Virtual Memory – Learning Objectives

MMU: keep many processes in memory simultaneously to allow multiprogramming.

- **Requirement:** entire process must be in memory before it can execute.

- Allows the execution of processes that are **not completely in memory**.
  - Pages are loaded into memory using **demand paging**
  - FIFO, optimal, and LRU page-replacement algorithms
  - Working set of a process, and explain how it is related to program locality
  - How Linux manages virtual memory

Background

- **Fact**: Code needs to be in memory to execute, but entire program **rarely** used
  - Error code, unusual routines, large data structures
- 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
  - Increased **CPU utilization and throughput** with no increase in response time or turnaround time
  - Less I/O needed to load or **swap** programs into memory -> each user program runs faster

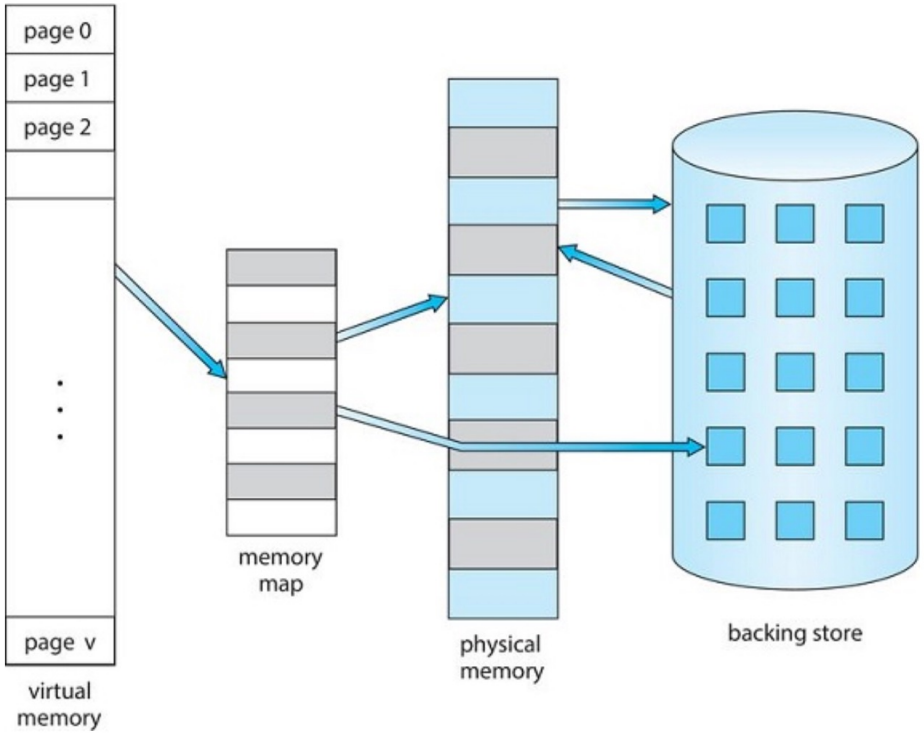
Background (Cont.)

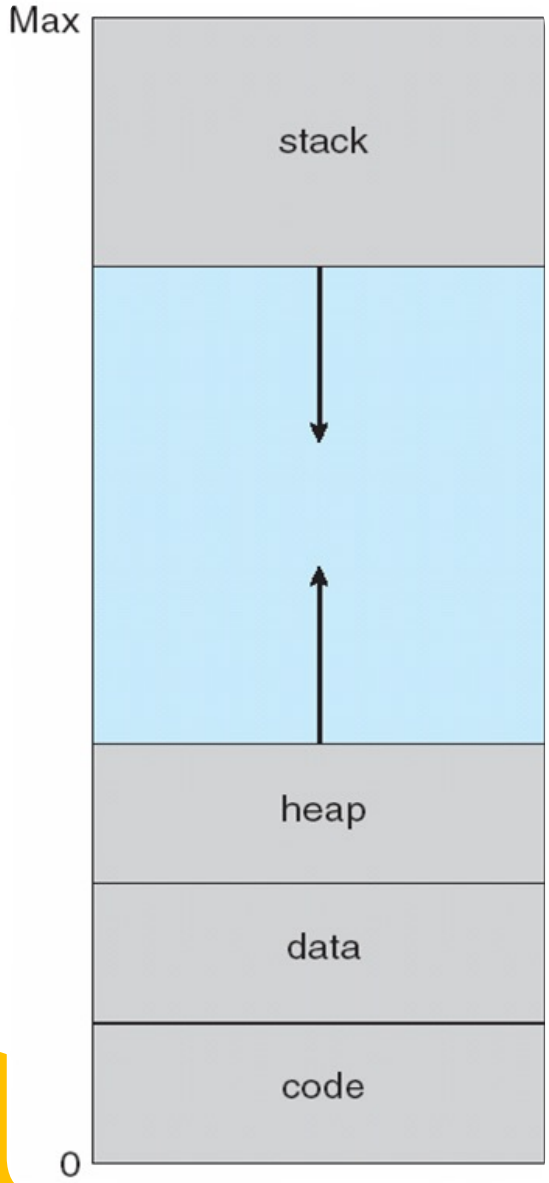
- **Virtual memory** – separation of user logical memory from physical memory.
- **Advantages:**
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Background (Cont.)

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely.
  - For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

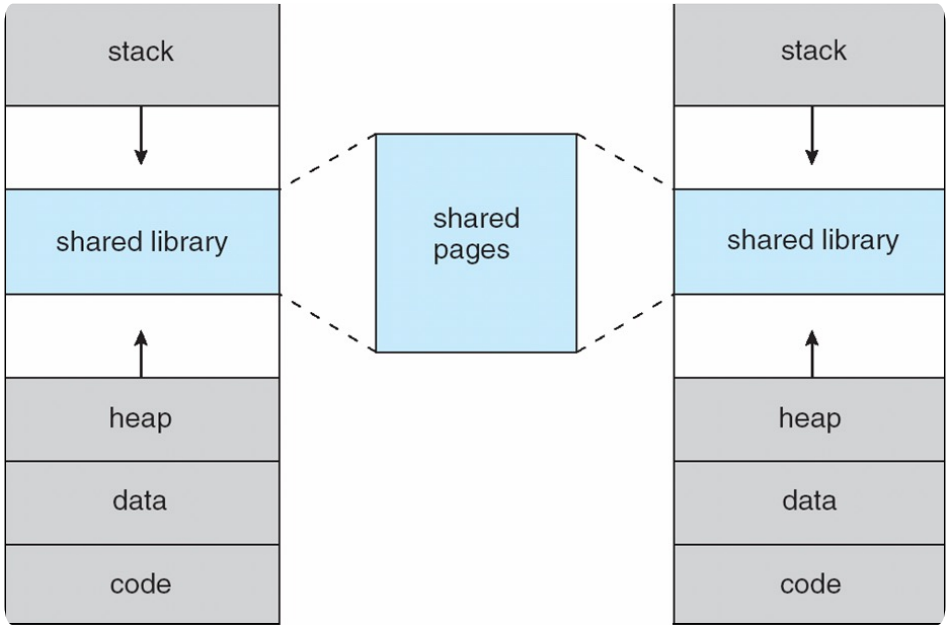
Virtual Memory That is Larger Than Physical Memory





Virtual-address Space

- We allow the **heap** to grow upward in memory as it is used for dynamic memory allocation.
- Similarly, we allow for the **stack** to grow downward in memory through successive function calls.
- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows

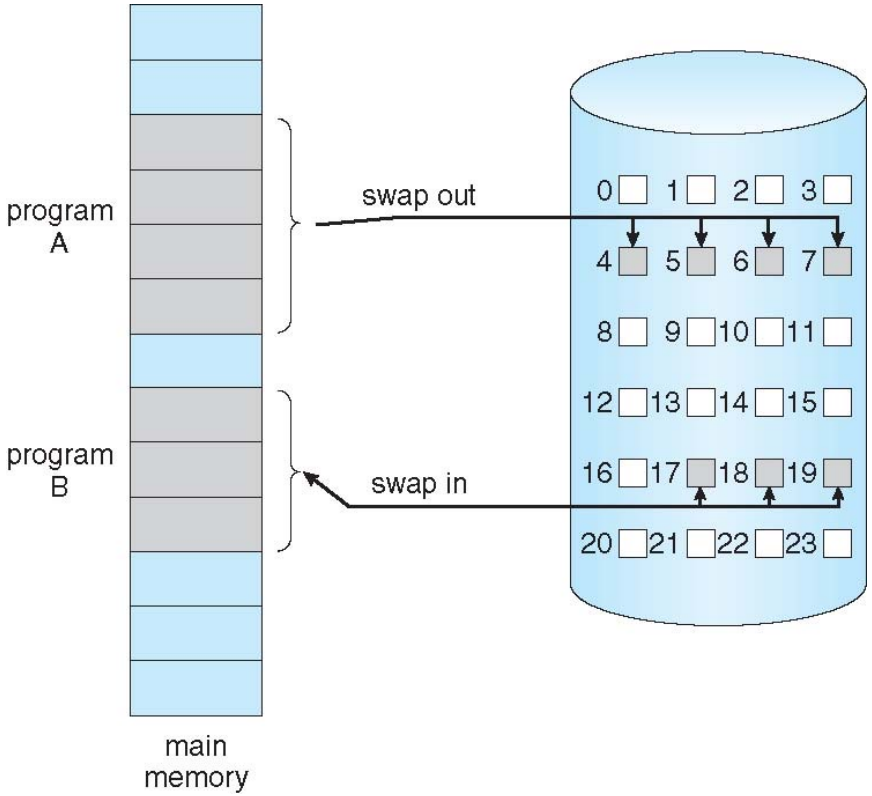


Shared Library Using Virtual Memory

Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.

Demand Paging

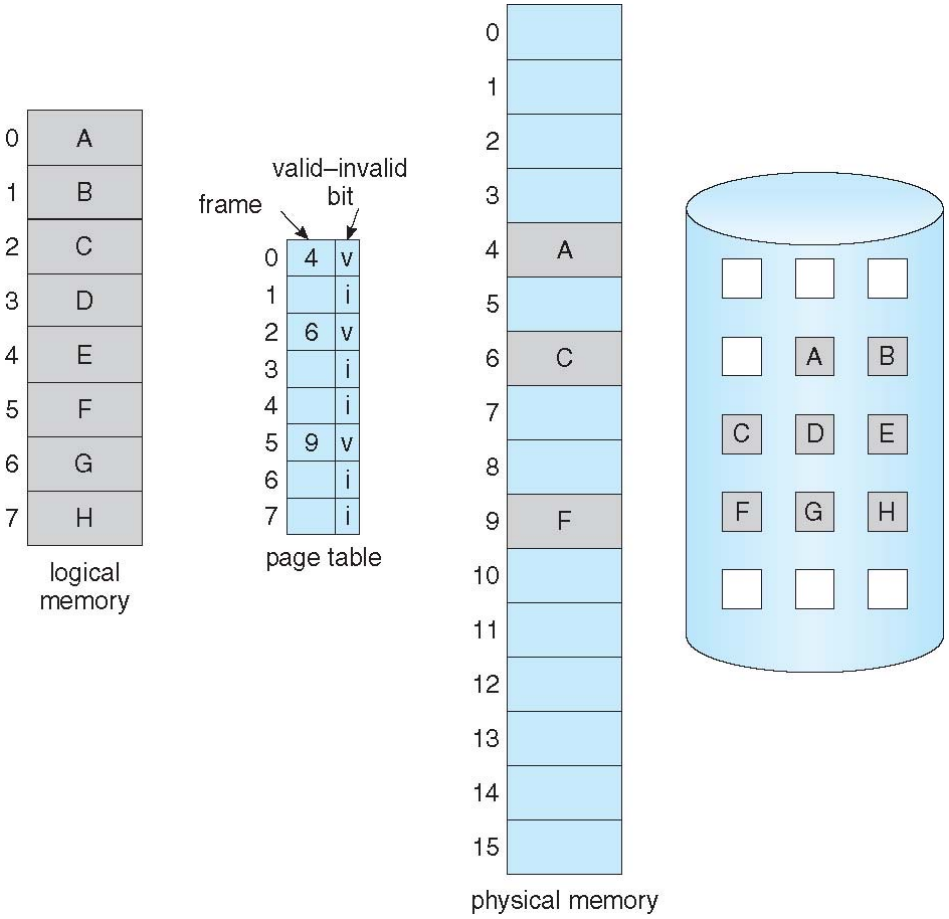
- Bring a page into memory only when it is needed/referenced
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users



Basic Concepts

- Pager brings in only required set of pages into memory
- How to determine that set of pages?
  - Need a new MMU functionality to implement **demand paging**
- If page needed and not memory resident
  - Need to detect and **load the page into memory** from storage

Page Table When Some Pages Are Not in Main Memory

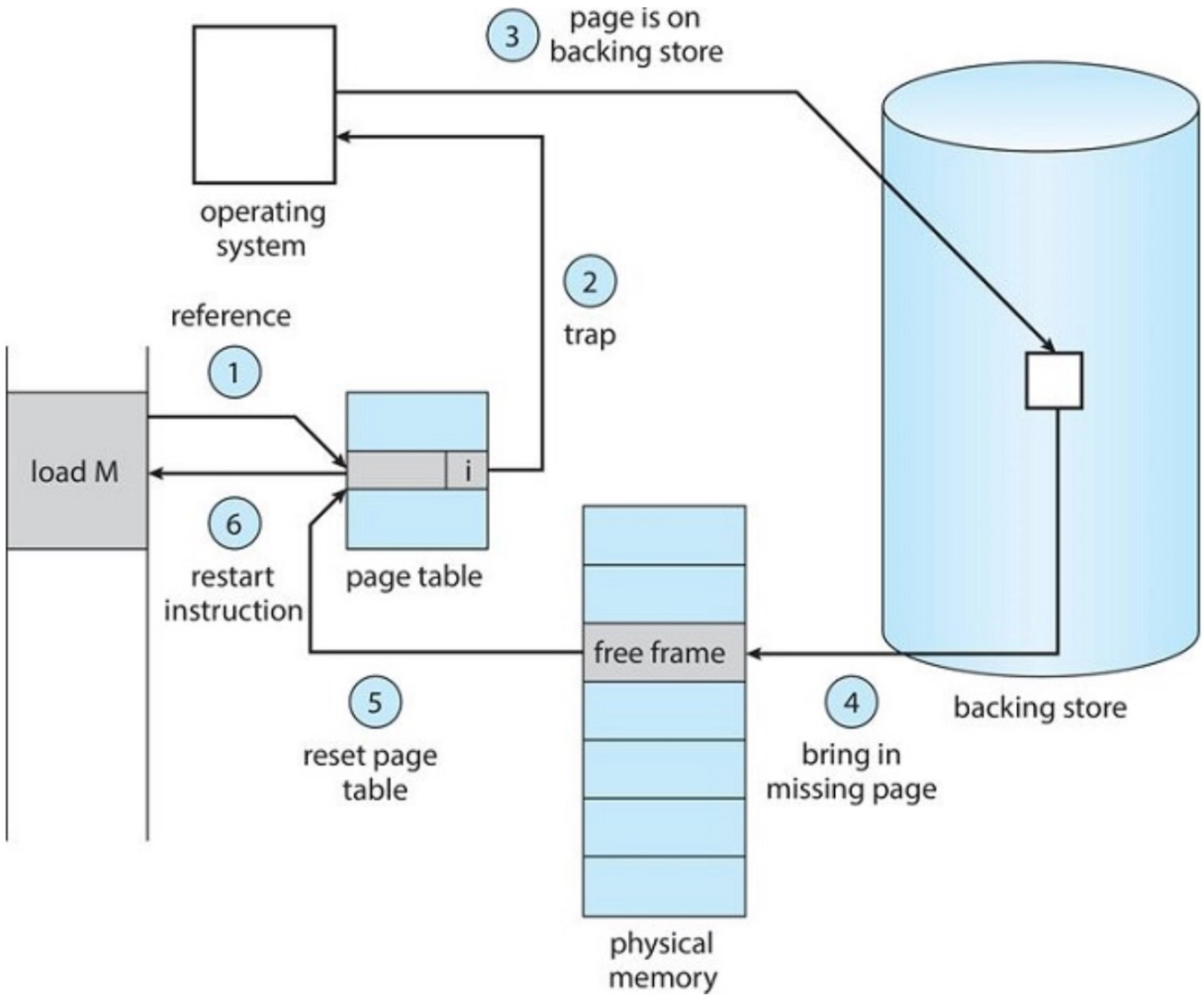


Page Fault

If there is a reference to a page, first reference to that page will **trap** to the operating system:

1. OS looks at an internal table (usually kept with the process control block) to decide:
   - Invalid reference $\Rightarrow$ **abort**
   - Just **not in memory**
2. Find a free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

- **Extreme case** – start a process with ***no*** pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **"Pure demand paging"**

- A given instruction could access **multiple pages** -> **multiple page faults**
  - One page for the instruction and many for data?
  - **Unlikely: due to locality of reference**

Hardware support

- **Page table:** has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory:** holds those pages that are not present in main memory. The section of storage used for this purpose is known as swap space.

Faults

- **Instruction restart:** A page fault may occur at any memory reference.
  - If the page fault occurs on the **instruction fetch**, we can restart by fetching the instruction again.
  - If a page fault occurs while we are **fetching an operand**, we must fetch and decode the instruction again and then fetch the operand.

Free-Frame list

- When a system starts up, all available memory is placed on the free-frame list. As free frames are requested, the size of the free-frame list shrinks.
- At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated.

Figure 10.6 List of free frames.

Performance of demand paging

Assume the memory-access time, denoted *ma*, is 10 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time.

- If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.
- Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time.}$$

Performance of Demand Paging (Cont.)

- **Three major activities:**
  1. Service the interrupt
  2. Read the page
  3. Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

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

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If we want performance degradation < 10 percent

  220 > 200 + 7,999,800 x p

  20 > 7,999,800 x p

  p < .0000025

  < one page fault in every 400,000 memory accesses

Demand Paging – a scenario

- If a process of 10 pages actually uses only half of them, then demand paging saves the I/O necessary to load the 5 pages that are never used.
- We could also increase our degree of multiprogramming by running twice as many processes.
- Thus, if we had 40 frames, we could run 8 processes, rather than the 4 that could run if each required 10 frames (5 of which were never used).

Need For Page Replacement

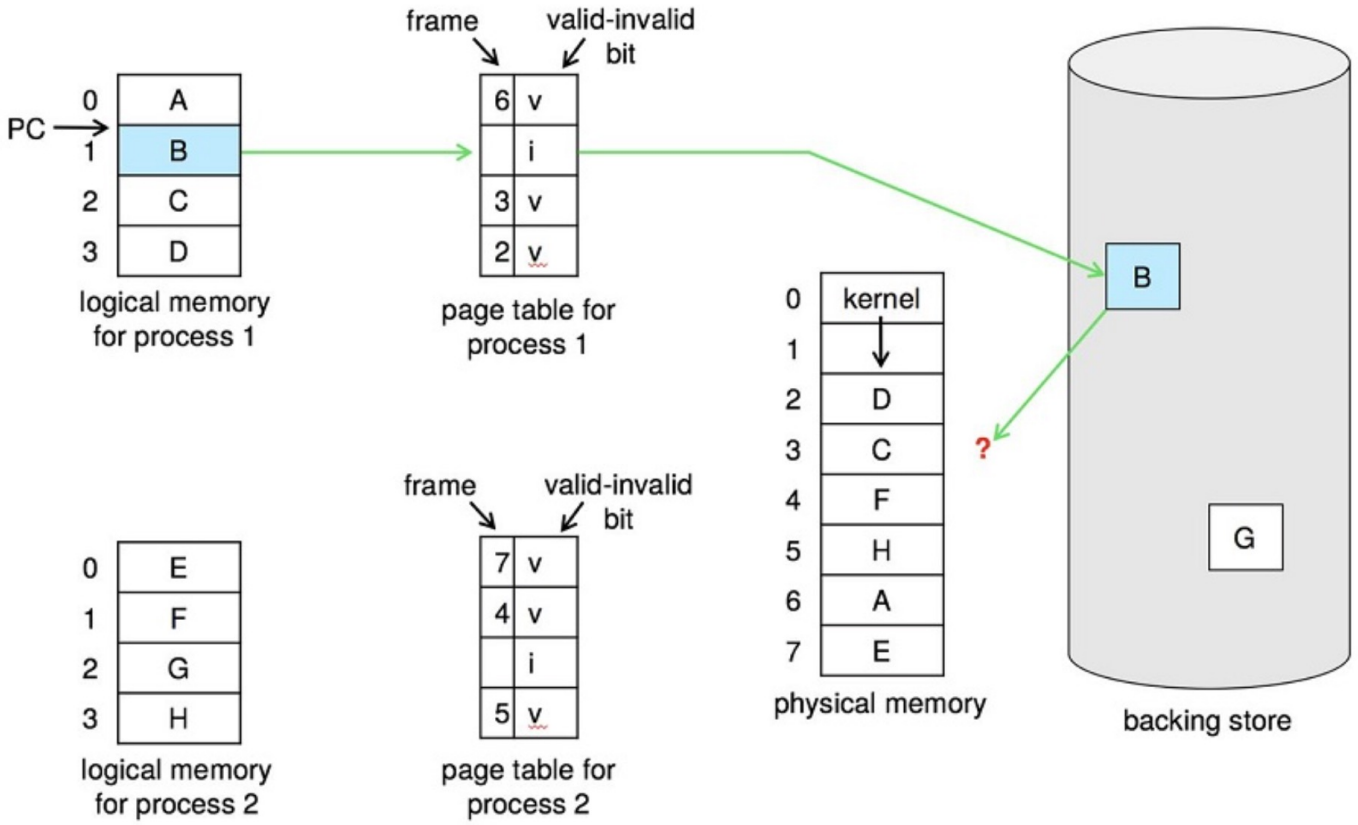


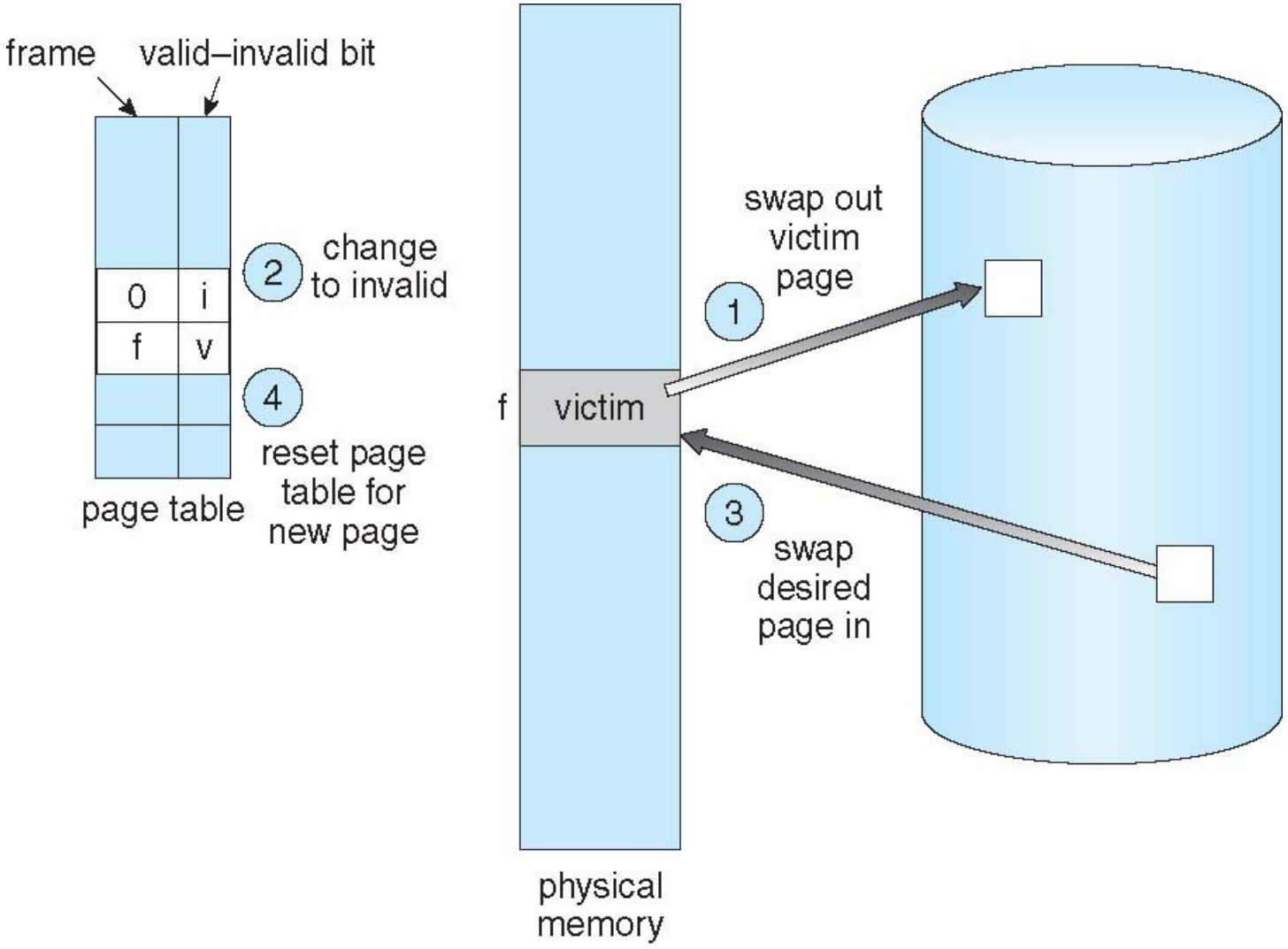
Figure 10.9 Need for page replacement.

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**
   - Write victim frame to disk if **dirty**
3. Bring the desired page into the free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Potentially 2 page transfers for page fault – increasing EAT!

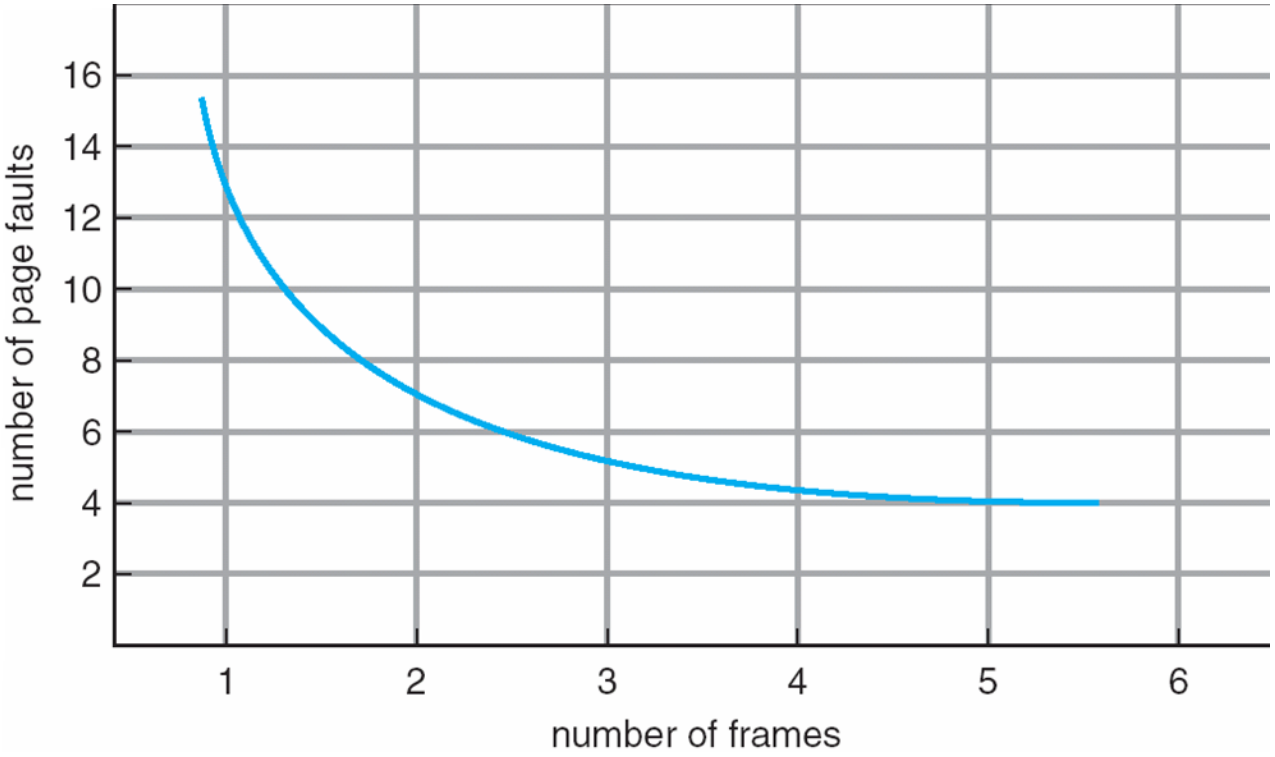
Page Replacement



Page and Frame Replacement Algorithms

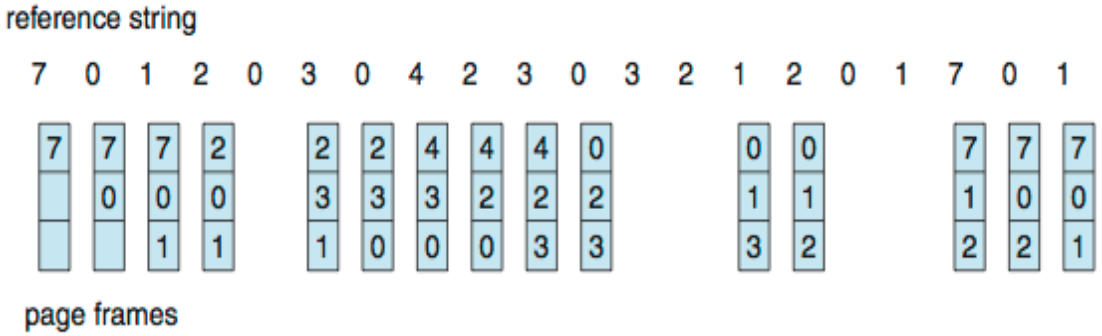
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

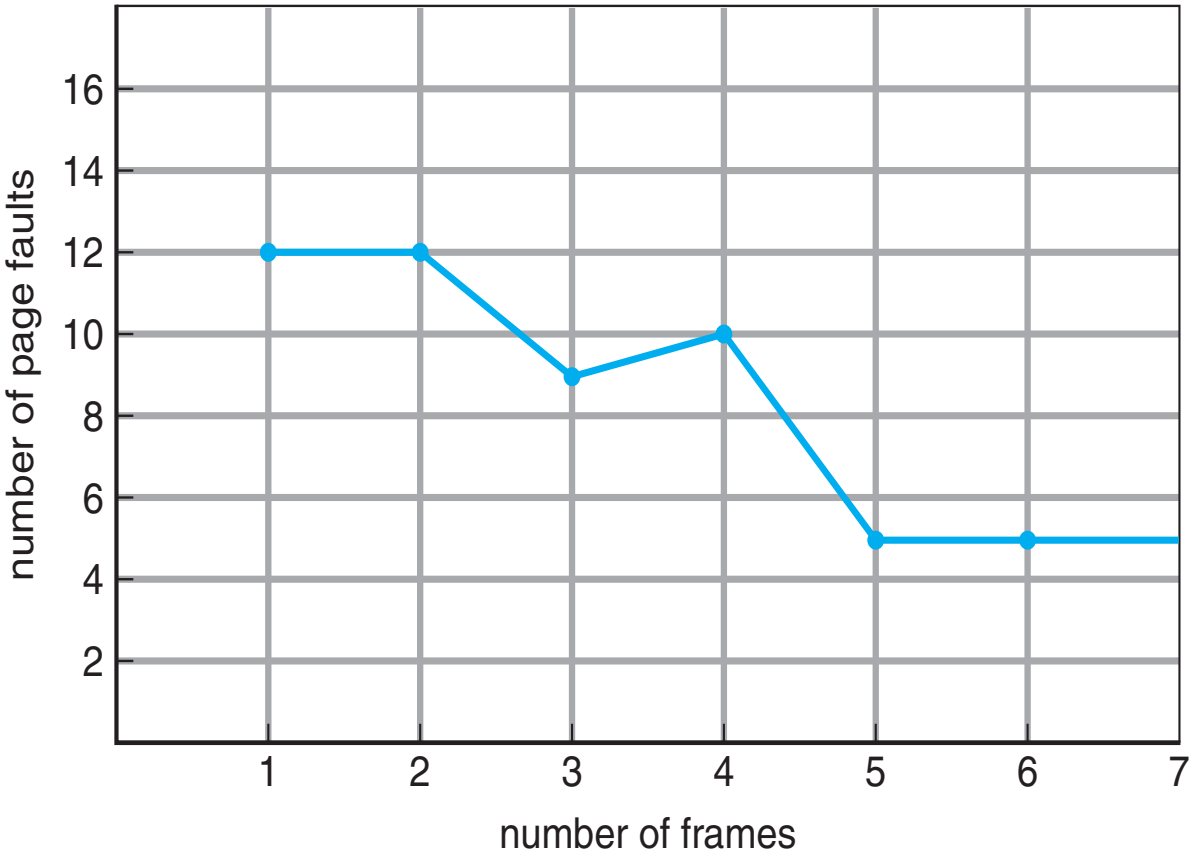
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

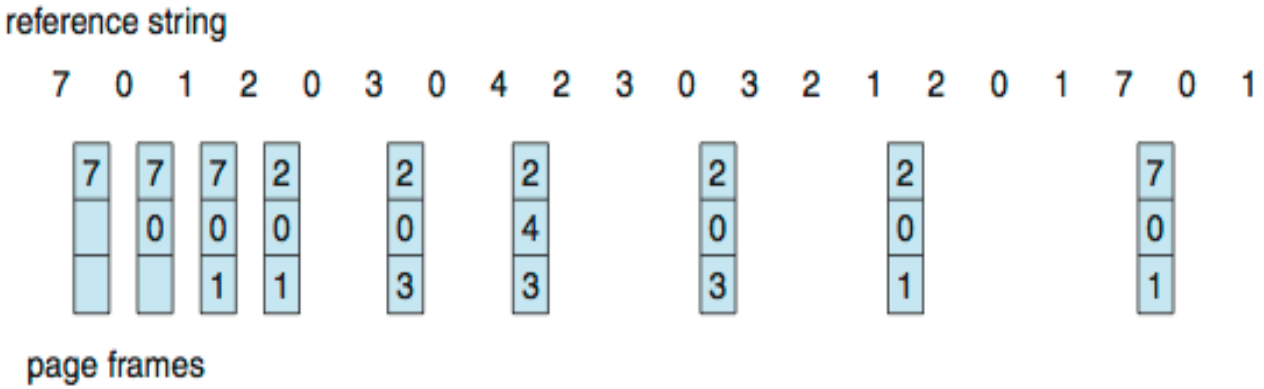
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



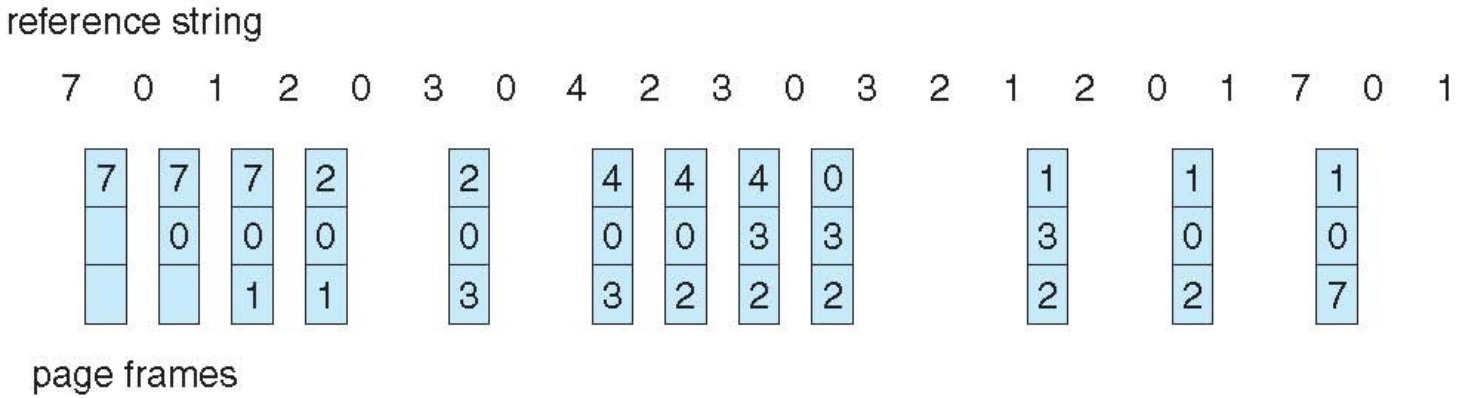
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 page faults is optimal for the example!
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

1. Counter implementation:
   - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
   - When a page needs to be changed, look at the counters to find smallest value
     - Search through table needed
2. Stack implementation:
   - Keep a stack of page numbers in a double link form:
   - Page referenced:
     - move it to the top
     - requires 6 pointers to be changed
   - But each update more expensive
   - No search for replacement

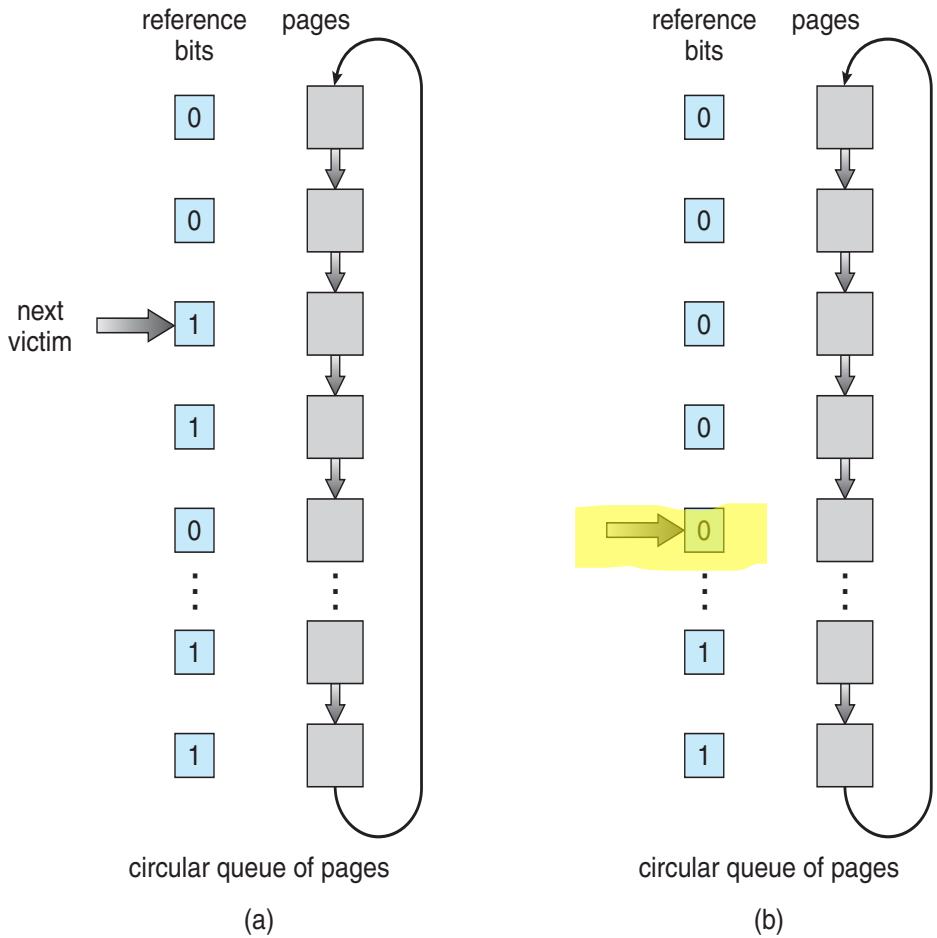
- **LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly**

LRU Approximation Algorithms

Second-chance algorithm

- Generally FIFO, plus hardware-provided reference bit
- **Clock** replacement
- If a page to be replaced has
  - Reference bit = 0 -> replace it
  - Reference bit = 1 then:
    - Give it a second chance, set reference bit 0, leave page in memory
    - A page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).
    - Replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Allocation of Frames

- If we have 93 free frames and two processes, how many frames does each process get?
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- **Equal allocation** –If there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool.
  - Keep some as free frame buffer pool

- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

- Use a **proportional allocation** scheme using priorities rather than size

- If process P_i generates a **page fault**,
  - select for replacement one of its frames
  - select for replacement a frame from a process with **lower priority number**

Global vs. Local Allocation

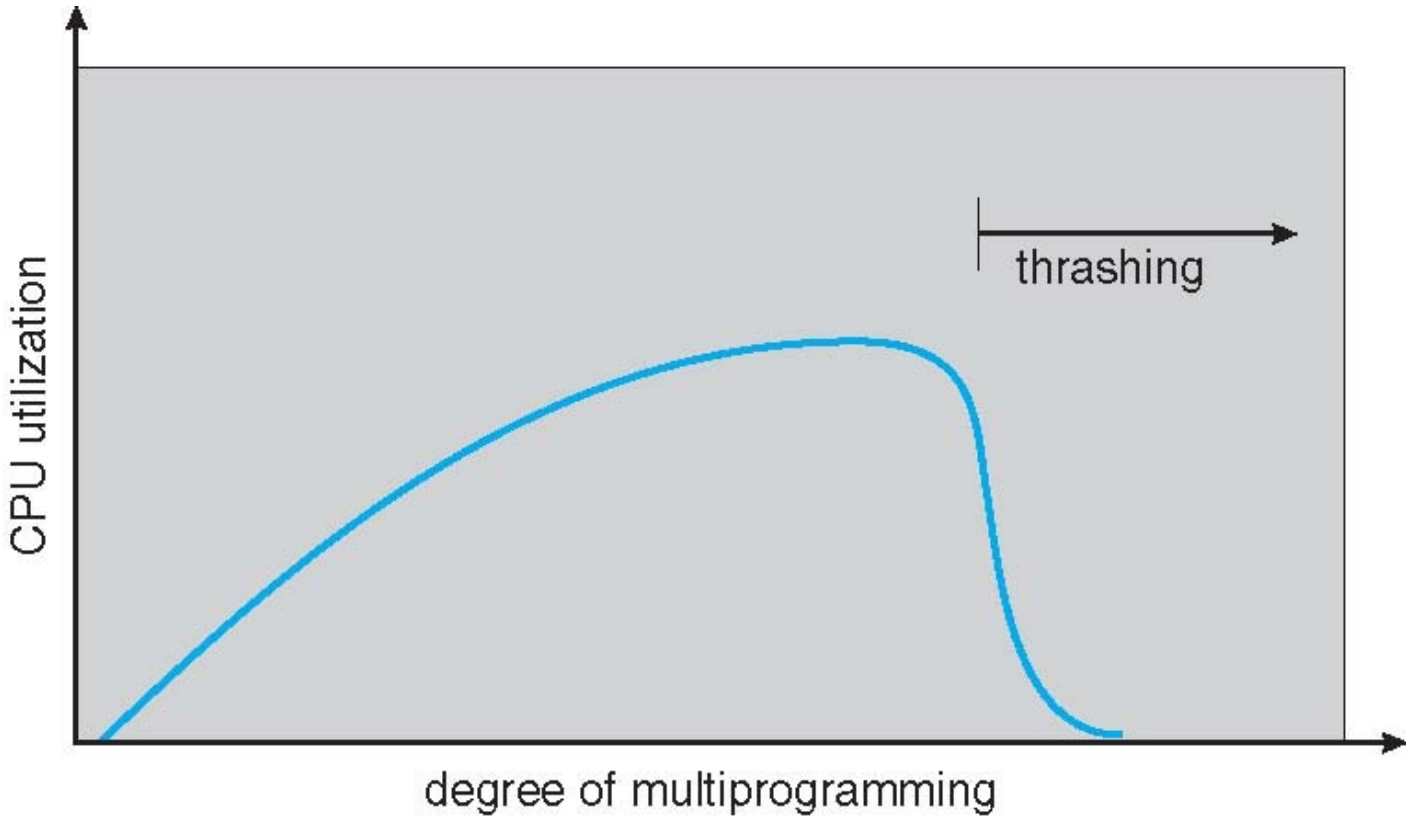
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - **Greater throughput, so more common**

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - Possibly **underutilized** memory

Thrashing

- If a process does not have "**enough**" frames, the page-fault rate is **very high**
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - **Low CPU utilization**
    - Operating system thinking that it needs to increase the degree of multiprogramming
      - Another process added to the system!
- **Thrashing** ≡ a process is busy swapping pages in and out; spending more time paging than executing

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work? → **Locality model**
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size
  - Possible solution: Limit effects by using **local or priority page replacement**

Allocating Kernel Memory

- Treated differently from user memory
- The kernel must use memory conservatively and attempt to **minimize waste due to fragmentation**.
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory **needs to be contiguous**
    - i.e. for device I/O
- Two strategies for managing free memory that is assigned to kernel processes: the “buddy system” and slab allocation.

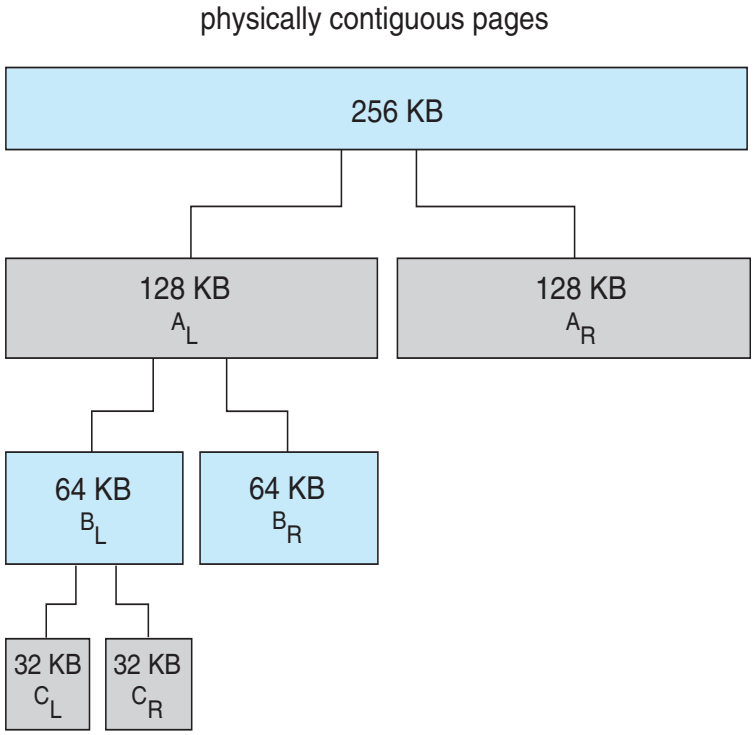
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System

- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

Buddy System Allocator



An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using **coalescing**.

Suffers from internal fragmentation!

Summary

- Benefits: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging: pages are loaded only when they are demanded during program execution.
- Page fault: when a page that is currently not in memory is accessed.
- Page-replacement algorithms: FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global/local page-replacement algorithms.
- Thrashing: when a system spends more time paging than executing.
- Kernel: allocated in contiguous chunks of varying sizes
- All OS: demand paging and a variation of an LRU approximation algorithm.