Chapter 10
Virtual Memory

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocation Kernel Memory
- Other Considerations
- Operating System Examples

Background

- So far, a program need to be entirely loaded into the memory before it can execute
  - Limit the number of processes in the memory
  - Process memory can not be lager than physical memory
- However, part of a program may be rarely used
  - Error code, unusual routines, large data structures
- So, the entire program does not needed at the same time

Background

- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running ➔ more programs run at the same time
    - improve performance
  - Less I/O needed to load/swap programs into memory ➔ faster program loading, shorter program startup time

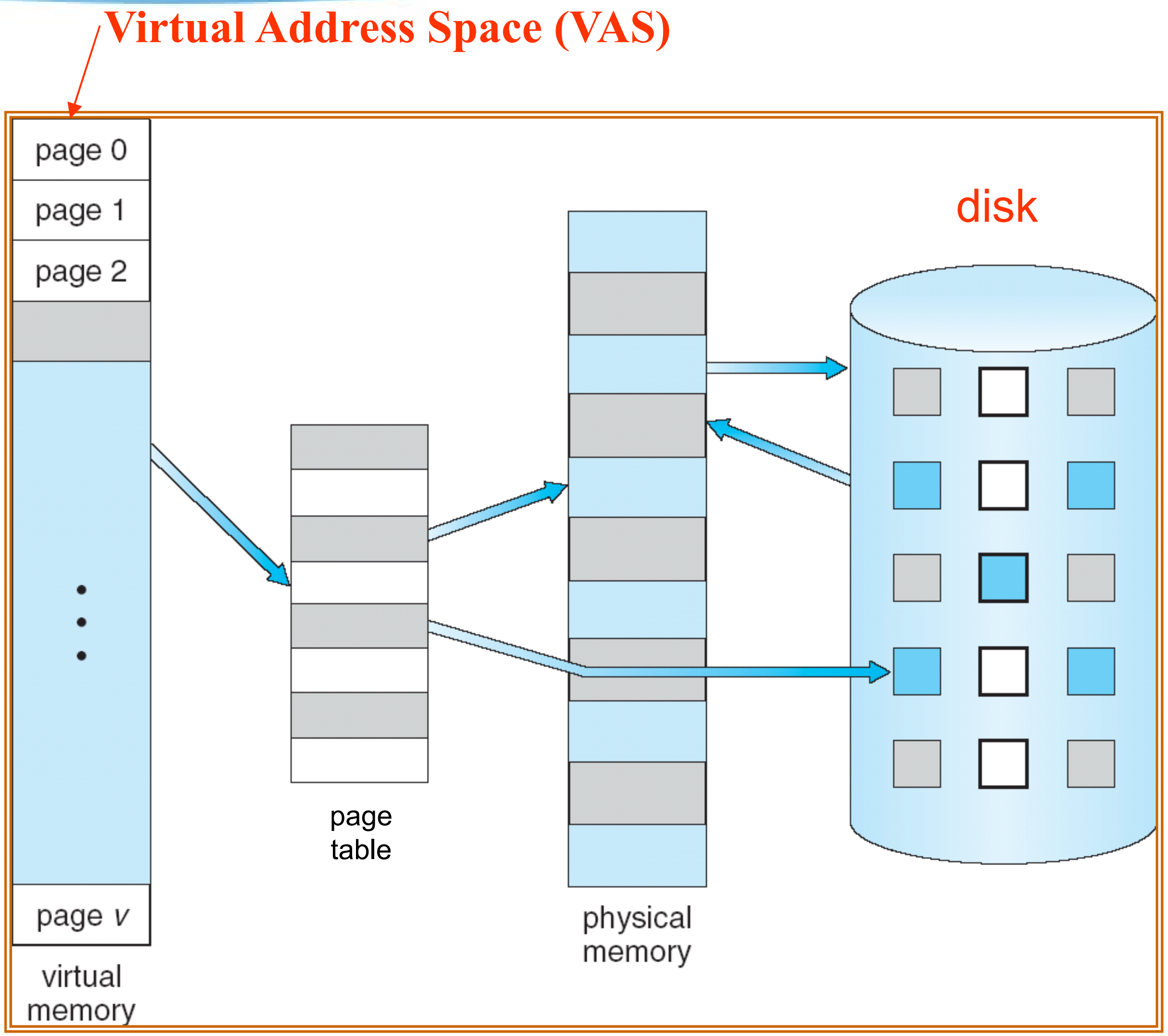
Background

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
    - As shown in Chapter 9
  - Allows for more efficient process creation
    - Less IO for program loading
  - Allows more processes in the memory
  - Makes the task of programming much easier
    - Programmer no longer needs to worry about the amount of physical memory available

Background

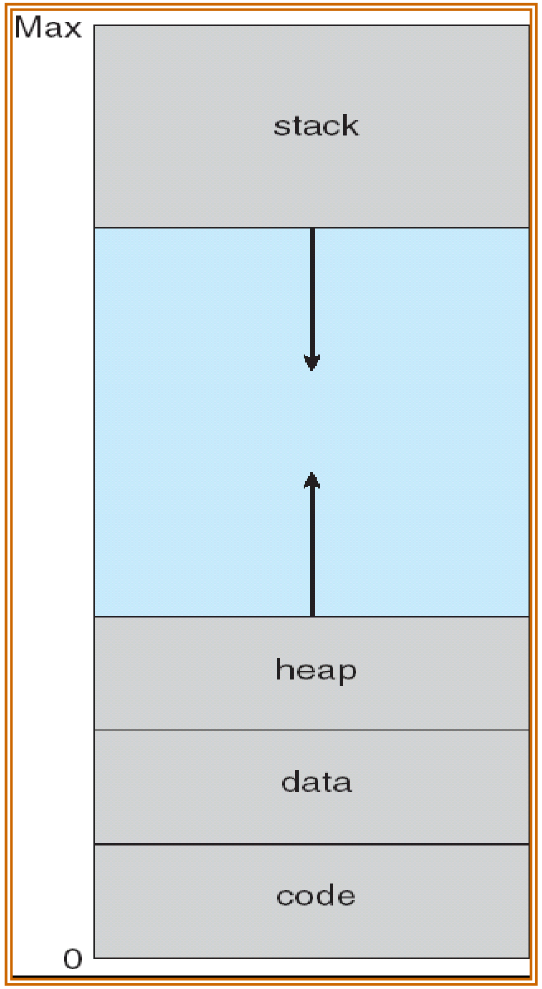
- Virtual memory can be implemented via
  - Demand paging (introduced later)
  - Demand segmentation (introduced later)

Virtual Memory > Physical Memory



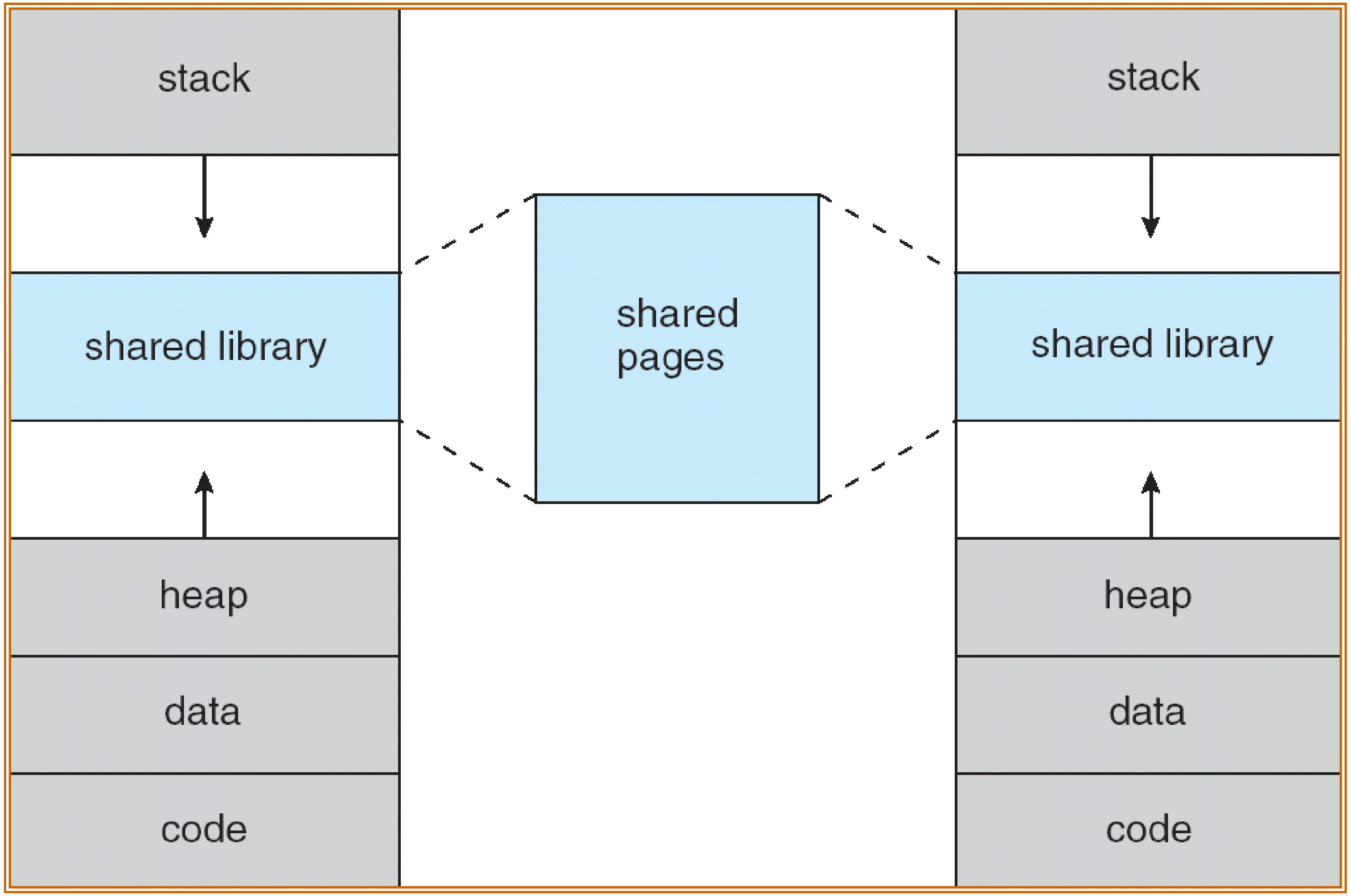
Virtual Address Space (VAS)

VAS is usually sparse…



- Usually, stack starts at max logical address and grow “down” while heap grows “up”
  - Unused address space between the two is hole
    - **No physical memory needed** for the hole
  - The hole is left for heap/stack growth, dynamically linked shared libraries, and etc.
- System libraries shared via mapping into virtual address space *(see next slide)*
- Shared memory by mapping pages read-write into virtual address space

Shared Library Using Virtual Memory

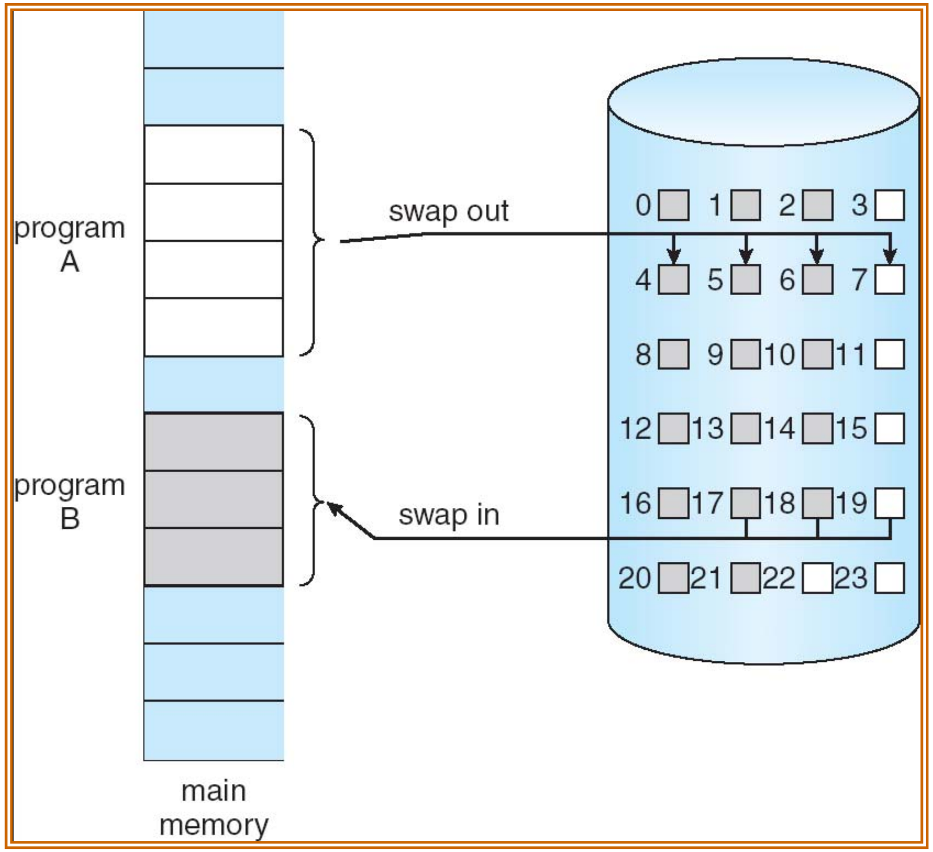


Shared memory can be implemented in a similar way

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed? ⇒ reference to the page
- Page reference
  - valid, in-memory ⇒ access the page
  - valid, not-in-memory ⇒ bring to memory (Page Fault)
  - invalid reference ⇒ abort/exception (Seg. Fault)

Transferring Pages between Memory and Disk

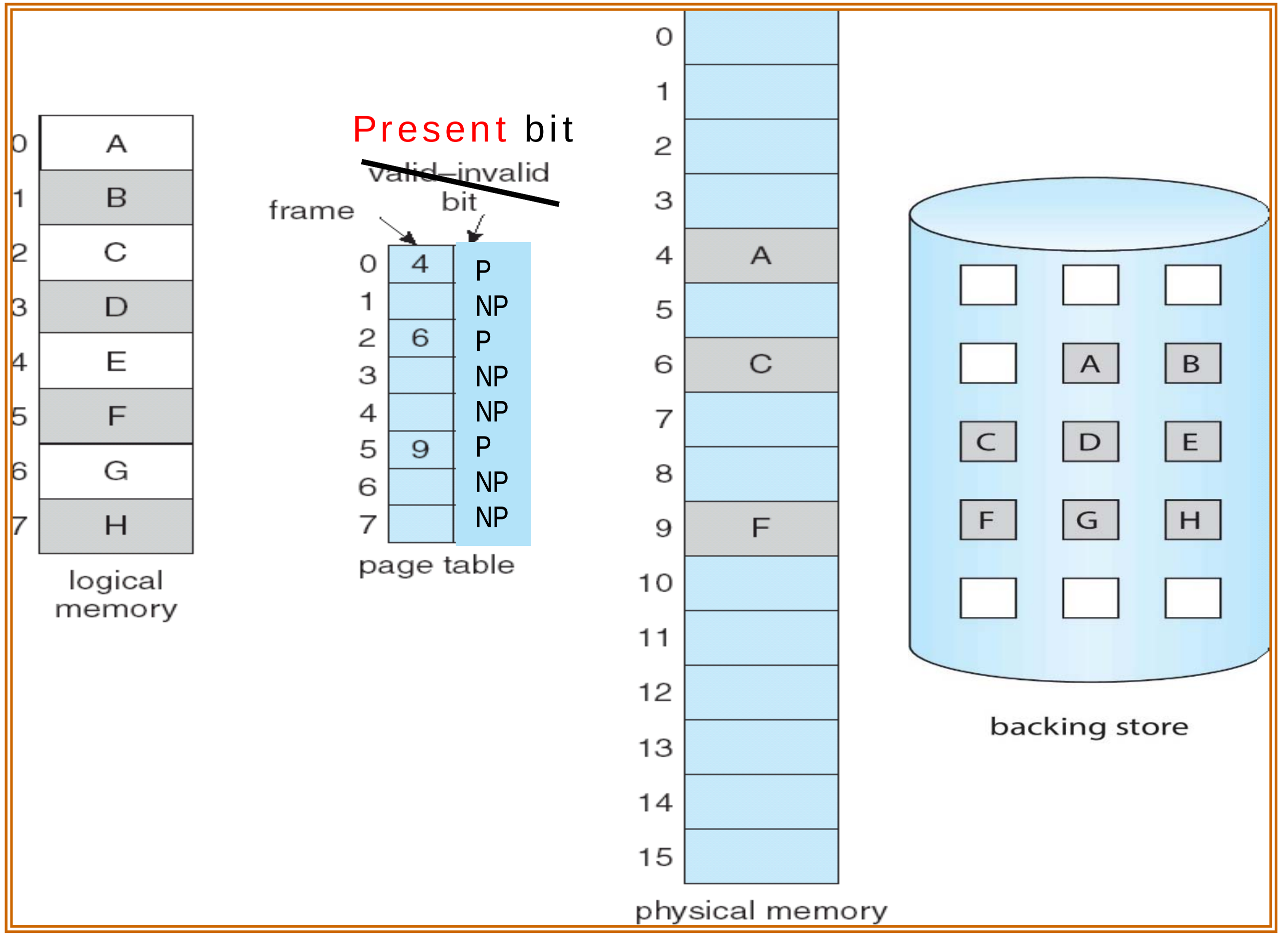


Demand paging is similar to a paging system with swapping
However, it uses a *lazy* swapper ➔ swap in a page only when the page is needed

The lazy swapper is called *pager*…

Not-in-Memory Pages

How do we know if a page is in memory or not?



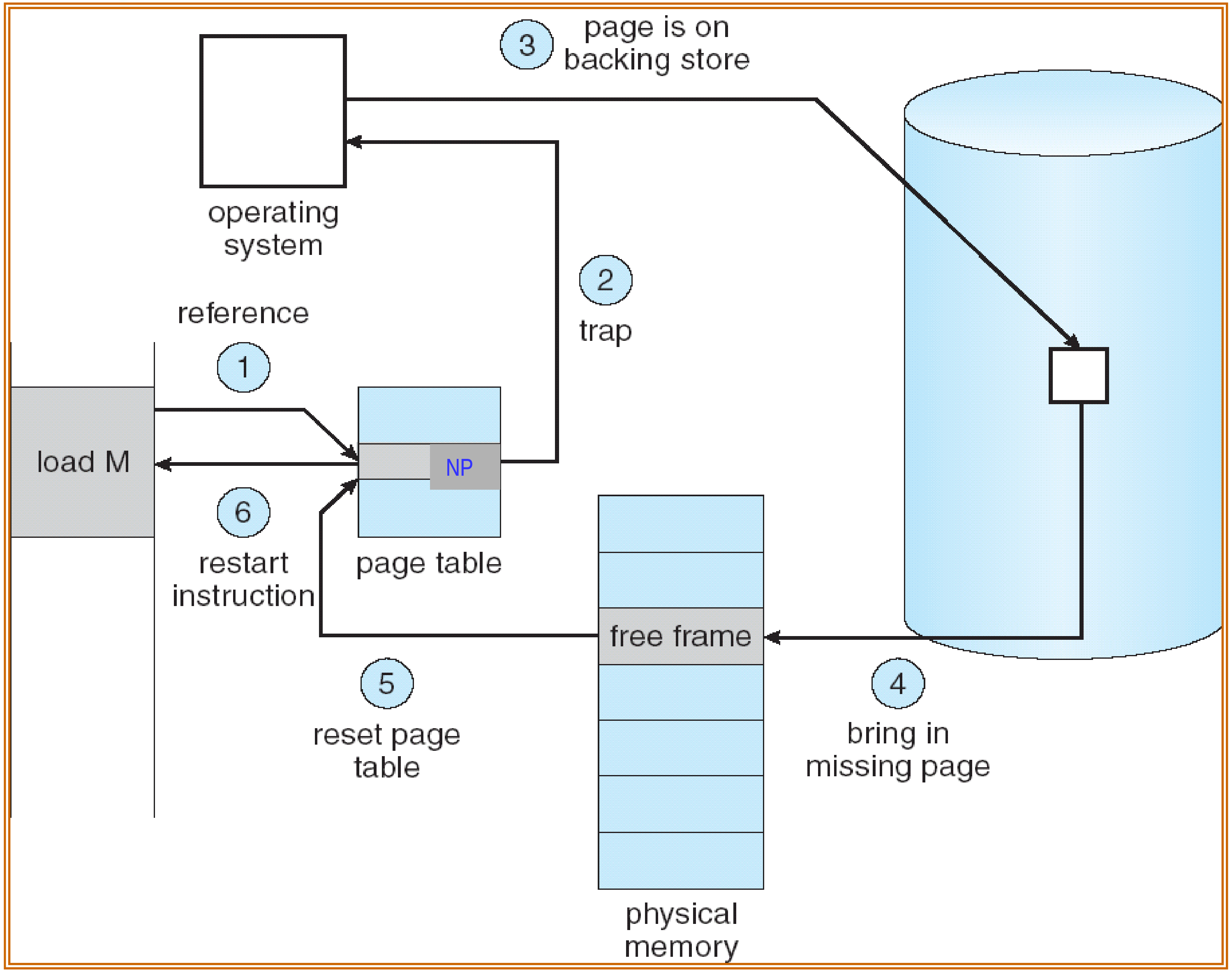
- A **page fault** occurs when a process accesses a not-in-memory (valid) page

Steps in Handling a Page Fault

1. Reference to a non-in-memory page ➔ trap to OS
   – Page fault
2. OS checks its data structures to decide
   a) Invalid reference ⇒ abort (segmentation fault), or
   b) Just not in memory
3. Find free frame
   – If no free frame, page out a used page to make room
   – **page replacement**
4. Page-in the page into a free frame via scheduled disk operation
5. Update PTE and MMU to record the virtual-physical mapping
6. Restart the instruction that caused the page fault

Page faults are transparent to processes!

Steps in Handling a Page Fault



Detailed Page Fault Steps

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk

5a. Read the page from the disk to a free frame

   a) Send a read request to the disk, wait for the disk queue until the read request is serviced
   b) Wait for the disk seek and/or latency time
   c) Begin the transfer of the page to a free frame (DMA)

5b. While waiting, allocate the CPU to other process(es)

   – Rescheduling, change the *current process*

Detailed Page Fault Steps

6. The I/O completed, receive an interrupt (from the disk)
7. Save the registers and state of the current process
8. Determine that the interrupt was from the disk
9. Update the page table and TLB to record the virtual-physical mapping
10. Wait for the CPU to be allocated to this process again
11. Restore the user registers & process state, and then resume the interrupted instruction

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference causes a fault

- Effective Access Time (EAT)

$$\text{EAT} = (1 - p) * \text{memory access time} + p * \text{page fault time}$$

- Page fault time
  - Service the page fault exception (short time)
  - **page in/out (long time)**
  - restart the process (short time)

Demand Paging Example

- Memory access time = 200 ns
- Average page-fault service time = 8 ms

- EAT $= (1 - p) * 200\text{ns} + p * (8\text{ ms})$
  $= (1 - p) * 200 + p * 8{,}000{,}000$
  $= 200 + p * 7{,}999{,}800$

- If $p = 0.001$ ➔ EAT = 8.2 us ➔ 40x slowdown!
- If performance degradation < 10%
  - $220 > 200 + 7{,}999{,}800 * p$
    $20 > 7{,}999{,}800 * p$ ➔ $p < .0000025$
  - < one page fault in every **400,000** memory accesses!!!

Aspects of Demand Paging

- Extreme case – pure demand paging
  - never bring a page into memory until it is required
  - a process is started with no pages in memory
- **Code** pages can be paged in from the executable file, and discarded (rather than paging out, i.e., writing back to disk) when freeing frames
  - Used in Solaris and current BSD
  - Swap space is still required for the other types of pages (e.g., data, stack, heap…)

Implementations of fork()

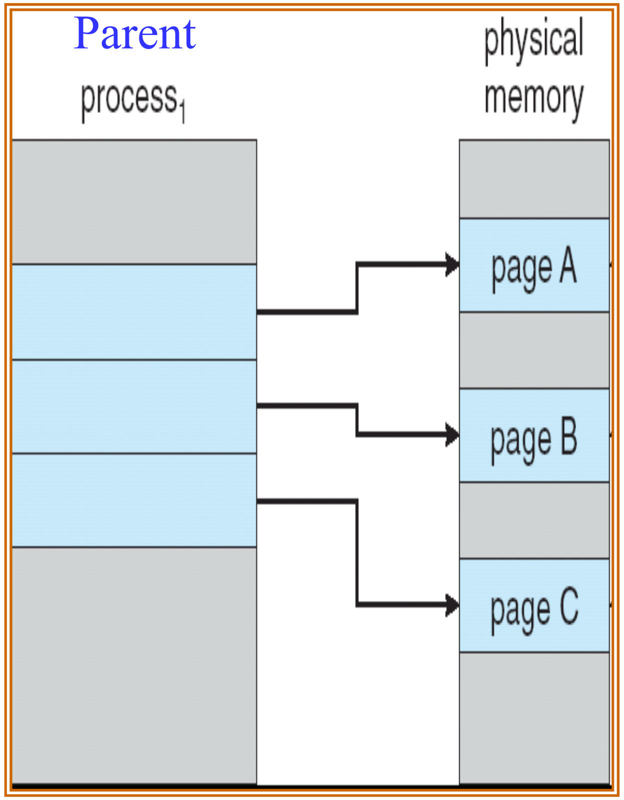
- Three types of implementations
  - Traditional fork
  - vfork
  - COW (Copy-on-Write) based fork

- Traditional fork
  - Copy all the frames from parent to child
  - Large memory copy overhead

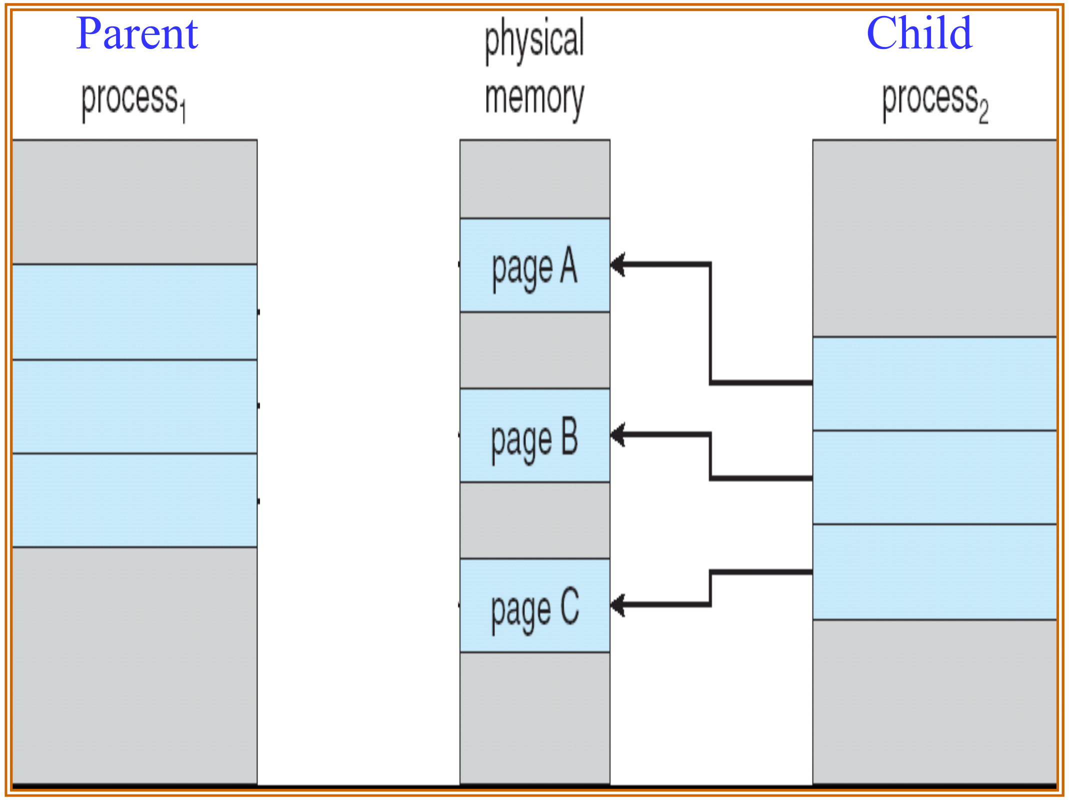
vFork

- Intended to be used when the child calls *exec()*
- The parent process is suspended
- The child process ***borrows*** frames from the parent
  - Changes to the pages are visible to the parent once the parent resumes
- The borrowed frames are returned when the child call *exec()*

vFork



Before fork

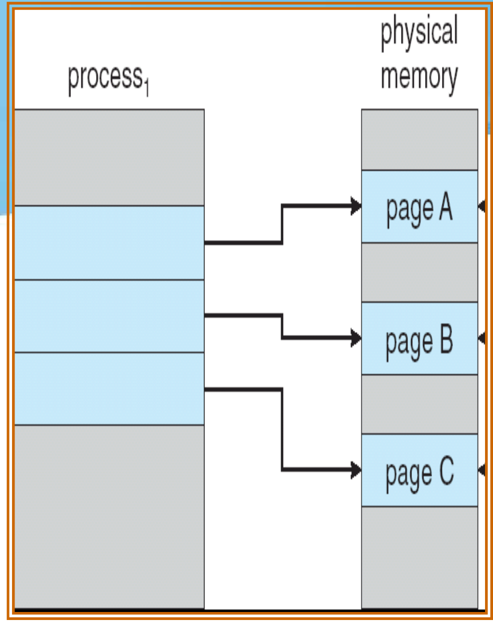


After fork

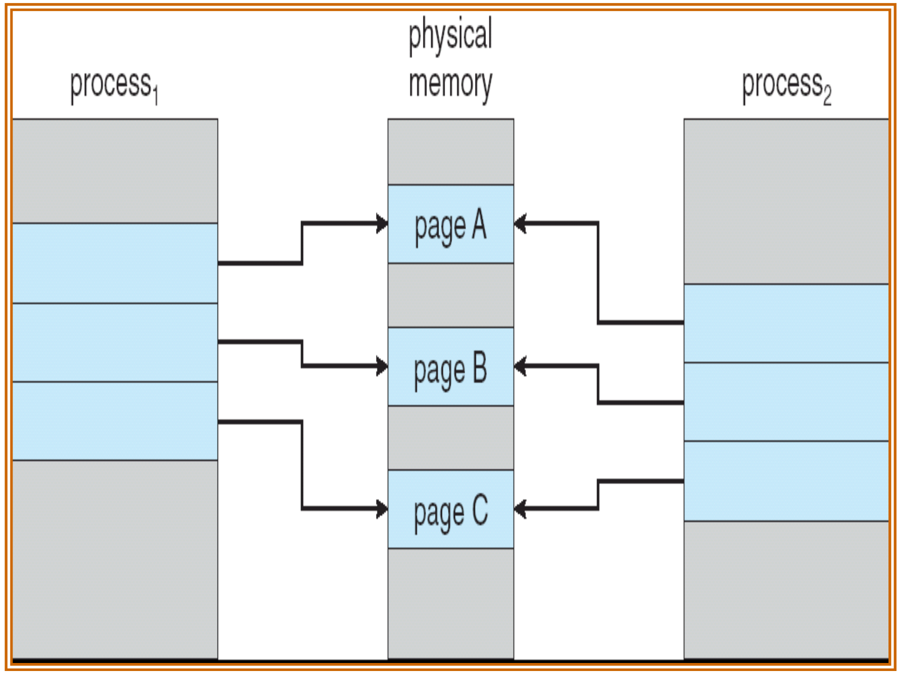
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
  - A way to implement fork()
  - Based on page fault
    - Not-in-memory page fault
    - **Protection violation** page fault
- The shared page is set as read-only initially. If either process modifies the shared page
  - Page fault occurs, and
  - The page is copied
- COW allows more efficient process creation as only modified pages are copied

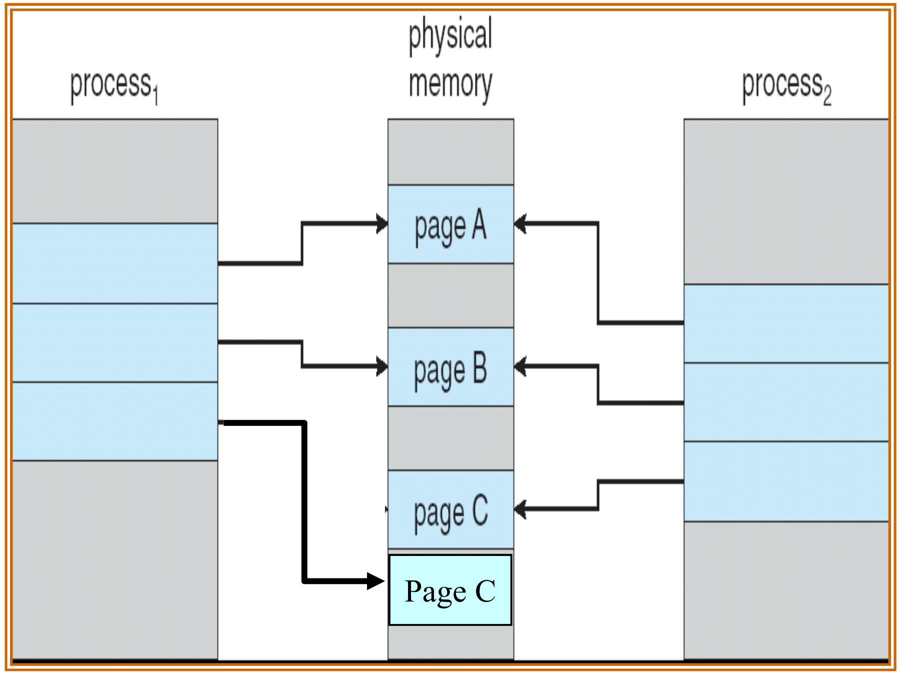
Copy-on-Write Fork



Before fork



After fork



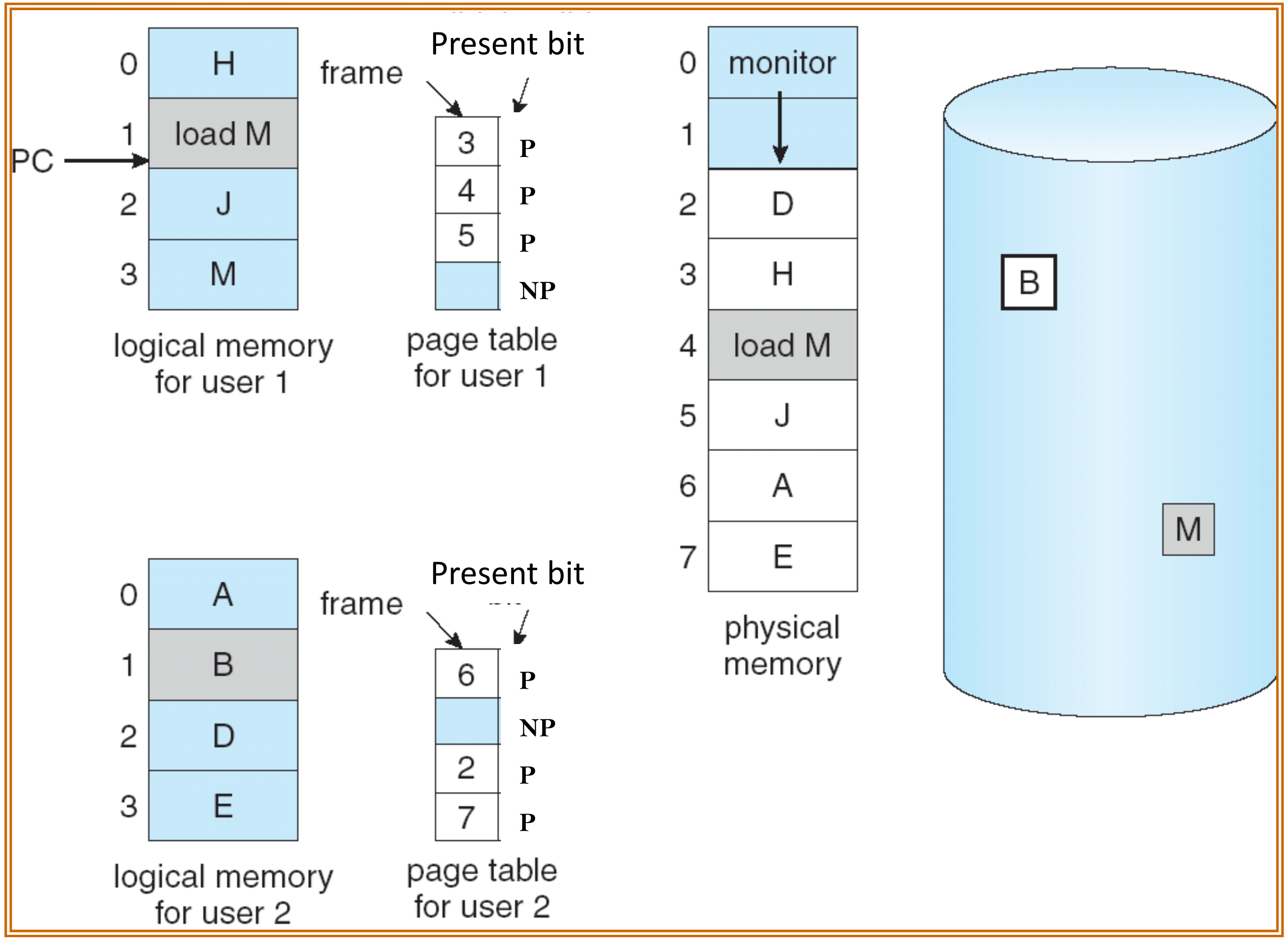
When parent tries to update page C

Page Replacement

- Swap in a page when all the frames are occupied ➔ page replacement is needed

- Page-fault service routine needs to deal with page replacement

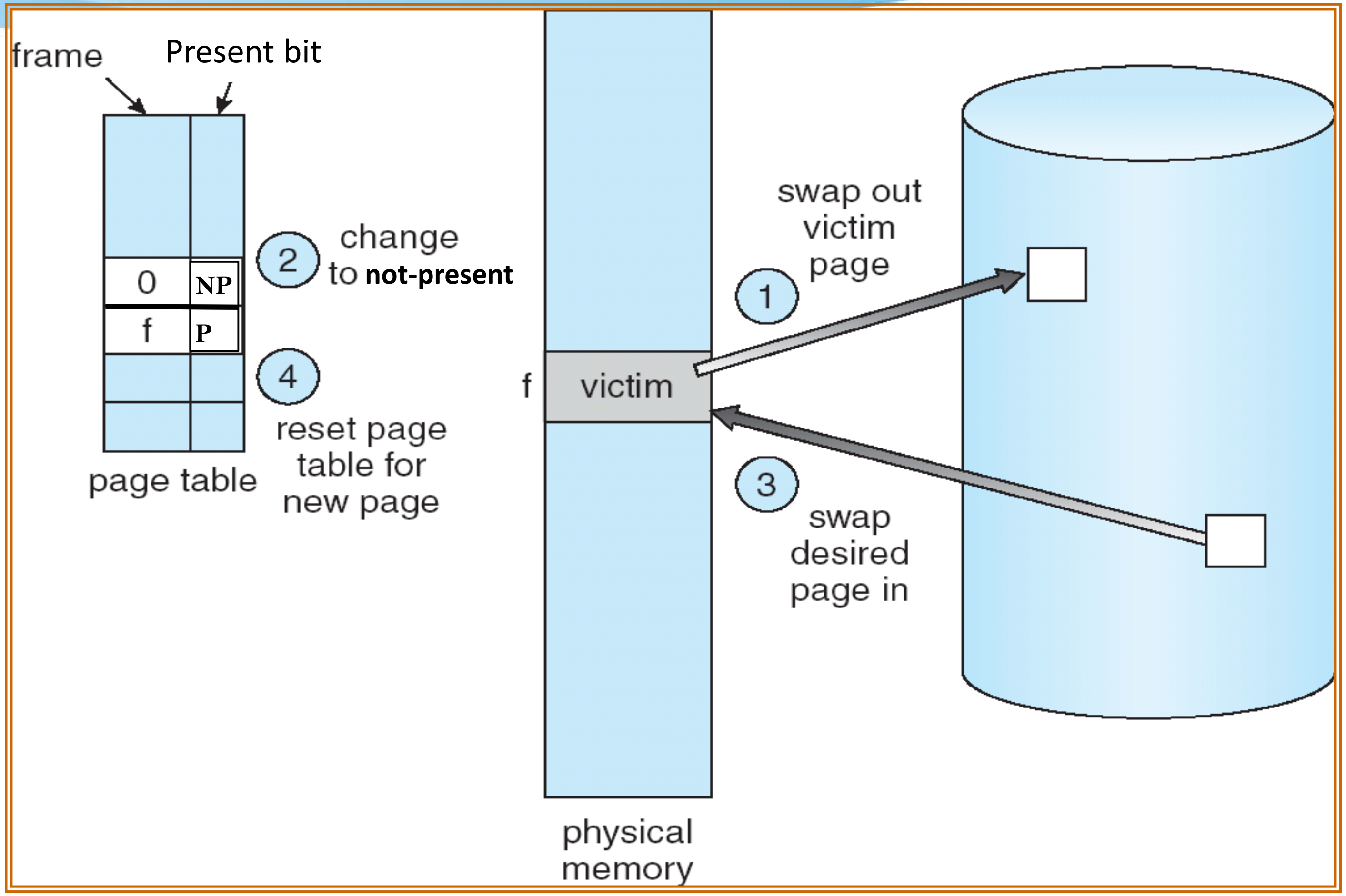
Need for Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Write the page in the victim frame to the disk (i.e., page out) if the page is ***dirty***
4. Read the desired page into the (newly) free frame (i.e., page in)
5. Update the page tables
6. Restart the instruction that caused the page fault

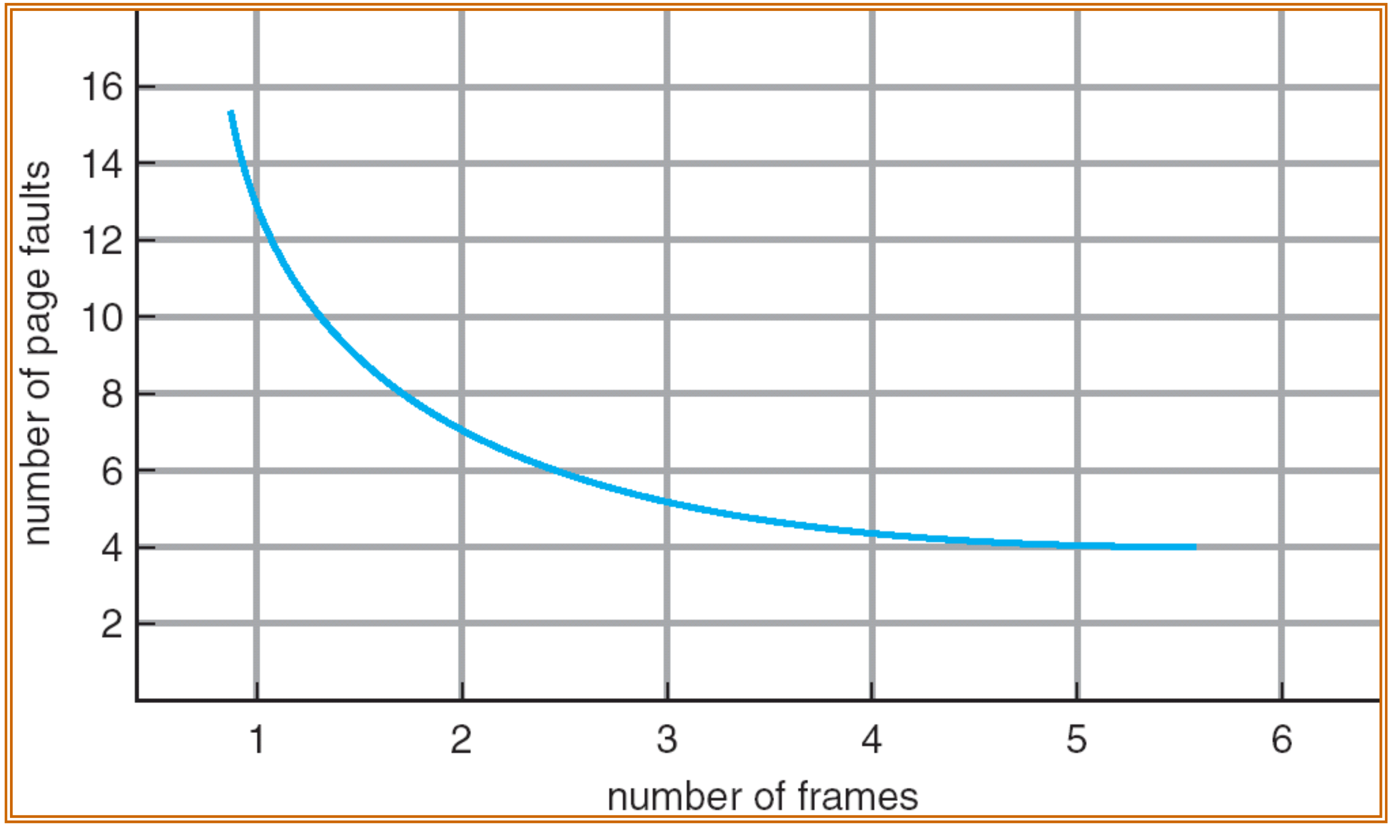
Page Replacement



.Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

.Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Number of Page Faults vs. Number of Frames

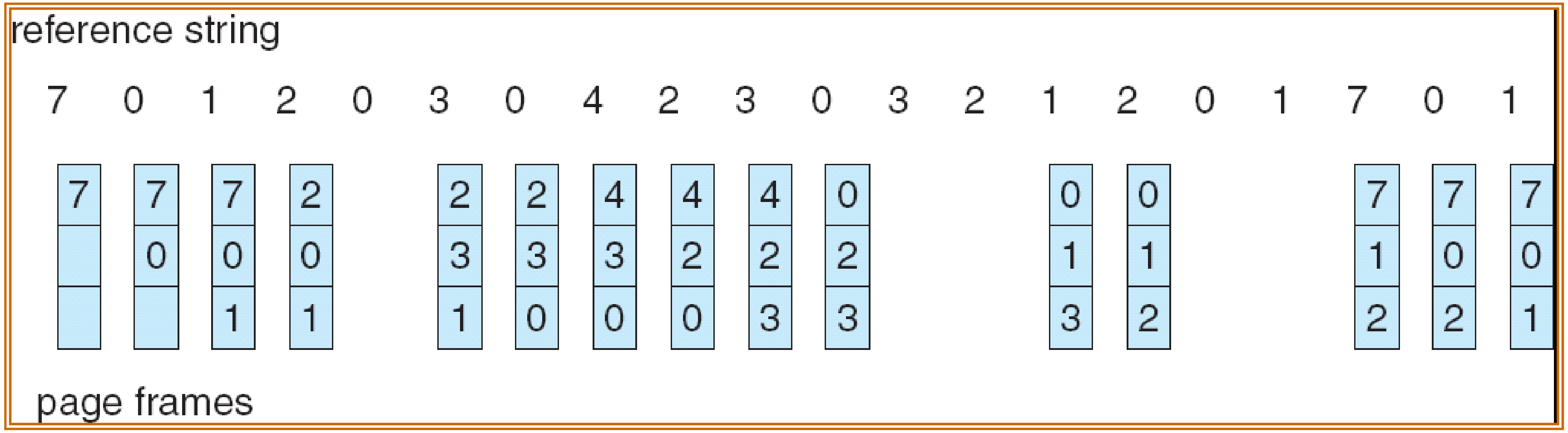


Page faults slow down the system

A good page replacement algorithm should not cause high page faults…

FIFO Page Replacement

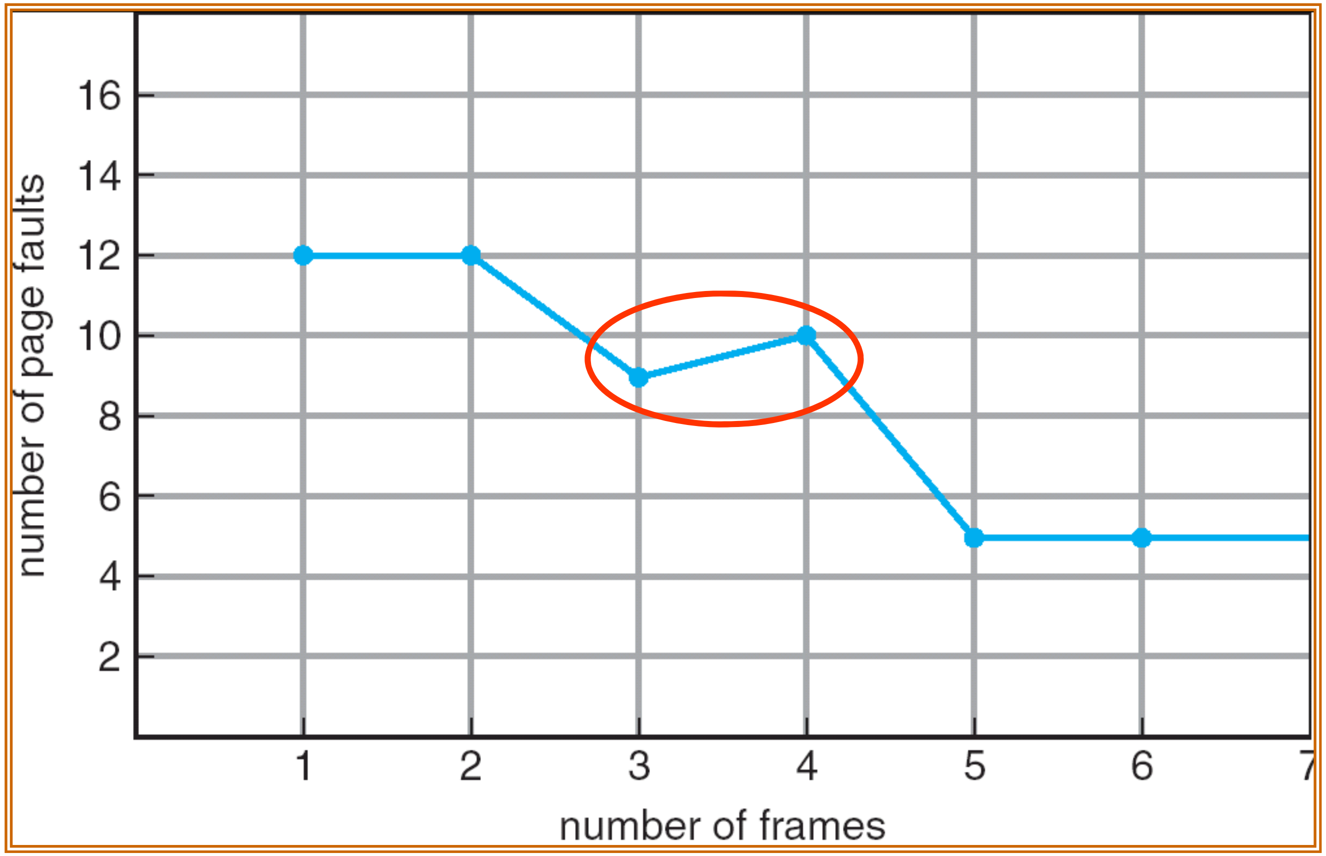
- **Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- **Number of frames: 3**



- **15 page faults in this case**
- **FIFO is not always good**
  - e.g. first-in != seldom-used
- **Suffers from the belady's anomaly**

Belady's Anomaly

Page reference string: 1 2 3 4 1 2 5 1 2 3 4 5



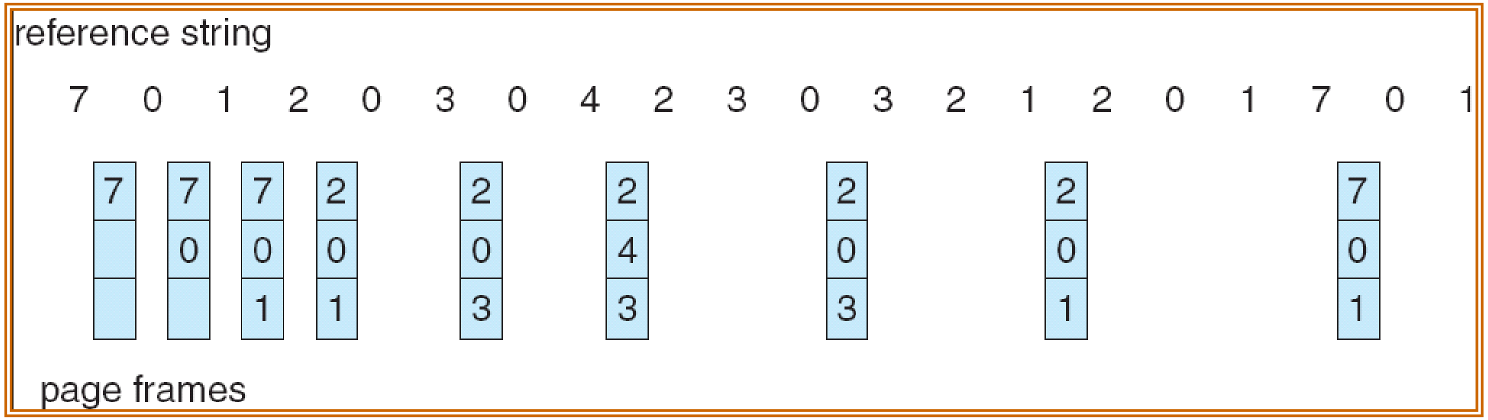
Belady's anomaly: page fault rate may increase as the number of allocated frames increases

Stack Algorithms

- Can be shown that the set of pages in memory for **n** frames is always a subset of the set of pages that would be in memory with **n+1** frames
- Never exhibit belady's anomaly
- FIFO is not a stack algorithm, prove it by yourself…
  - You can test the cases of 3 & 4 frames

Optimal Page Replacement

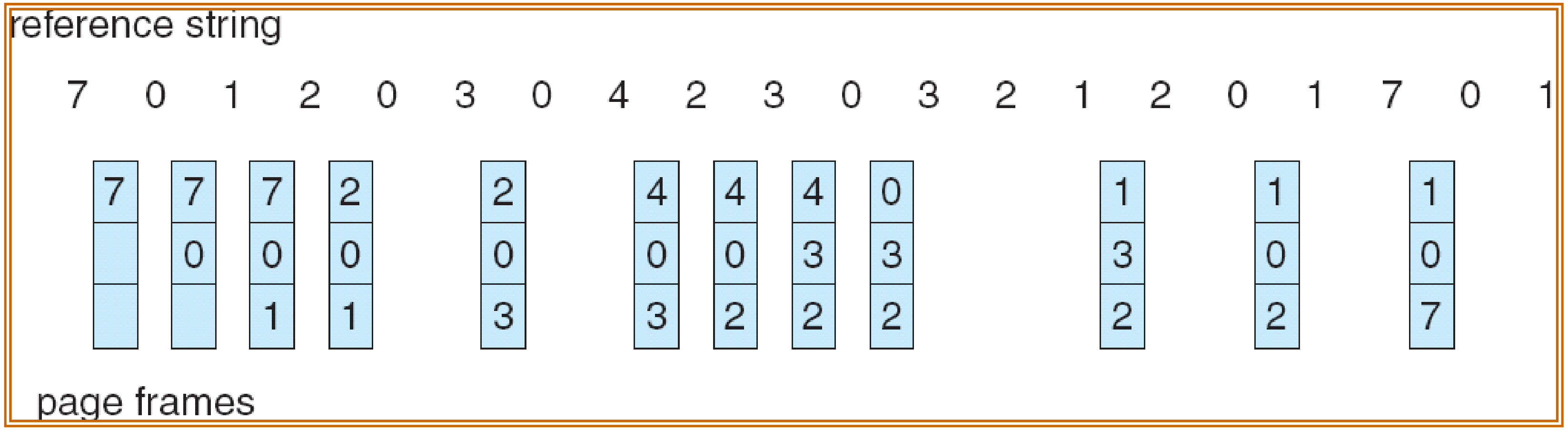
- Replace the page that will not be used for the longest period of time



- **9 faults in this case**
- **Has the lowest page fault rate**
- **Never suffers from the belady’s anomaly**
- **Optimal, but NOT Feasible !**

LRU (Least Recently Used)

- Replace the page that *has not been used* for the longest period of time
  - Use the recent past as the approximation of the near future



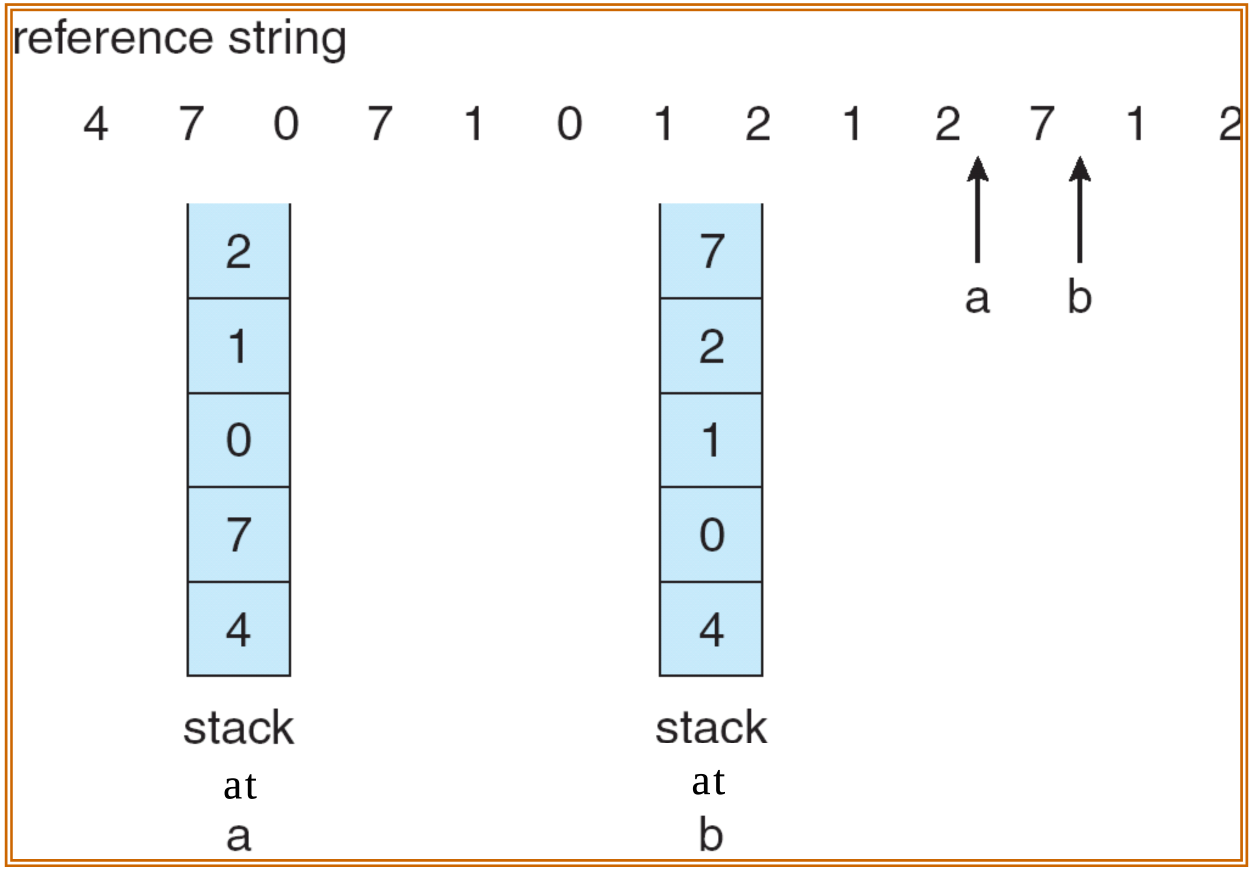
- **12 faults in this case**
- **Never suffers from the belady's anomaly**
- **How to implement it?**
  - **-Clock counters (Timers)**
  - **-Stack**

LRU Implementation Based on Timer

- Associate with each PTE a time-of-use field (i.e. clock counter)
  - Access a page ➔ update the field with current time
- When page replacement is needed, find the page with the smallest value of the counter

- Problems
  - Requires a search of the PT to find the LRU page
  - Clock counters can overflow

LRU Implementation Based on Stack



- Record the access order, instead of the absolute access time
- Use doubly-linked list to implement the stack
  - because removal from the stack is needed

LRU

- Counter and stack implementations are not efficient without special HW support
  - Updating the clock field or the stack must be done for **every** memory reference
  - Without HW support, use an interrupt for every reference to allow the SW to update the timer/stack ➔ Huge overhead !!!
- **Slow** even with HW support
  - Update multiple pointers for each memory reference, or search the PTEs to find the smallest counter value
- HW support is required, but it should be **efficient** and **easy to implement**
  - Several approximation approaches have been proposed

LRU Approximation Algorithms

- **Reference bit algorithm**
  - With each page associate a bit, initially = 0
  - When page is referenced, the bit is set to 1
  - Replace the page with reference bit = 0 (if one exists)
  - However, we do not know the access order
    - No reset!!!! Pages that have not been accessed for a long time cannot be identified…

LRU Approximation Algorithms

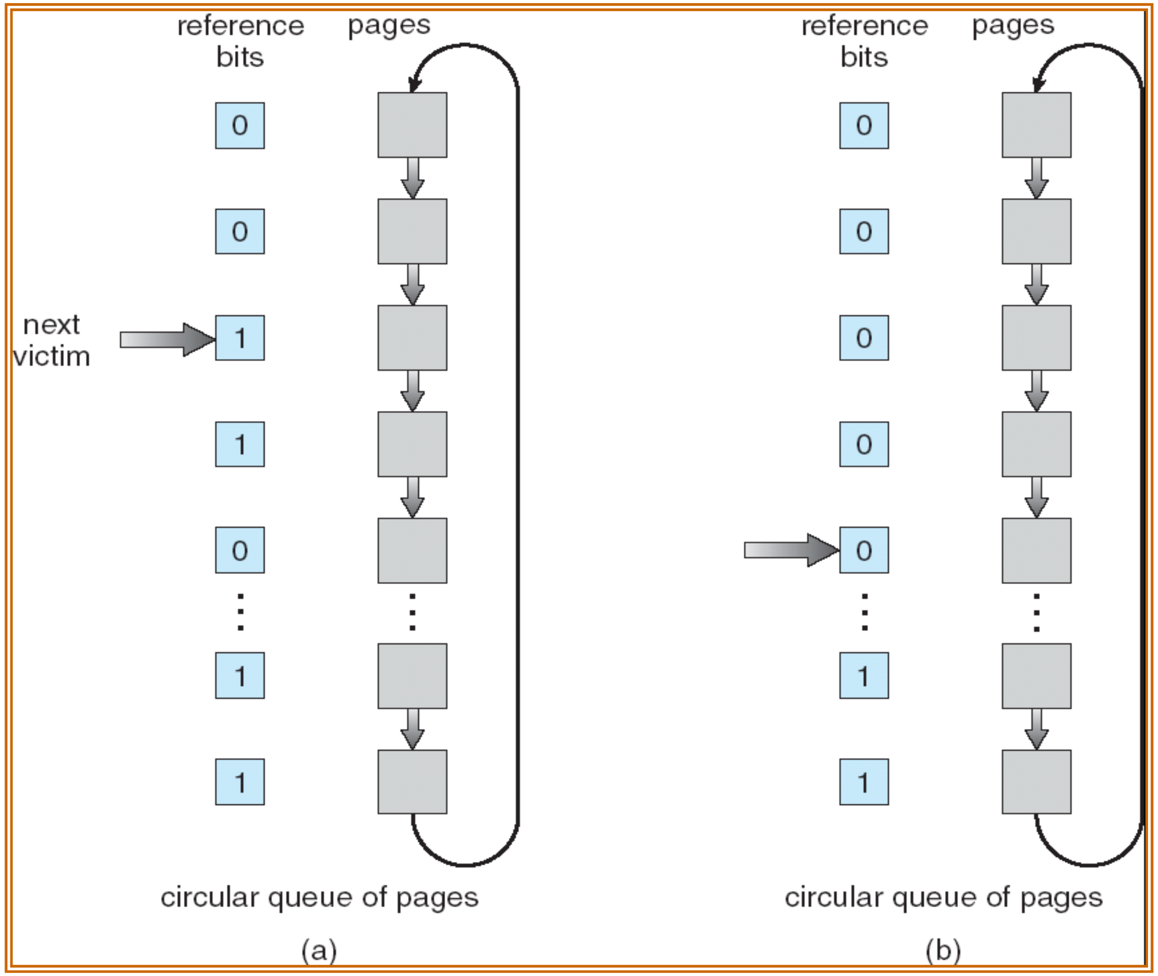
- **Additional reference bit algorithm**
  - Keep several history bits (e.g., 8 bits) for each page
  - Periodically shift the reference bit into the MSB of the history bits
    - Reference bit becomes 0 after each shift
    - History bits of pages that have not been accessed for a long time will become 0…
  - Replace the page with the lowest number

LRU Approximation Algorithms

- **Second chance algorithm (clock algorithm)**
  - Need reference bit
  - Clock replacement
    - Scan the PTEs in a clock order
  - If the page has reference bit = 0 ➔ replace it
  - If the page has reference bit = 1
    - set reference bit as 0
    - leave the page in memory
    - check next page in clock order
  - If the page is accessed often enough, it will never been replaced

Second-Chance (Clock) Page-Replacement Algorithm

implemented by using a circular queue



LRU Approximation Algorithms

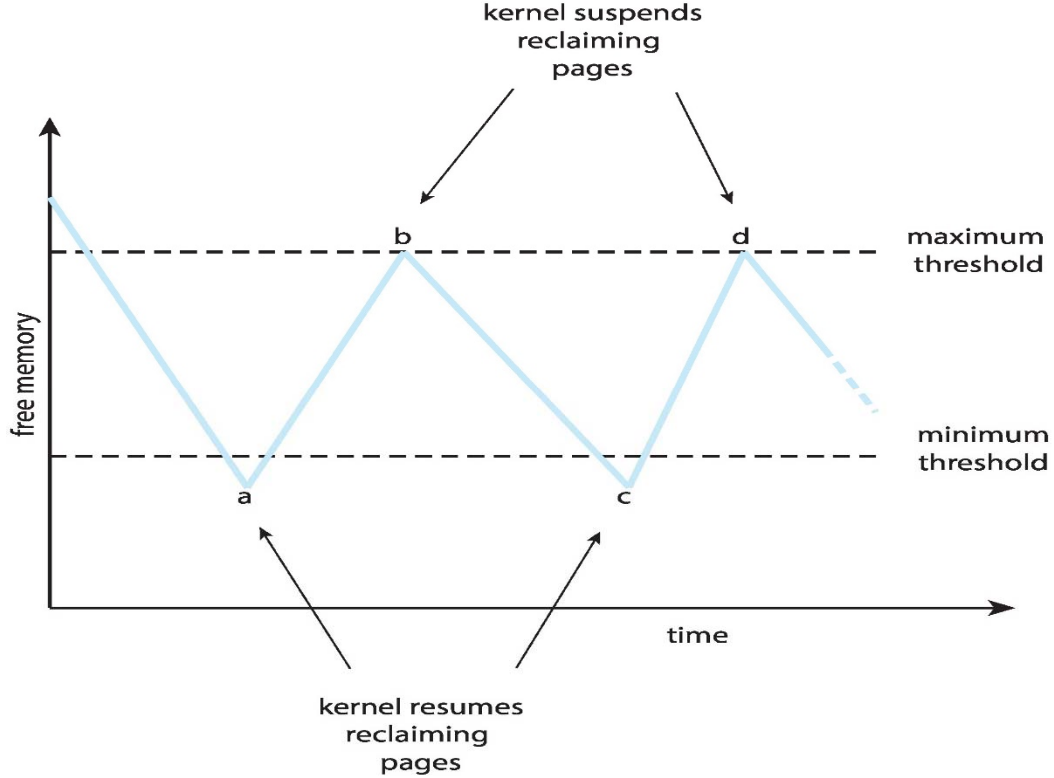
- **Enhanced second chance algorithm**
  - Consider the pair (reference bit, modify/dirty bit) and divide the pages into four classes
    - (0, 0): neither recently used nor modified
      - Best page to replace
      - Why do we check the modify (dirty) bit? **Reduce IO**
    - (0, 1): not recently used but modified
    - (1, 0): recently used but clean
    - (1, 1): recently used and modified
      - Should keep in memory
  - Replace the first page encountered in the lowest nonempty class

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
  - Since an actively used page should have a large reference count
  - A page used heavily before but never used recently still remains in the memory
    - *Aging* (*i.e., reduce the counts periodically*) can be used here
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page Buffering Algorithms

- Several buffering mechanism to reduce the page fault time
  - Keep a pool of free frames
    - Fewer page replacement operations in page faults



Page Buffering Algorithms

- Several buffering mechanism to reduce the page fault time
  - Other techniques…
    - Write back the modified pages to the paging device when the device is idle (before the pages are selected as victims)
      - Fewer page out operations in page replacement
    - Remember which page was in each frame in the free-frame pool
      - The old page can be reused from the free-frame pool before that frame is rewritten

Allocation of Frames

- Each process needs a *minimum* number of pages
  - Must at least hold the pages that any **single instruction** can reference
- Example 1: in a machine that all memory reference instruction have only one memory address (and no indirection)
  - 2 frames are needed (1 for instruction, 1 for data memory reference)
- Example 2: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Example 3: multiple level of indirection
  - LD r1, [[100]] ➔ 3 pages are needed

Allocation of Frames

- Each process should have a minimum number of pages
- Beyond this number, how many pages a process should have?
  - Equal allocation
  - Proportional allocation
    - Allocate available memory to each process according to its size

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

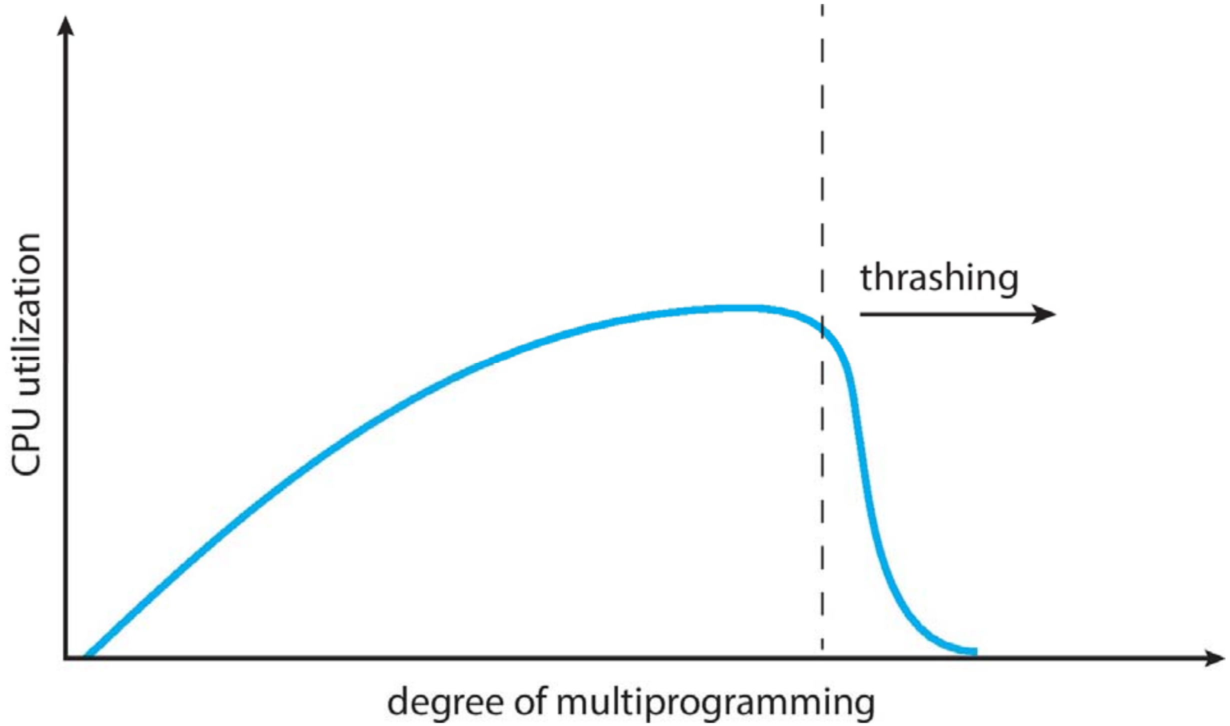
Global vs. Local Allocation

- Global replacement algorithms
  - Select a replacement frame from all frames
  - A process cannot control its own page fault rate
- Local replacement algorithms
  - A process select a replacement frame from its own frames
  - Number of frames allocated to a process doesn't change
  - Seldom used frames of the other processes cannot be used by the process
- Global replacement is more common since it generally results in better throughput…

Thrashing

- If a process does not have "**enough**" page frames, the page-fault rate will be quite high
  - Since the swapped out pages will be swapped in soon
- In early systems, this led to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system, page out some other pages
- **Thrashing**
  - a process is busy swapping pages in and out

Thrashing (Cont.)



Thrashing can be prevented based on

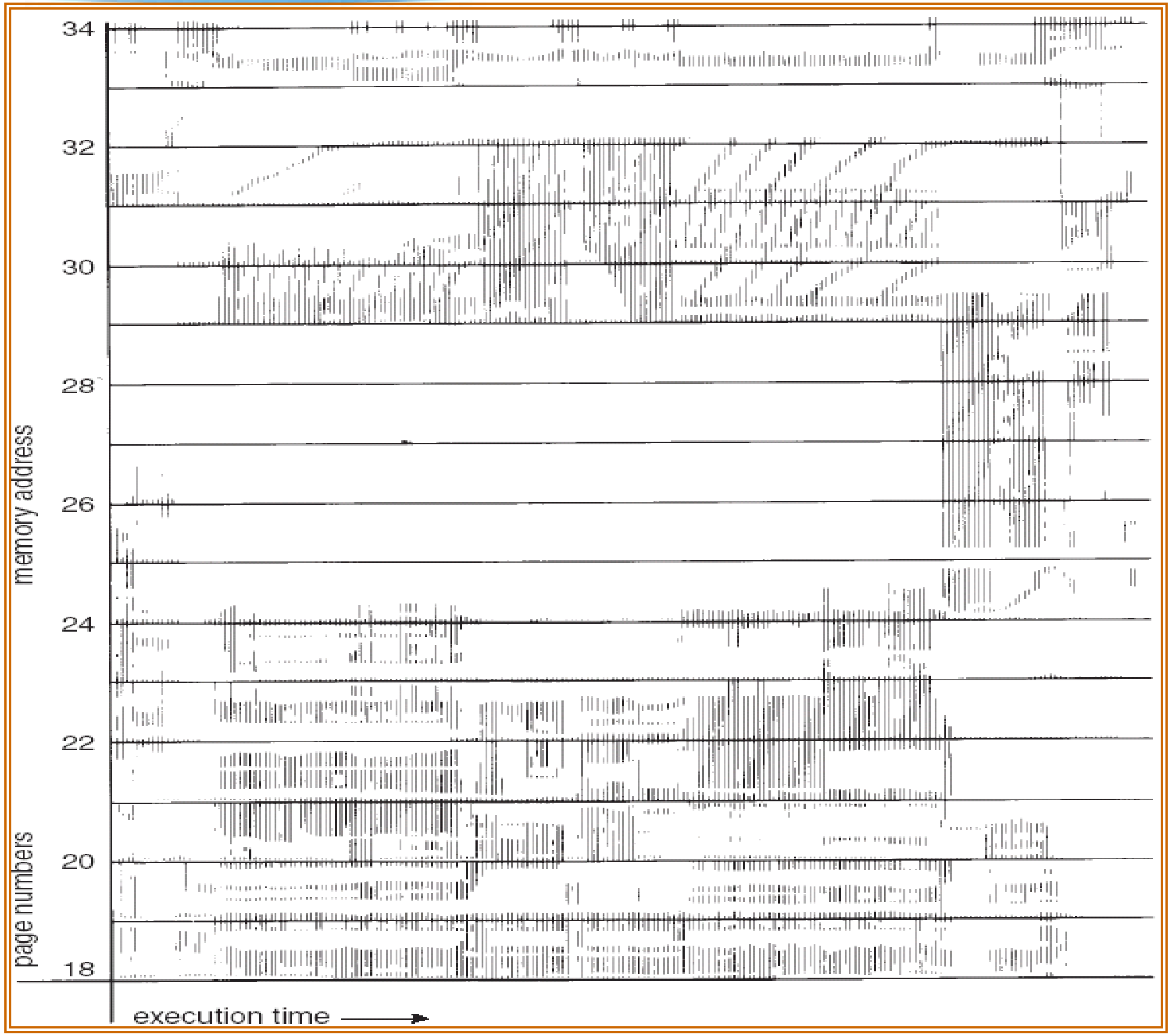
- Working set model, or

- Page fault frequency

Locality

- A process should have enough frames to prevent thrashing
- How do we know the number of frames needed by a process?
- The locality model
  - As a process executes, it moves from locality to locality
  - A locality: set of pages that are actively used together
    - E.g., a function call changes from one locality to another
- Allocate frames to accommodate the current locality of each process

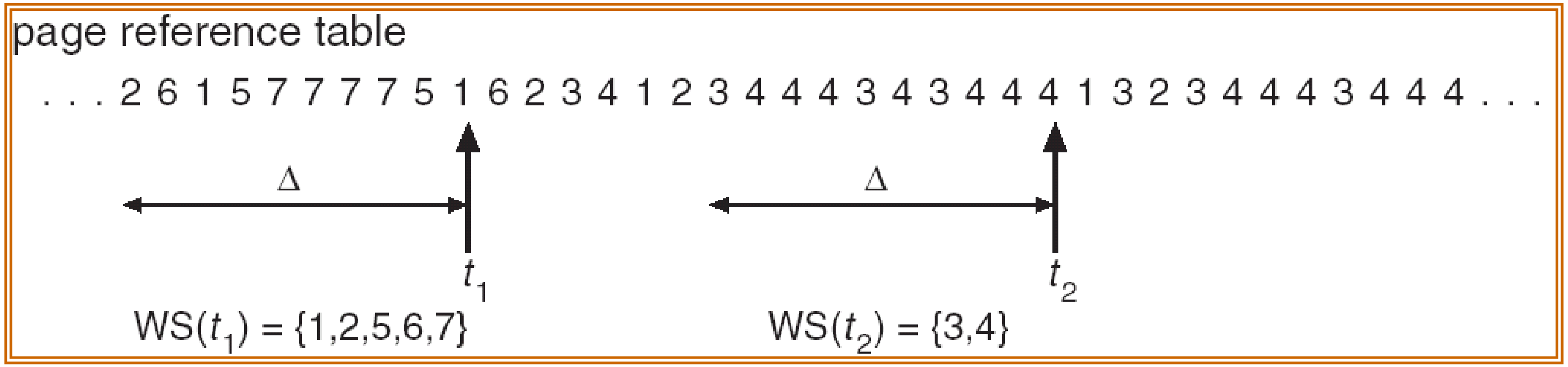
Memory Reference Pattern



Working-Set Model

- Based on the assumption of locality
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 references
- WSS_i (working set size of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small ➔ will not encompass entire locality
  - if Δ too large ➔ will encompass several localities
  - if $\Delta = \infty$ ➔ will encompass entire program
- **Allocate at least *WSSi* pages for *Pi***
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- if $D > m$ (total # of available frames) $\Rightarrow$ Thrashing
  - if $D >$ m, suspend processes 1-by-1 until $D <= m$

Working-Set Model



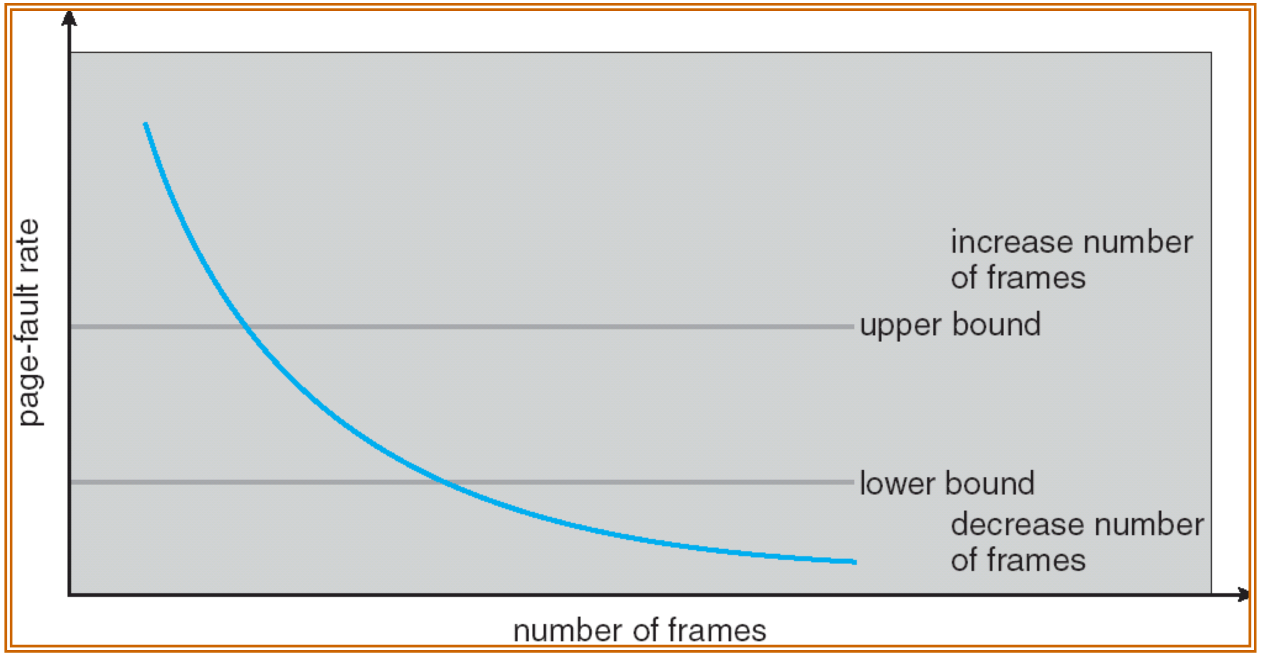
Tracking the WSS at each memory reference leads to a huge overhead

Implement the WS model by using timer interrupt & reference bit

- just an approximation

Page-Fault Frequency Scheme

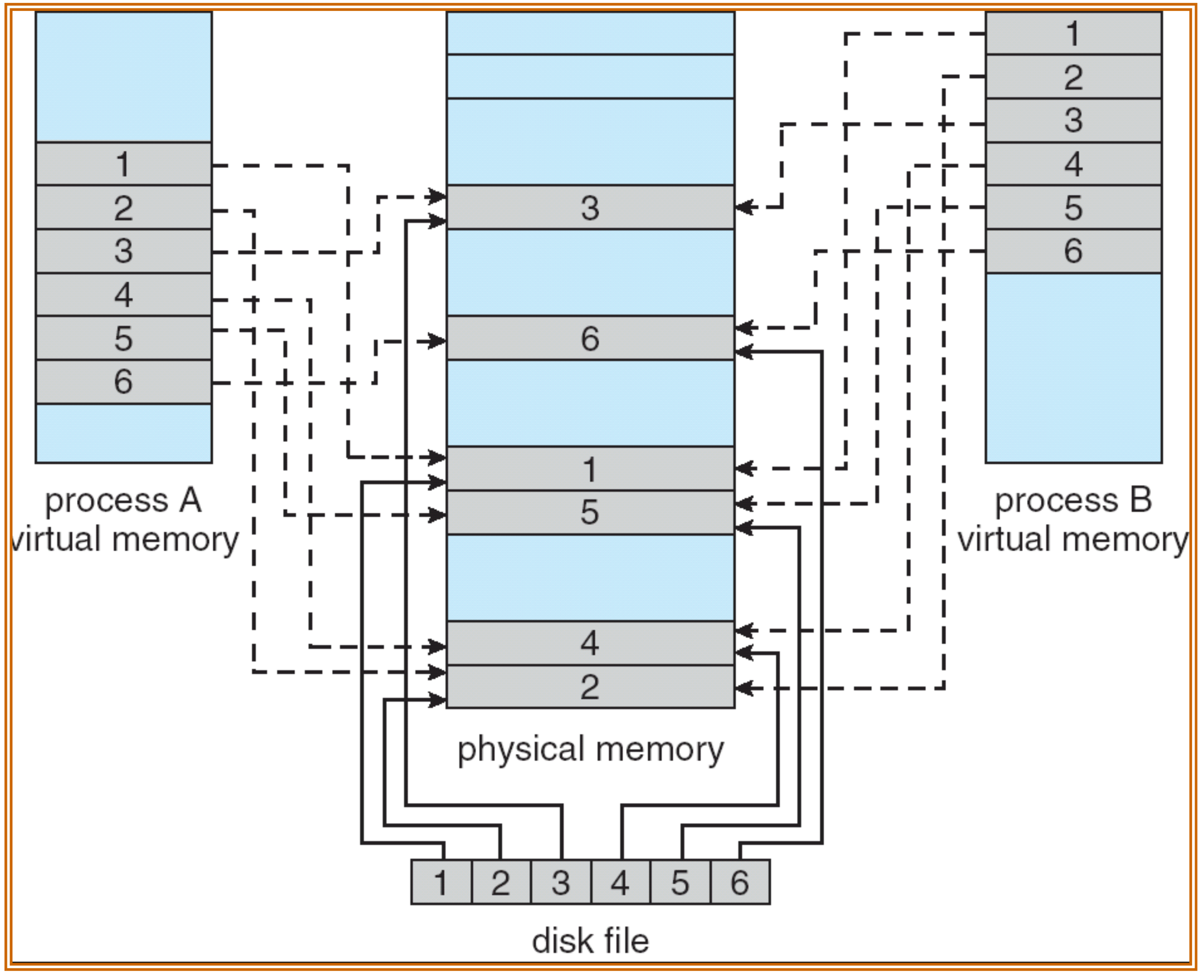
- Another way to prevent thrashing
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
  - If not all the process have acceptable rates ➔ suspend a process



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as memory access by mapping a disk block to a page in memory
  - Usually, using the mmap() system call
- A file is read using ***demand paging***. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- **Avoid** read()/write() system calls during file access
  - Reduce overhead
- Also allows several processes to map the same file→ allowing the pages in memory to be shared

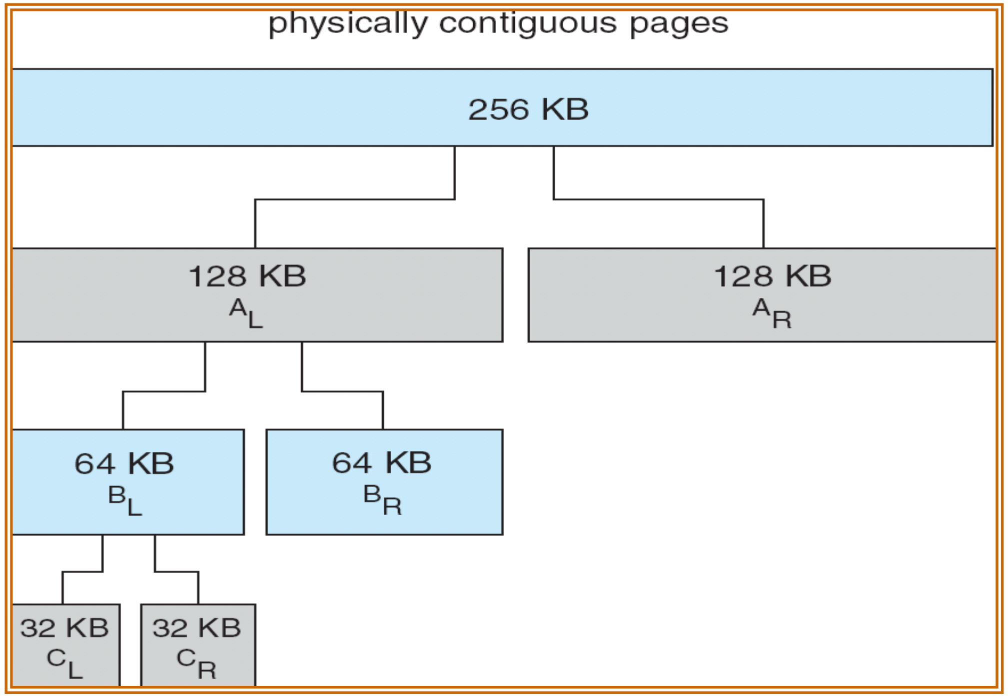
Memory Mapped Files



Shared memory is realized by using memory mapped files in Windows NT,2000, XP

Allocating Kernel Memory

- **Buddy System**
  - Power-of-2 allocator
  - Split (if necessary) when allocation
  - Coalesce (if possible) when free



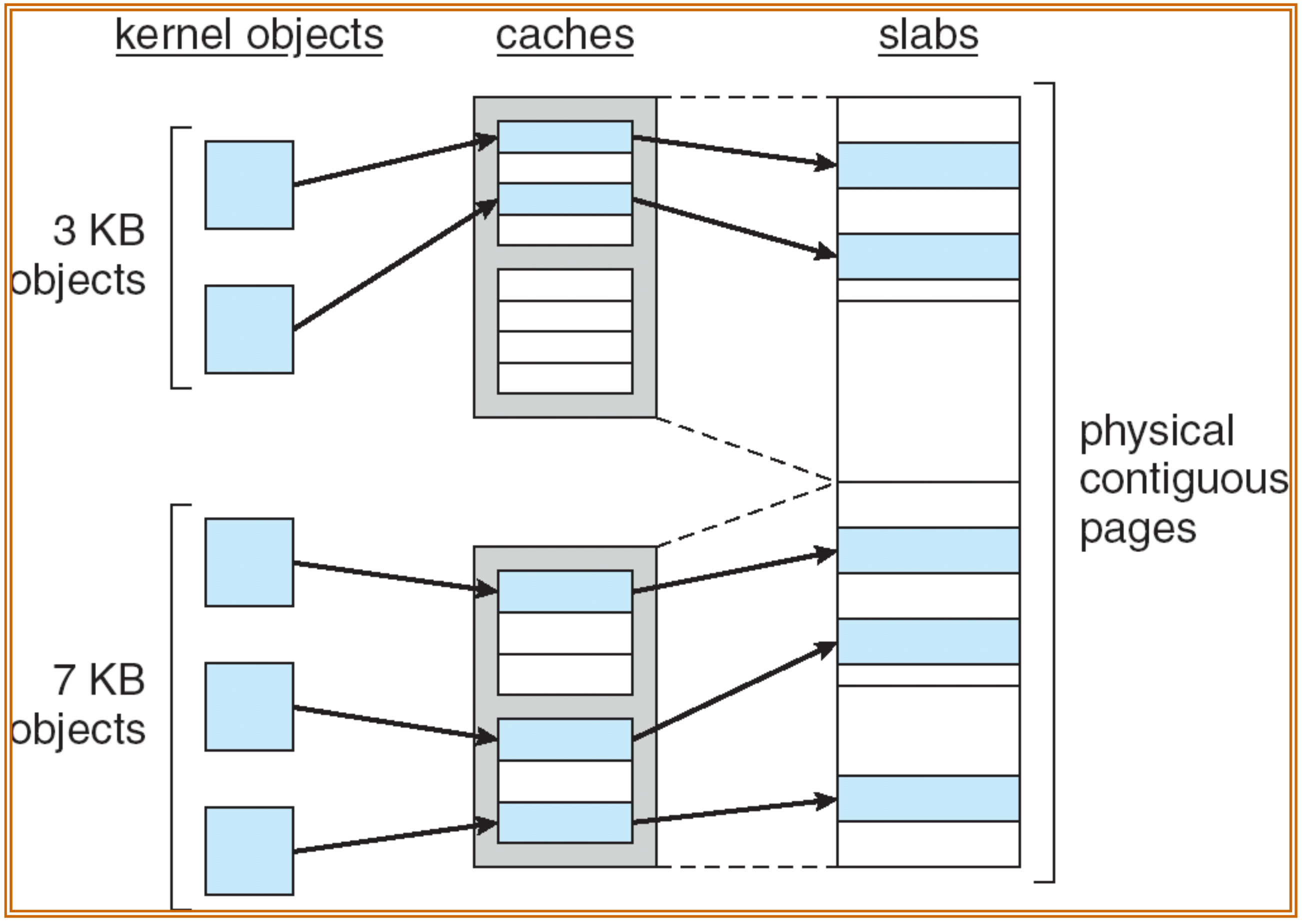
- Advantage
  - ✓ quick alloc/free
- Disadvantage
  - ✓ fragmentation

Allocating Kernel Memory

- **Slab Allocator**
  - **Slab**: one or more physically contiguous pages
  - **Cache:** one or more slabs
  - **a cache for each unique kernel data structure**
    - Each cache filled with a type of **objects** (data structures)
  - When cache created, filled with free objects
  - When structures stored, objects marked as used
  - If slab is full of used objects, use a new empty slab for the coming requests
  - Benefits: no fragmentation, quick allocation/free

Allocating Kernel Memory

- Slab allocator



Slab Allocator in Linux

- PCB in Linux: struct task_struct
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Other Considerations of a Paging System

- Major decisions of a paging system
  - Page replacement algorithm
  - Page allocation policy
- Other considerations
  - Prepaging
  - Page size
  - TLB reach
  - Inverted Page Tables
  - Program structure
  - IO interlock

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and a fraction α of the pages is used
  - Is cost of $s * \alpha$ saved pages faults > or < the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
  - α near zero $\Rightarrow$ prepaging loses

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (Number of TLB Entries) x (Page Size)
- **Ideally**, the working set of each process is stored in the TLB. Otherwise, a lot of TLB misses and page table accesses.
- Increase the Page Size can increase TLB reach
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
  - Solaris on UltraSPARC uses both 8KB and 4MB page sizes
- Supporting multiple page sizes requires OS to manage TLB
  - TLB contains the page size field, which is managed by the OS

Page Size

- Page size selection must take into consideration
  - Large page size, good for
    - page table size
    - page fault #
    - I/O overhead
      - Transferring a large page is more cost effective
    - TLB reach
  - Small page size, good for
    - locality
    - fragmentation

Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - OS allocates **fewer than** 128 frames to this process
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

Data structures

-pointers, hash ➔ poor locality

-stack➔ good locality

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

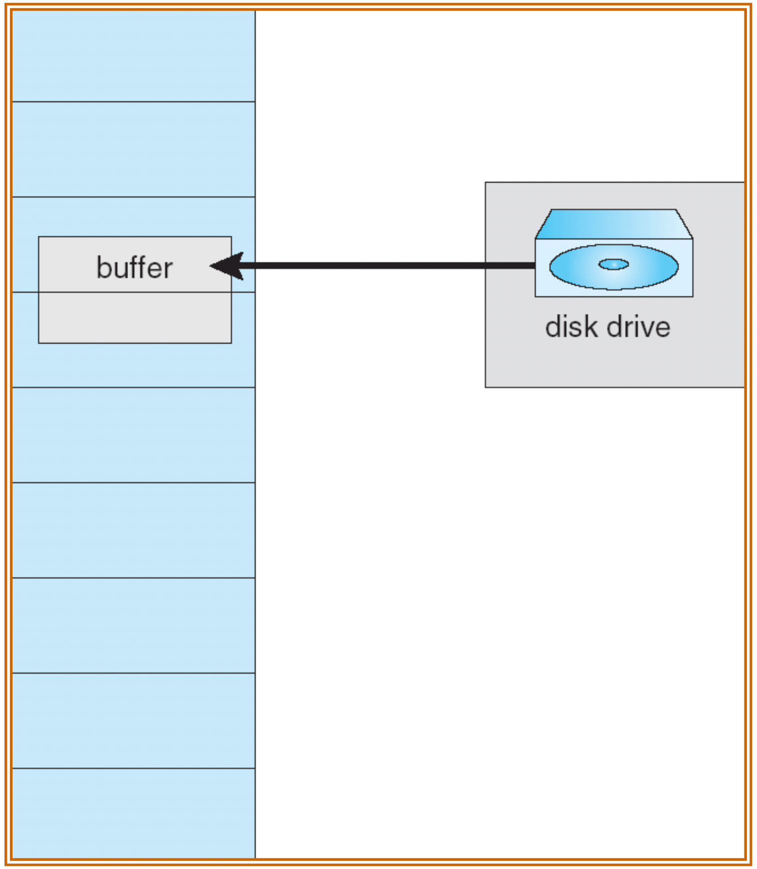
128 page faults

Inverted Page Table (IPT)

- Reduce the memory requirement of the page table
- However, IPT does not contain the complete information about the VAS of a process
  - No information about the not-in-memory pages!!
    - E.g., where the page is (in the swap area), the protection bits…
- Therefore, an external page table (one per process) must be kept
  - Just like the traditional page tables
  - These tables are referenced only when a page fault occurs
  - These tables are themselves paged in/out
  - A page fault may cause the VM manager to generate another page fault to page in the external page table

I/O Interlock

- **I/O Interlock** – pages must sometimes be ***locked*** in the memory
  - e.g. pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.
- Each page is associated with a ***lock*** bit



Operating System Examples

- Windows XP
- Solaris

Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign to faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by ***pageout* process**
- Pageout process scans pages using a modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner