimal policy for a general-purpose operating system.

- The optimal policy will thus serve only **as a comparison** point, to know **how close we are to “perfect”**.

提示：与最优策略对比非常有用

虽然最优策略非常不切实际，但作为仿真或其他研究的比较者还是非常有用的。比如，单说你喜欢的新算法有 80%的命中率是没有意义的，但加上最优算法只有 82%的命中率（因此你的新方法非常接近最优），就会使得结果很有意义，并给出了它的上下文。因此，在你进行的任何研究中，知道最优策略可以方便进行对比，知道你的策略有多大的改进空间，也用于决定当策略已经非常接近最优策略时，停止做无谓的优化[AD03]。

Tracing the FIFIO Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

Even though page 0 had been accessed a number of times, FIFO still kicks it out.

补充：Belady 的异常

Belady（最优策略发明者）及其同事发现了一个有意思的引用序列[BNS69]。内存引用顺序是：1，2，3，4，1，2，5，1，2，3，4，5。他们正在研究的替换策略是 FIFO。有趣的问题：当缓存大小从 3 变成 4 时，缓存命中率如何变化?

一般来说，当缓存变大时，缓存命中率是会提高的（变好）。但在这个例子，采用 FIFO，命中率反而下降了！你可以自己计算一下缓存命中和未命中次数。这种奇怪的现象被称为 Belady 的异常（Belady's Anomaly）。

其他一些策略，比如 LRU，不会遇到这个问题。可以猜猜为什么？事实证明，LRU 具有所谓的栈特性（stack property）[M+70]。对于具有这个性质的算法，大小为 $N+1$ 的缓存自然包括大小为 N 的缓存的内容。因此，当增加缓存大小时，缓存命中率至少保证不变，有可能提高。先进先出（FIFO）和随机（Random）等显然没有栈特性，因此容易出现异常行为。

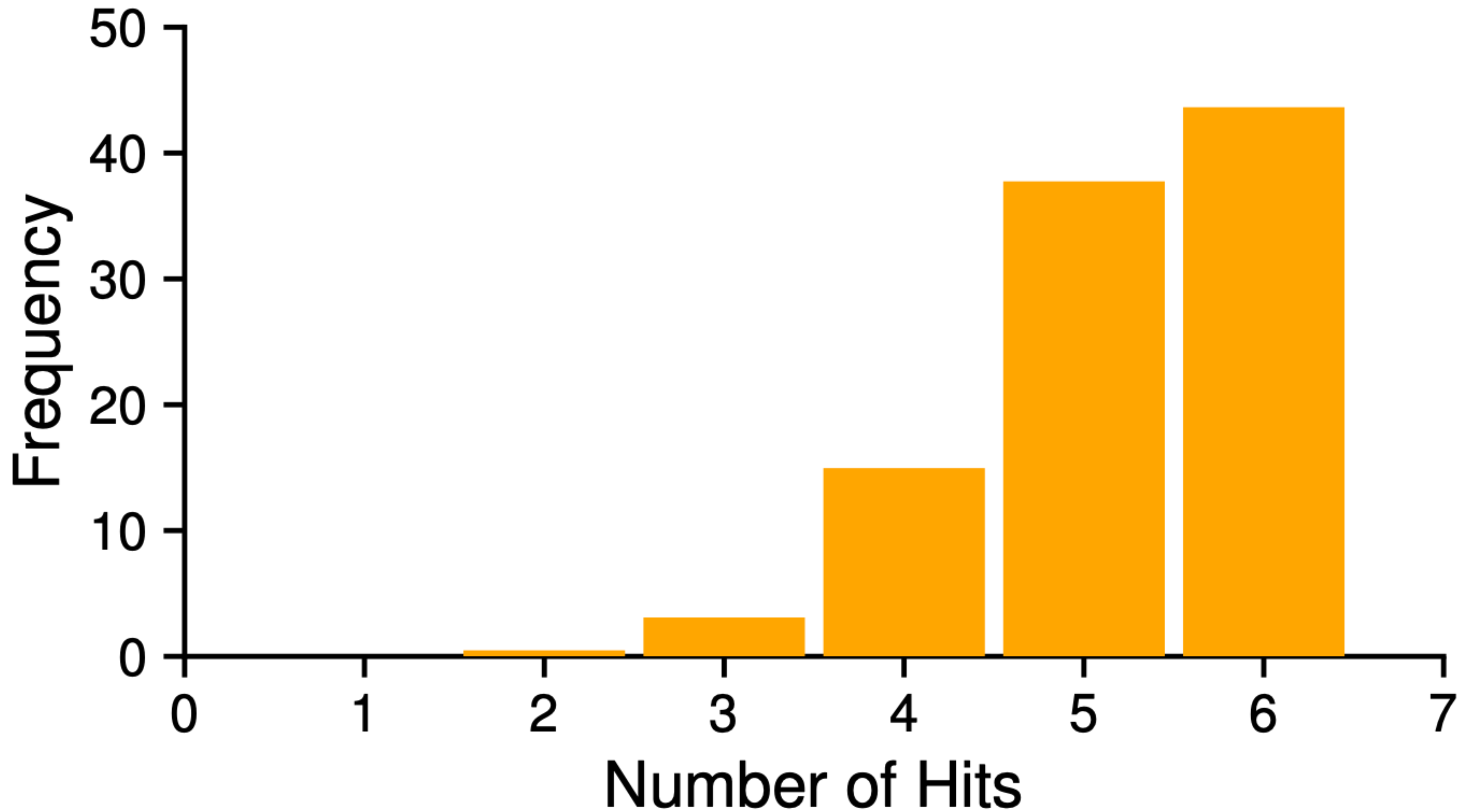
Another Simple Policy: Random

- Random has properties similar to FIFO; it is **simple to implement**, but it doesn’t really try to be too intelligent in picking which blocks to evict.

- How Random does depends entirely upon how lucky (or unlucky) Random gets in its choices.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Tracing the Random Policy



Random Performance over 10,000 Trials

- 当然，随机的表现完全取决于多幸运（或不幸）。在上面的例子中，随机比 FIFO 好一点， 比最优的差一点。事实上，我们可以运行数千 次的随机实验，求得一个平均的结果。
- 图 22.1 显示了 10000 次试验后随机策略的平均命中 率，每次试验都有不同的随机种子。正如你所 看到的，有些时候（仅仅 40%的概率），随机 和最优策略一样好，在上述例子中，命中内存的次数是 6 次。有时候情况会更糟糕，只有 2 次或更少。随机策略取决于当时的运气。

Using History: LRU

- Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it **might kick out an important page** that is about to be referenced again.

- As we did with **scheduling policy**, to improve our guess at the future, we once again lean on the past and use history as our guide.

- For example, if a program has accessed a page in the near past, it is likely to access it again in the near future.

- 页替换策略可以使用的一个历史信息是频率（frequency）。如果一个页被访问了很多次，也许它不应该被替换，因为它显然更有价值。
- 页更常用的属性是访问的近期性（recency），越近被访问过的页，也许再次访问的可能性也就越大。
- 这一系列的策略是基于人们所说的局部性原则（principle of locality）[D70]，基本上只是对程序及其行为的观察。这个原理简单地说就是程序倾向于频繁地访问某些代码（例如 循环）和数据结构（例如循环访问的数组）。因此，我们应该尝试用历史数据来确定哪些页 面更重要，并在需要踢出页时将这些页保存在内存中
- 一系列简单的基于历史的算法诞生了。“最不经常使用”（Least-Frequently-Used，LFU）策略会替换最不经常使用的页。同样，“最少最近使用”（Least-Recently-Used，LRU）策略替换最近最少使用的页面。.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		LRU→	0
1	Miss		LRU→	0, 1
2	Miss		LRU→	0, 1, 2
0	Hit		LRU→	1, 2, 0
1	Hit		LRU→	2, 0, 1
3	Miss	2	LRU→	0, 1, 3
0	Hit		LRU→	1, 3, 0
3	Hit		LRU→	1, 0, 3
1	Hit		LRU→	0, 3, 1
2	Miss	0	LRU→	3, 1, 2
1	Hit		LRU→	3, 2, 1

Tracing the LRU Policy

- 给某作业分配了三块主存（开始时为空），采用先进先出页面置换算法，该作业依次访问的页号为：1，2，3，4，1，2，5，1，2，3，4，5，6,将产生（　）次缺页中断。
- 如果采用LRU，将会产生（　）次缺页中断。
- 如果是最优置换，将会产生（　）次缺页中断。

Implementing Historical Algorithms

- To keep track of which pages have been least- and most-recently used, the system has to do some accounting work **on every memory reference**.

- One method that could help speed this up is to add a little bit of hardware support, such as a **time field** in memory.

- When replacing a page, the OS simply **scan all the time fields** in the system to find the least-recently-used page.

- Unfortunately, as the number of pages in a system grows, **scanning a huge array** of times just to find the absolute least-recently-used page is **prohibitively expensive**.

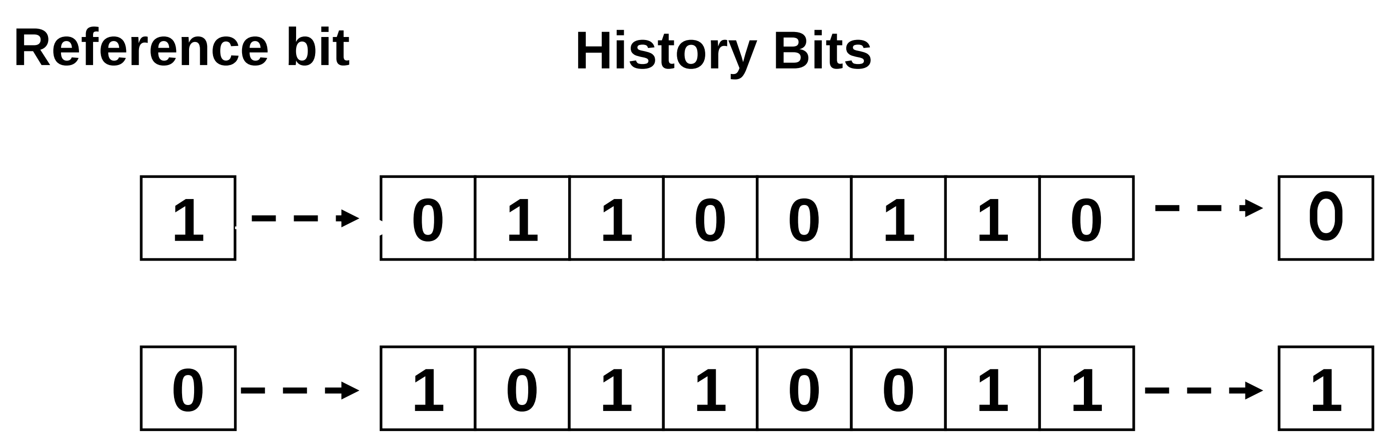
关键问题：如何实现 LRU 替换策略

由于实现完美的 LRU 代价非常昂贵，我们能否实现一个近似的 LRU 算法，并且依然能够获得预期的效果？

随着系统中页数量的增长，扫描所有页的时间字段只是为了找到最精确最少使用的页，这个代价太昂贵。想象一下一台拥有 4GB 内存的机器，内存切成 4KB 的页。这台机器有一百万页，即使以现代 CPU 速度找到 LRU 页也将需要很长时间。这就引出了一个问题：我们是否真的需要找到绝对最旧的页来替换？找到差不多最旧的页可以吗？

LRU Approximation Algorithms

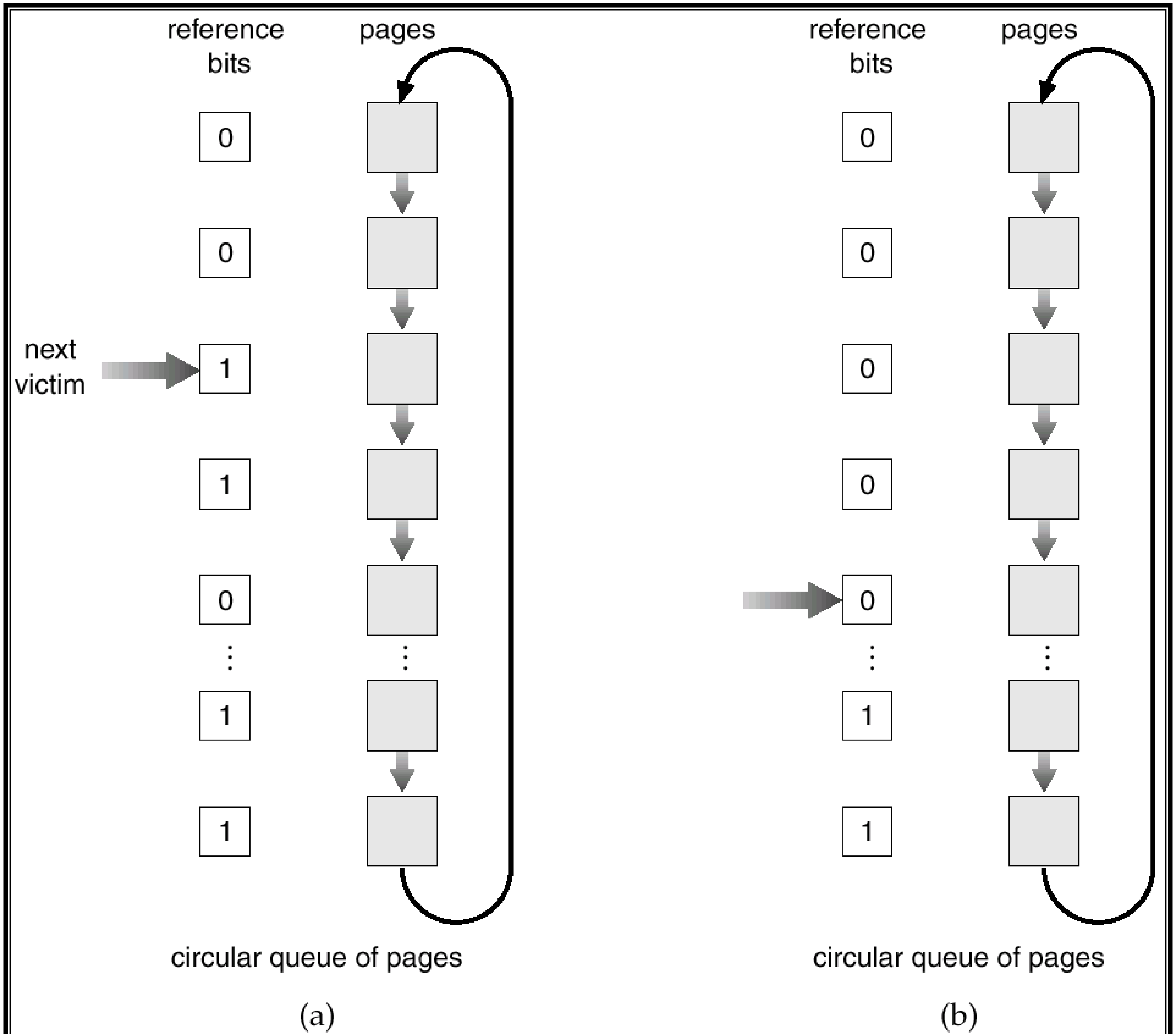
- Reference bit
  - With each page associate a bit, initially -= 0
  - When page is referenced bit set to 1 at （在每个时间间隔范围内，当页被访问时设置为1）.
  - The page with the smallest number is the LRU page;



Clock Algorithm

- 系统中的所有页都放在一个循环列表中。时 钟指针（clock hand）开始时指向某个特定的页（哪个页不重要）。
- 当必须进行页替换时，操 作系统检查当前指向的页 P 的使用位是 1 还是 0。如果是 1，则意味着页面 P 最近被使用，因此不适合被替换。然后，P 的使用位设置为 0，时钟指针递增到下一页（P + 1）。
- 该算法 一直持续到找到一个使用位为 0 的页，使用位为 0 意味着这个页最近没有被使用过（在最 坏的情况下，所有的页都已经被使用了，那么就将所有页的使用位都设置为 0）。

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-chance Algorithm

由访问位A和修改位M可以组合成下面四种类型的页面:

1类(A=0, M=0): 表示该页最近既未被访问， 又未被修改， 是最佳淘汰页。

2类(A=0, M=1)： 表示该页最近未被访问， 但已被修改， 并不是很好的淘汰页。

3类(A=1, M=0)： 最近已被访问， 但未被修改， 该页有可能再被访问。

4类(A=1, M=1): 最近已被访问且被修改， 该页可能再被访问。

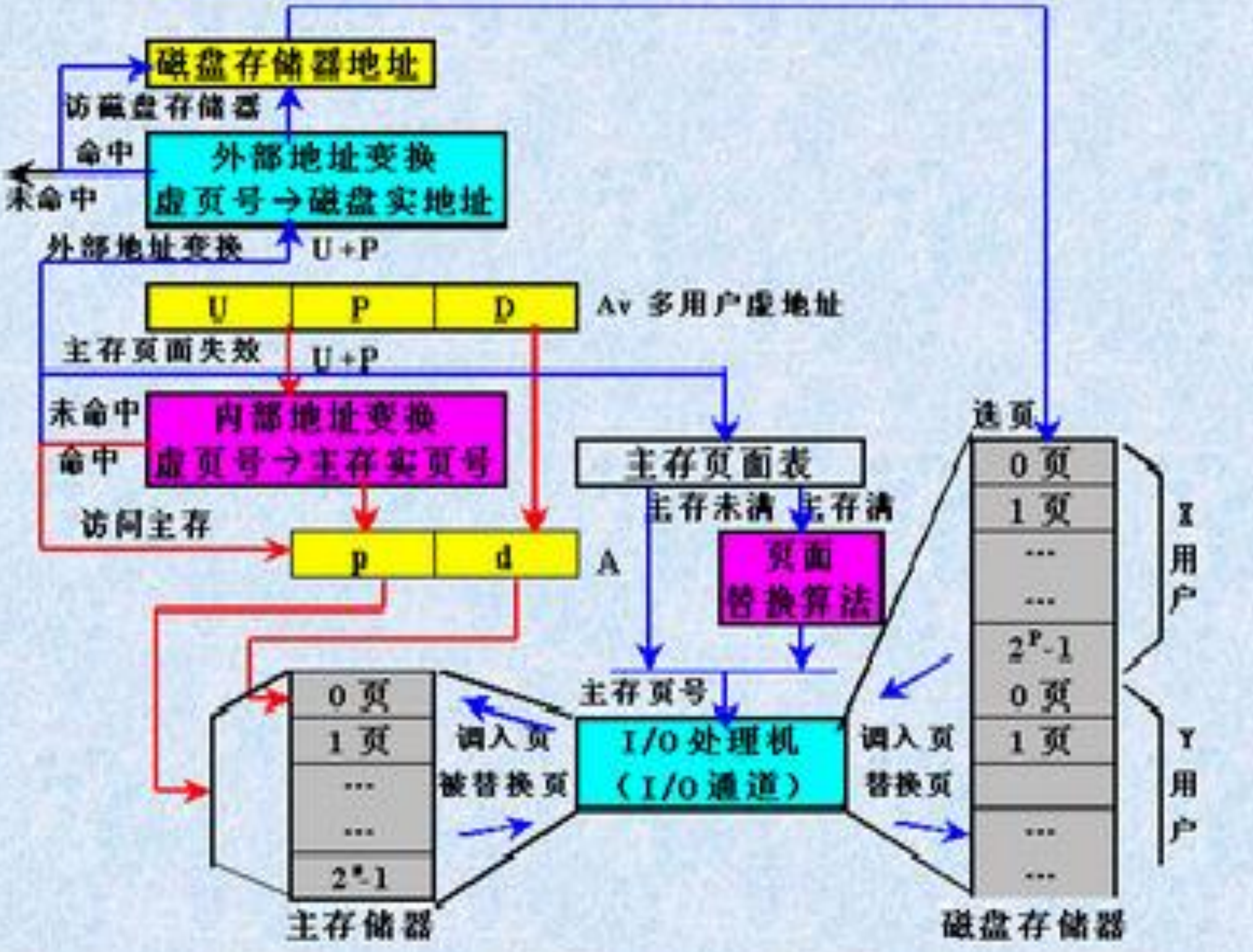
如果页已被修改（modified）并因此变脏（dirty），则踢出它就必须将它写回磁盘，这很昂贵。如果它没有被修改（因此是干净的，clean），踢出就没成 本。物理帧可以简单地重用于其他目的而无须额外的I/O。因此，一些虚拟机系统更倾向于踢出干净页，而不是脏页

Other VM Policies

- 除了按需请求调页。
- 操作系统可能会猜测一个页面即将被使用，从而提前载入。这种行为被称为预取（prefetching），只有在有合理的成功机会时才应该这样做。例如，一些系统将假设如果代码页 P 被载入内存，那么代码页 P + 1 很可能很 快被访问，因此也应该被载入内存.

Thrashing

- 当内存就是被超额请求时，操作系统应该做 什么，这组正在运行的进程的内存需求是否超出了可用物理内存？在这种情况下，系统将不断地进行换页，这种情况有时被称为抖动.
- 一些早期的操作系统有一组相当复杂的机制，以便在抖动发生时检测并由系统决定不运行部分进程，从而减少进程的工作集.
- 某些版 本的 Linux 会运行“内存不足的杀手程序（out-of-memory killer）”。这个守护进程选择一个 内存密集型进程并杀死它，从而以不怎么委婉的方式减少内存。.



磁盘存储器地址
访磁盘存储器
命中
外部地址变换
虚页号→磁盘实地址
未命中
外部地址变换
U+P
U
P
D
Av 多用户虚地址
主存页面失效
U+P
未命中
内部地址变换
虚页号→主存实页号
命中
访问主存
p
d
A
主存页面表
主存未满
主存满
页面
替换算法
选页
0 页
1 页
...
...
2^P-1
0 页
1 页
...
...
X
用
户
Y
用
户
主存页号
I/O 处理机
（I/O 通道）
调入页
被替换页
调入页
替换页
0 页
1 页
...
...
2^n-1
主存储器
磁盘存储器

End