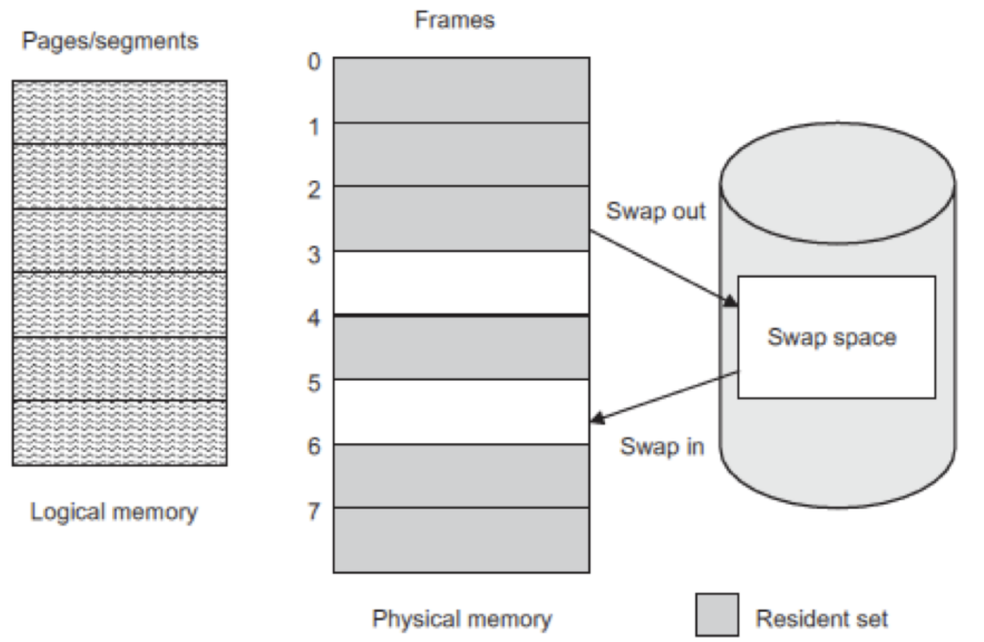
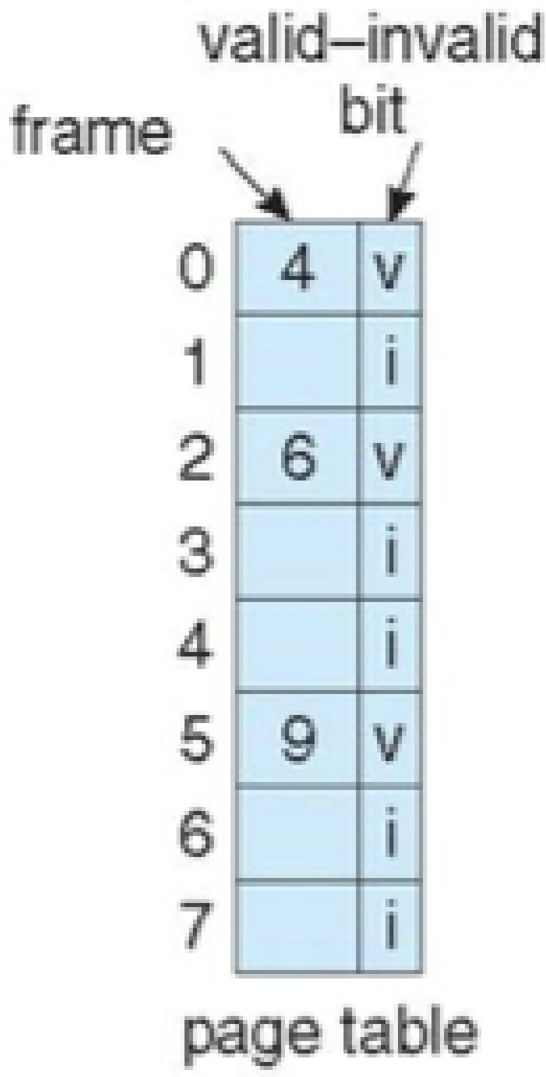
9 Virtual Memory

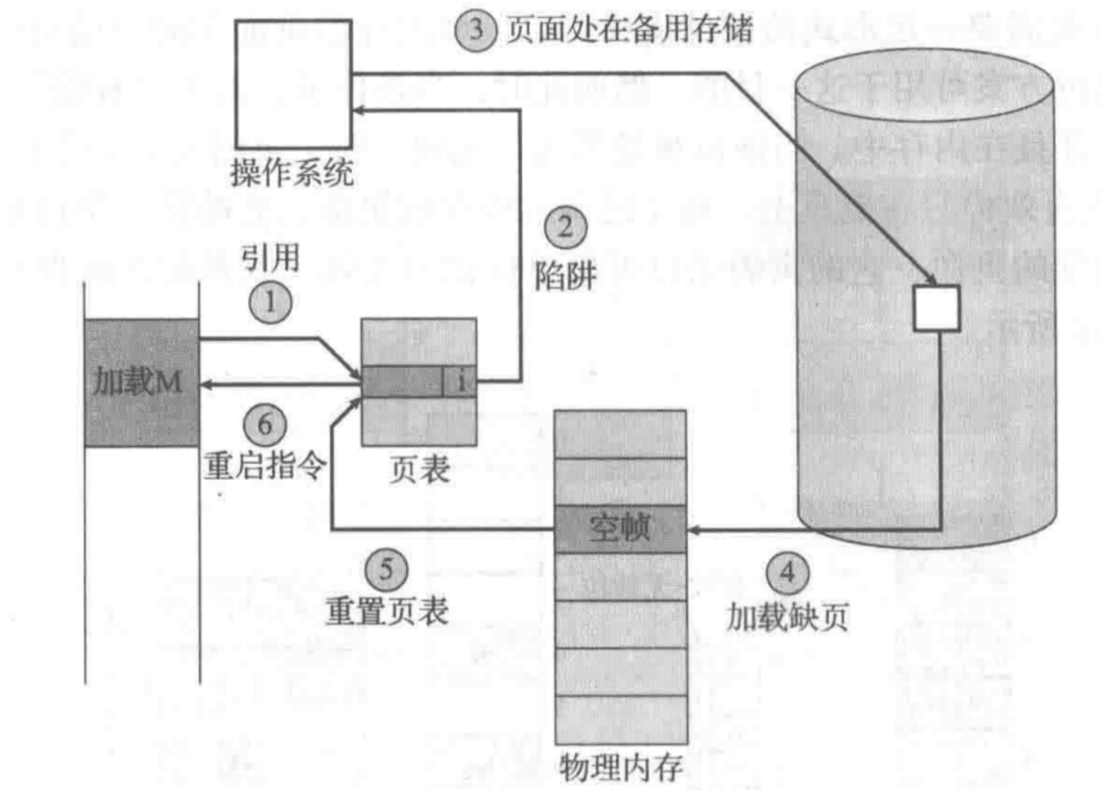
- Background
    - Virtual memory (VM) is a method that manages the exceeded sizeof larger processes as compared to the available space in the memory.
        - 虚拟内存(VM)是一种管理较大进程超出内存可用空间大小的方法。
    - Virtual memory - separation of user logical memory from physical memory.
        - 虚拟内存——用户逻辑内存与物理内存的分离。
        - ▪ Only part of the program needs to be in memory for execution.
        - ▪ The components of a process that are present in the memory are known as resident set of the process
        - ▪ Need to allow pages/segments to be swapped in and out.
            - 只有部分程序需要在内存中执行。
            - 存在于内存中的进程组件称为该进程的驻留集
            - ▪需要允许页面/段的交换进出。
    - The implementation of a VM system requires both hardware and software components.
        - • The software implementing the VM system is known as VM handler.
        - • The hardware support is the memory management unit built into the CPU.
        - 虚拟机系统的实现需要硬件和软件两个组件。
        - •实现虚拟机系统的软件称为虚拟机处理程序。
        - •硬件支持是内置于CPU中的内存管理单元mmu。
    - Swap space / Swap partition
        - Virtual memory targets the organization of the memory when the process size is too large to fit in the real memory.
            - 当进程大小太大而无法装入实际内存时，虚拟内存的目标是内存的组织。
        - The VM system realizes a huge memory only due to the hard disk.
            - 虚拟机系统只有通过硬盘才能实现巨大的内存。
        - With the help of the hard disk, the VM system is able to manage larger-size processes or multiple processes in the memory.
            - 在硬盘的帮助下，虚拟机系统可以管理内存中较大的进程或多个进程。
        - For this purpose, a separate space known as swap space isreserved in the disk.
            - 为此，在磁盘中保留一个称为交换空间的单独空间。
        - Swap space requires a lot of management so that the VM system woks smoothly.
            - 交换空间需要大量的管理，这样VM系统才能正常工作。
        - What if there isn’t enough physical memory available to load a program?如果没有足够的物理内存来加载程序怎么办?

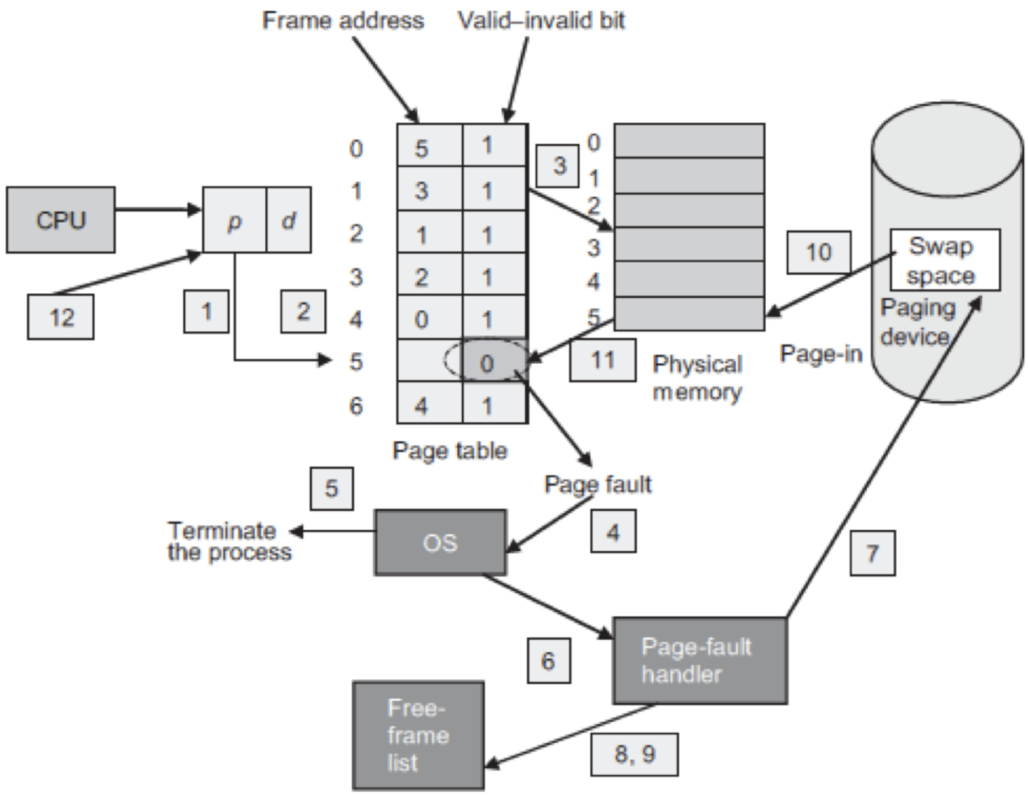
        - – A large portion of program’s code may be unused
            - -程序的大部分代码可能是未使用的,即使使用也可能是一段
        - – A large portion of program’s code may be used infrequently
            - -程序的大部分代码可能不经常被使用
        - • Idea: load a component of a process only if you need it
            - •理念:只在需要时才加载进程的组件
        - 

    - 好处
        - 运行更多程序,增加cpu利用率吞吐量throughput,无响应和周转时间
        - io变少,运行速度更快
- Demand Paging
    - The concept of loading only a part of the program (page) into memory for processing.将程序(页)的一部分装入内存以供处理的概念。
    - when the process begins to run, its pages are brought into memory only as they are needed, and if they’re never needed, they’re never loaded.
        - 当进程开始运行时，它的页面只在需要时才被放入内存，如果它们永远不需要，则永远不会加载它们。
    - - when a logical address generated by a process points to a page that is not in memory.
        - -当进程生成的逻辑地址指向不在内存中的页面时。
        - Lazy swapper – never swapsa page into memory unlesspage will be needed
            - ❖ Swapper that deals with pages is a pager
            - 惰性交换器——永远不会将一个页面交换到内存中，除非需要这个页面
            - 处理页的交换器是寻呼机

- No page fault: the effective access time = the memory access time无页面故障:有效访问时间=内存访问时间
- How to recognize whether a page is present in the memory?识别内存中是否存在页面
  - 为了区分内存的页面和磁盘的页面,设置有效无效,在内存中,设置无效页面无效或者只在磁盘中,不在内存的页面,他的页表条目标记为无效
  - set to “**valid**” → the associated page **is both legal and in memory.关联的页面既合法又在内存中**
  - set to “**invalid**” → the page either is **not valid** (not in the logical address space of the process) **or is valid but is currently on the disk**.页面无效(不在进程的逻辑地址空间中)或有效但当前在磁盘上。
  - 

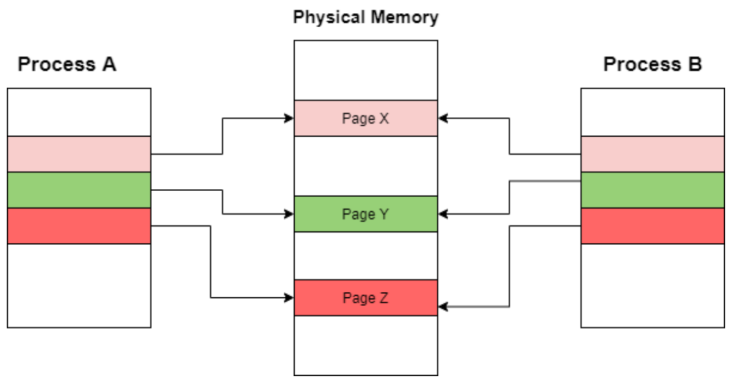
- **What happens if the process tries to access a page that is not in the memory?**
  - while translating the address through the page table notices that the page-table entry has an invalid bit. **It causes a trap to the OS** so that a page fault can be noticed.在通过页表转换地址时，会注意到页表项有一个无效的位。它会给操作系统产生一个陷阱，以便发现页面错误
  - when the page referenced is not present in the memory → **page fault**.

- 

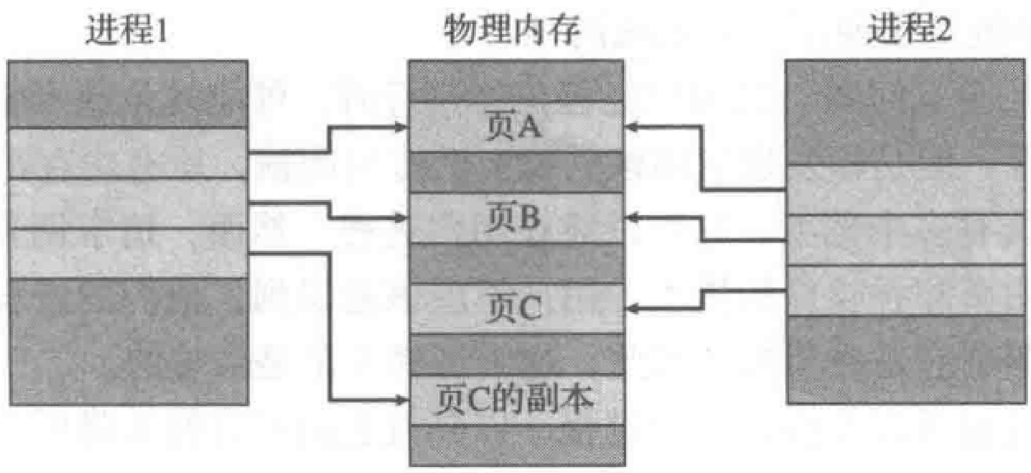
  

- What happens if there is no free frame?
  - the existing page in the memory needs to be paged-out. page-replacement algorithms需要将内存中的现有页面换出。页面置换算法
- *p* be **the probability of a page fault缺页的概率**
  - ▪ if $p = 0$, no page faults

        - ▪ if $p = 1$, every reference is a fault
    - **Effective Access Time (EAT)**
        - **EAT = (1 – p) x Memory Access Time + p x Page Fault Time**
        - 不缺页的概率x内存访问时间+缺页的概率x页面故障的时间
    - Major components of the page-fault (service) time:内存访问时间+ p x页面故障时间
        - 缺页进行以下一组操作
        - 1. Service the page-fault interrupt.
        - 2. Read in the page.
        - 3. Restart the process.
    - VM system also uses TLB to reduce the memory accesses and increase - Translation Lookaside Buffer
    - the system performance
- Copy-on-Write in Operating System
    - 允许父子进程最初共享相同的页面工作,创建副本
    - **Copy-on-Write** = strategy that those pages that are never written need not be copied. Only the pages that are written need be copied.那些从未写过的页面不需要复制。只有写好的那几页需要复制
    - The parent and child process to **share the same pages of the memory** initially父进程和子进程最初共享相同的内存页
    - **Process creation** using the **fork() system call** may (initially) **bypass the need for demand paging** by using a technique similar to *page sharing使用fork()系统调用创建进程可以(最初)通过使用类似于页面共享的技术绕过对需求分页的需求*
    - If any process either parent or child modifies the shared page, only then the page is copied如果任何进程，无论是父进程还是子进程，都修改了共享的页面，只有在这个时候，页面才会被复制。
        - 只有当任何一个进程，包括父进程或子进程，修改了共享页面时，该页面才会被复制。也就是说，只要父进程和子进程共享同一页面，只要没有进行修改，它们都会继续共享这个页面。只有在其中一个进程试图修改共享页面时，才会发生复制，以确保每个进程都有它自己的页面副本
    - 仅复制任意进程修改的页面,所有为修改的页面都可以由父子进程共享,只有可以修改的页面才需要标记为cow,
    -



Process A creates a new process - Process B

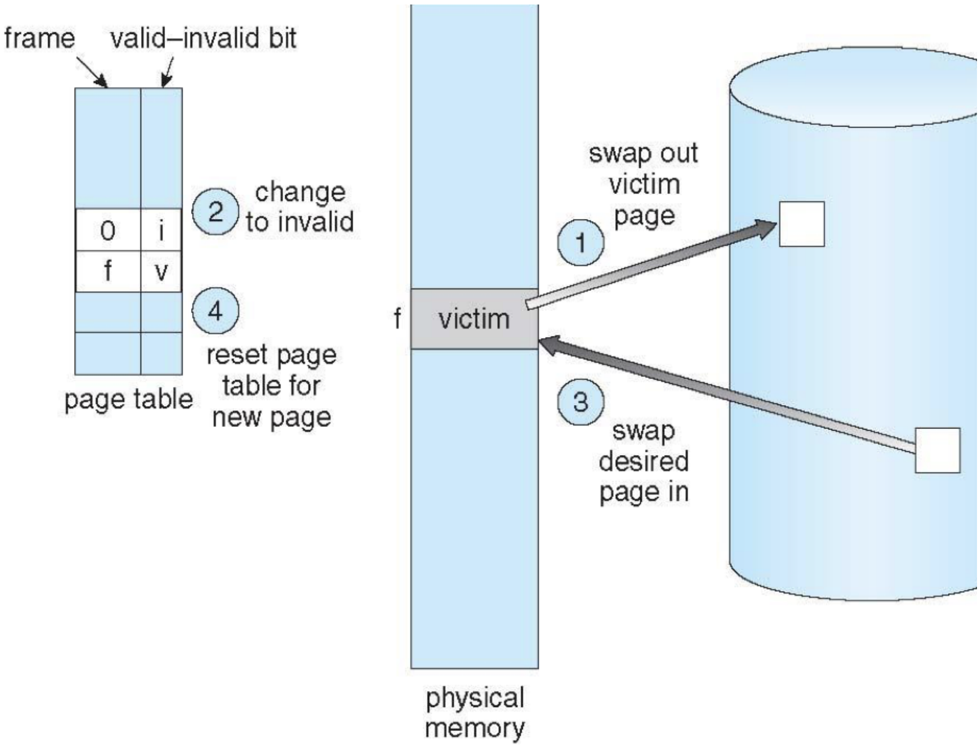


- Page Replacement
  - **What happens if there is no free frame? - Pagereplacement**
  - **How many frames shall be allocated to each process? -Frame allocation**
  - The degree of multiprogramming increases □ over-allocatingmemory □ NO free frames on the free-frame list, all memory is in use.
    - a low page fault rate低页面错误率
      - ➢ ensure that heavily used pages stay in memory确保频繁使用的页面留在内存中
      - ➢ the replaced page should not be needed for some time被替换的页面在一段时间内不需要，页面故障延迟时间低
    - a low latency of a page fault页面故障的低延迟
      - ➢ efficient code高效代码
      - ➢ replace pages that do not need to be written out替换不需要写入的页面
      - ➢ a special bit called the modify (dirty) bit can be associated with each page.每个页面都可以关联一个特殊的位，称为修改(脏)位。
    - Basic Page Replacement
      - 1. Find the location of the desired page on disk找到所需页面在磁盘上的位置
      - 2. Find a free frame:找一个自由的框架:
        - - If there is a free frame → use it

- - If there is no free frame → use a page replacement algorithm to select a victim frame-使用页面替换算法选择受害帧
- Check the modify (dirty) bit with each page or frame.
  - If the bit is set → the page has been modified.则表示页面修改成功
  - If the bit is not set → the page has not been modified. It need not be paged-out for replacement and can be overwritten by another page because its copy is already on the disk. This mechanism reduces the page-fault service time.
    - 如果没有设置位，则页面未被修改。由于它的副本已经在磁盘上，因此不需要将其换出以进行替换，并且可以被另一个页面覆盖。这种机制减少了页面故障服务时间。页面可以直接覆盖,不需要写会磁盘
-

  1. 找到所需页面的磁盘位置。
  2. 找到一个空闲帧：
     a. 如果有空闲帧，那么就使用它。
     b. 如果没有空闲帧，那么就使用页面置换算法来选择一个**牺牲帧**（victim frame）。
     c. 将牺牲帧的内容写到磁盘上，修改对应的页表和帧表。
  3. 将所需页面读入（新的）空闲帧，修改页表和帧表。
  4. 从发生缺页错误位置，继续用户进程。

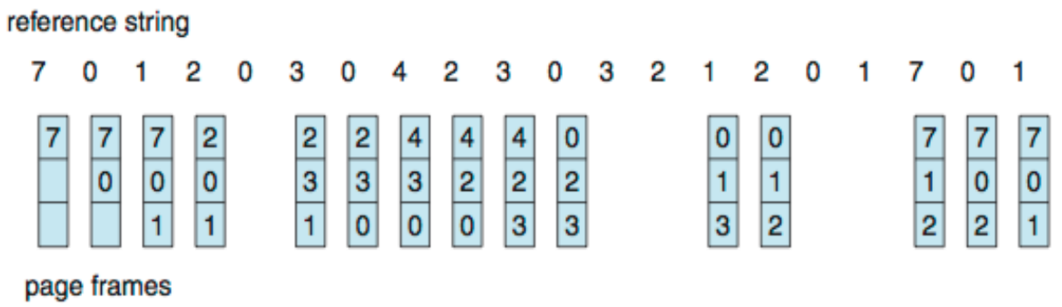
  请注意，如果没有空闲帧，那么需要两个页面传输（一个调出，一个调入）。这种情况实际上加倍了缺页错误处理时间，并相应地增加了有效访问时间。

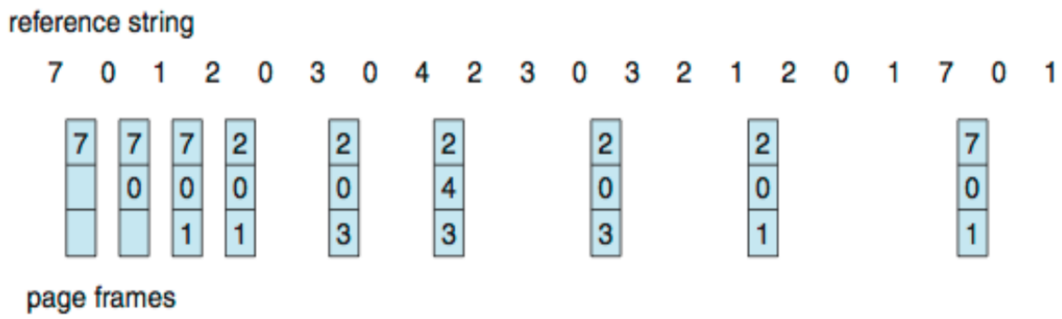
- 3. Bring the desired page into the (newly) free frame; update the page and frame tables.将所需的页面放入(新)空闲帧;更新页面和框架表
- 4. Continue the process by restarting the instruction that caused the trap通过重新启动引起陷阱的指令继续该过程

- Page-replacement algorithms

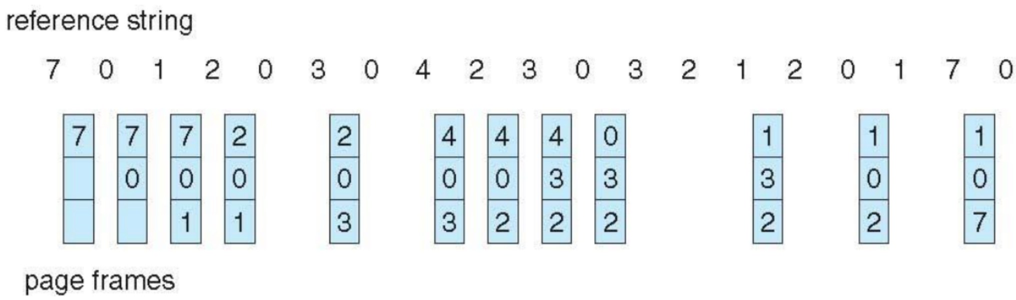
- First-In First-Out (FIFO) Algorithm
    -
        - When a page must be replaced, **the oldest page is chosen**.
        - Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
        - **3 frames** (3 pages can be in memory at a time per process)

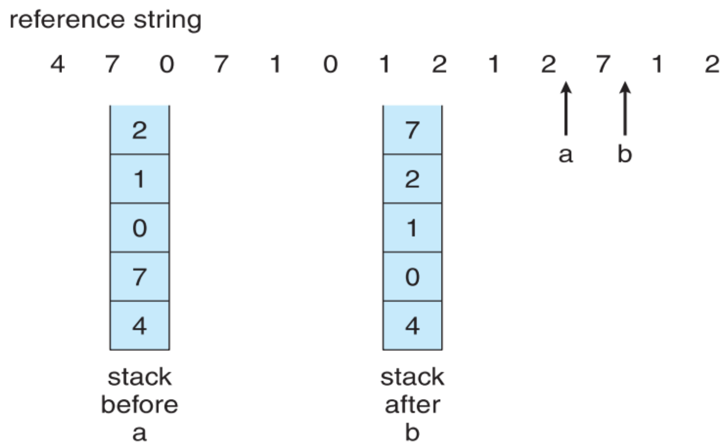
        - **15 page faults**
- Optimal Algorithm(OPT)
    - Replace page that will not be used for longest period of time替换长时间不使用的页面
        - Replace page that **will not be used for longest period of time**
            - **9 page fault**

    - Optimal algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
    - It cannot be implemented - there is no provision in the OS to know the future memory references.
    - The idea is to predict future references based on the past data,
    - 优化算法保证了在固定帧数下页面错误率最低。
    - 它无法实现——操作系统中没有知道未来内存引用的规定。
    - 这个想法是基于过去的数据来预测未来的参考资料
- Least Recently Used (LRU) Algorithm
    -



- **12 faults** – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

- when a page fault occurs, throw out the page that has been unused for the longest time当页面出现故障时，丢弃长时间未使用的页面
- Two implementations are feasible:
  - \- **Counter** implementation
    - associate with each page-table entry a time-of-use field or a counter; every time page is referenced through this entry, copy the clock into the counter
      - 与每个页表条目关联一个使用时间字段或计数器;每次通过此条目引用页面时，将时钟复制到计数器中
      - ➢ When a page needs to be changed, look at the counters to find the smallest value
      - ➢ replace the page with the smallest time value
        - 当需要更改页面时，请查看计数器以查找最小值
        - 更换时间值最小的页面
  - \- **Stack** implementation
    - whenever a page is referenced, it is removed from the stack and put on the top.每当一个页面被引用时，它就会从堆栈中移除并放在堆栈顶部。
    - the most recently used page is always at the top of the stack andthe least recently used page is always at the bottom最近使用的页面始终位于堆栈顶部最近最少使用的页面总是在底部
    - 

- Reference bit - will say whether the page has been referred in the last clock cycle or not.引用位——表示页面在上一个时钟周期中是否被引用
- The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in thepage).页面的引用位是由硬件在页面被引用时设置的(对页面中任意字节的读或写)。
- LRU Approximation Algorithms
    - LRU needs special hardware and still slow
        - LRU需要特殊的硬件并且仍然很慢
    - Reference bit - will say whether the page has been referred in the last clock cycle or not.
        - 参考位-表示该页是否在最后一个时钟周期中被引用。
    - The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
        - 页面的引用位由硬件在引用该页时设置(对该页中的任何字节进行读或写操作)。
    - Reference bits are associated with each entry in the page table.
        - With each page associate a bit, initially = 0
        - When page is referenced, bit set to 1
            - 引用位与页表中的每个条目相关联。
            - 与每页关联一个位，初始= 0
            - 当page被引用时，位设置为1

- Second-Chance (Clock) Algorithm
    - 如果是1,变0跳到下一个

- keeps a circular list of pages in memory, with the "**hand**" (iterator) pointing to the last examined page frame in the list.

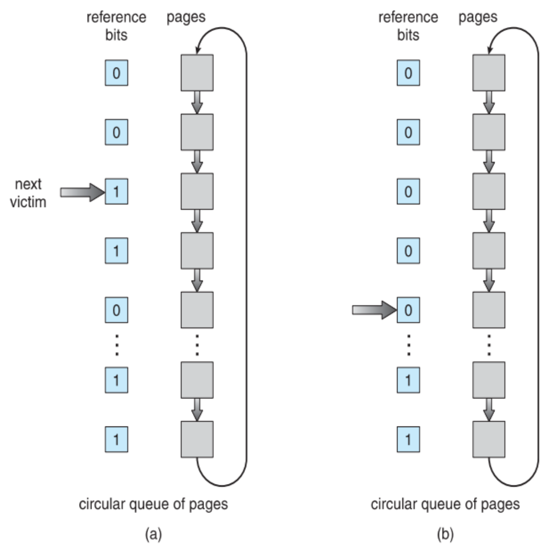
RB = Reference bit or **Use bit**
give information regarding whether the page has been used

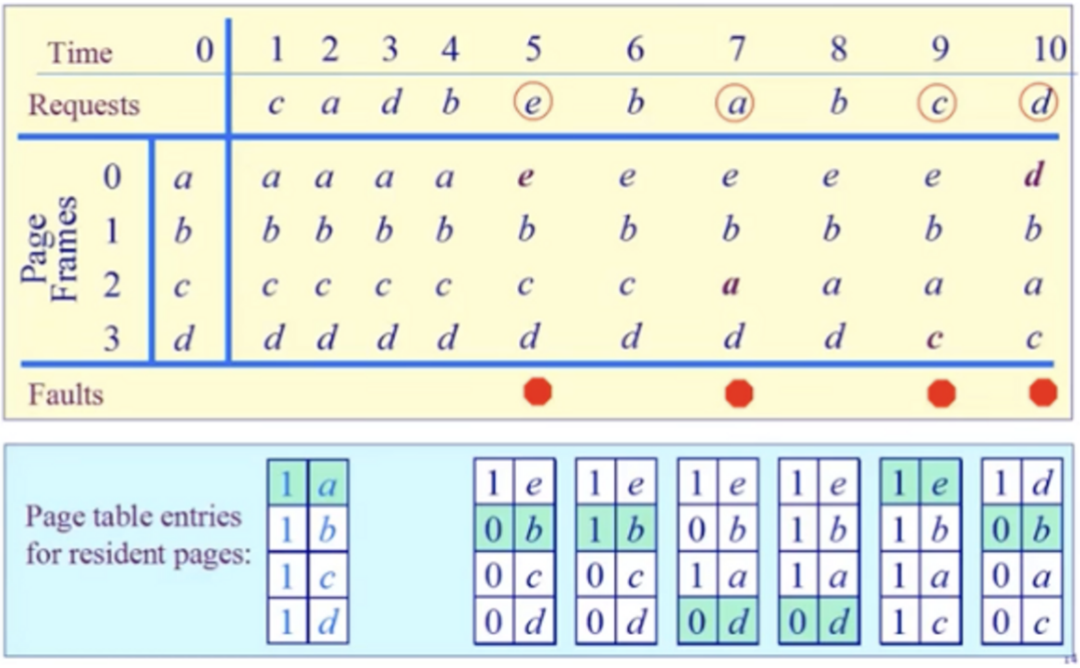
Iterator Scan:

- IF page's RB = 1, set to 0 & skip
- ELSE if RB = 0, remove

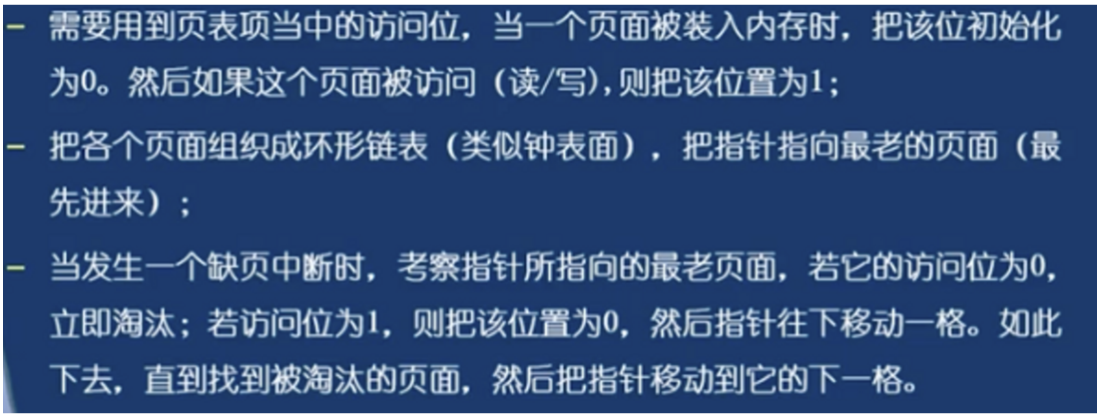
The idea behind:

A page that is being frequently used will not be replaced (RB=1).





Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	e	e	e	e	e	d
1	b	b	b	b	b	b	b	b	b	b	b
2	c	c	c	c	c	c	c	a	a	a	a
3	d	d	d	d	d	d	d	d	d	c	c
Faults						●		●		●	●

- d b a c
- 1 0 0 0
- /
- 

- 先走一圈,都变成0,第二次访问a,碰到0,切换为e,指针指向b,访问了,再置为1

- Counting Algorithms
  - Keep a counter of the number of references that havebeen made to each page.
    - 记录每页被引用的次数。
  - Least Frequently Used (LFU) Algorithm replaces page with smallest count.
    - 最少频繁使用(LFU)算法用最小计数替换页面。
  - Most Frequently Used (MFU) Algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used
    - 最常用(MFU)算法基于这样的论点:计数最小的页面可能刚刚引入，尚未被使用
- Frame Allocation

- **Equal allocation** - In a system with *x* frames and *y* processes,each process gets equal number of frames在有x帧和y进程的系统中
  - *Example*, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames每个进程得到相同数目的帧
  - 例如，如果有100帧(在为操作系统分配帧后)5个过程，每个过程20帧
- Proportional allocation – Frames are allocated to each process according to the process size.
  - Dynamic as degree of multiprogramming, process sizes change随着多路编程程度的动态变化，进程大小也在变化
    - $$s_i = \text{size of process } p_i$$
      $$S = \sum s_i$$
      $$m = \text{total number of frames}$$
      $$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$
    - **Proportional allocation - *example***

      A system with 62 frames

      - P1 = 10KB
      - P2 = 127KB

      Then

      - for P1 will be allocated (10 / 137) * 62 = 4 frames
      - P2 will get (127 / 137) * 62 = 57 frames.
- Thrashing
  - 如果分配给一个进程的物理页面太少，不能包含整个的工作集，即常驻集二 工作集，那么进程将会造成很多的缺页中断，需要频繁地在内存与外存之间替换页面，从而使进程的运行速度变得很慢，我们把这种状态称为“抖动”。
  - 产生抖动的原因：随着驻留内存的进程数目增加，分配给每个进程的物理页面数不断减小，缺页率不断上升。所以OS要选择一个适当的进程数目和进程需要的帧数，以便在并发水平和缺页率之间达到一个平衡。

- If a process does not have “enough”pages, the page-fault rate is very high.一个进程没有“足够”的页面，那么页面错误率就会非常高
- This leads to:
  - ☐Low CPU utilization
  - ☐Operating system thinks that it needs to increase the
  - degree of multiprogramming
  - ☐Another process added to the system
  - CPU利用率低
  - ☐操作系统认为它需要增加
  - 多路编程度
  - ☐系统中添加的另一个进程
- **Thrashing** =a process is busy swapping pages in and out
  - 进程忙于交换页面
    - ➢ a process spends more time paging then executing
    - 进程分页时间大于执行时间