Chapter 9: Virtual Memory

Edition 9

Operating Systems (CS-220)
Fall 2020, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID
anaum.hamid@nu.edu.pk

Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations

Objectives

To describe the benefits of a virtual memory system

To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

To discuss the principle of the working-set model

To explore how kernel memory is managed

Virtual Memory

- **Virtual memory** is a **memory** management capability of an OS, uses hardware and software to allow a computer to compensate for physical **memory** shortages by temporarily transferring data from random access **memory** (RAM) to disk storage.
- Virtual memory is not RAM (Random access memory) on memory chips.
- Virtual memory is an area of hard drive or other storage space, which OS uses as swap space (overflow) for the physical RAM.
- When the demands of the software and data exceed the physical RAM size, the operating system copies to hard disk blocks of data from RAM that have been seldom used and releases that RAM for use by the current process.
- When the block (or Page) is required again, the OS makes room in RAM by copying another block to hard drive and copying in the data from the first block.

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- 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

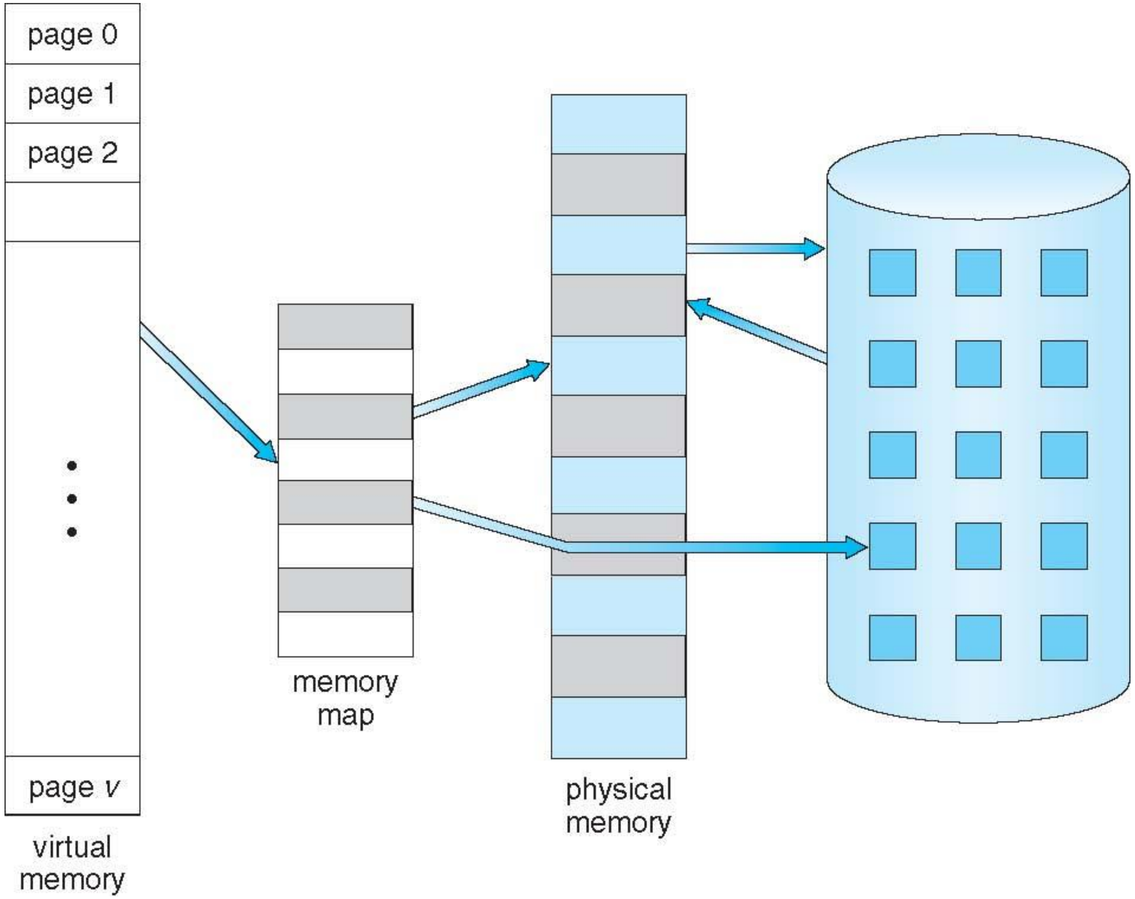
Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
  - 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.)

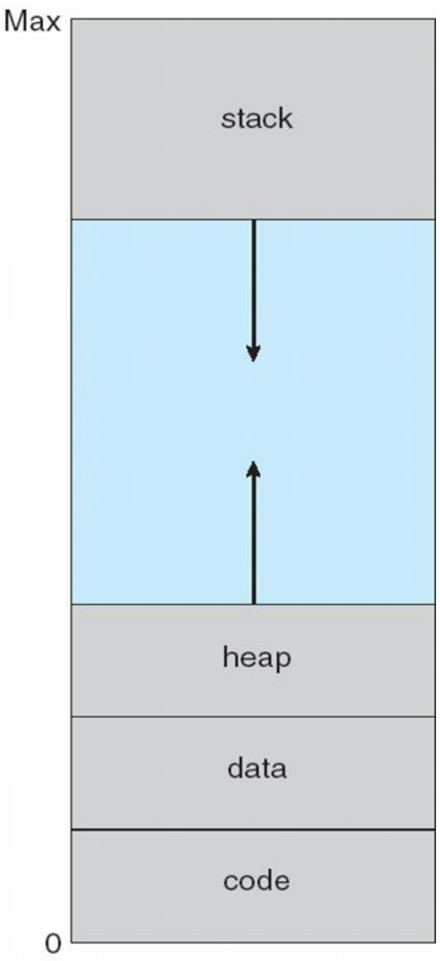
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

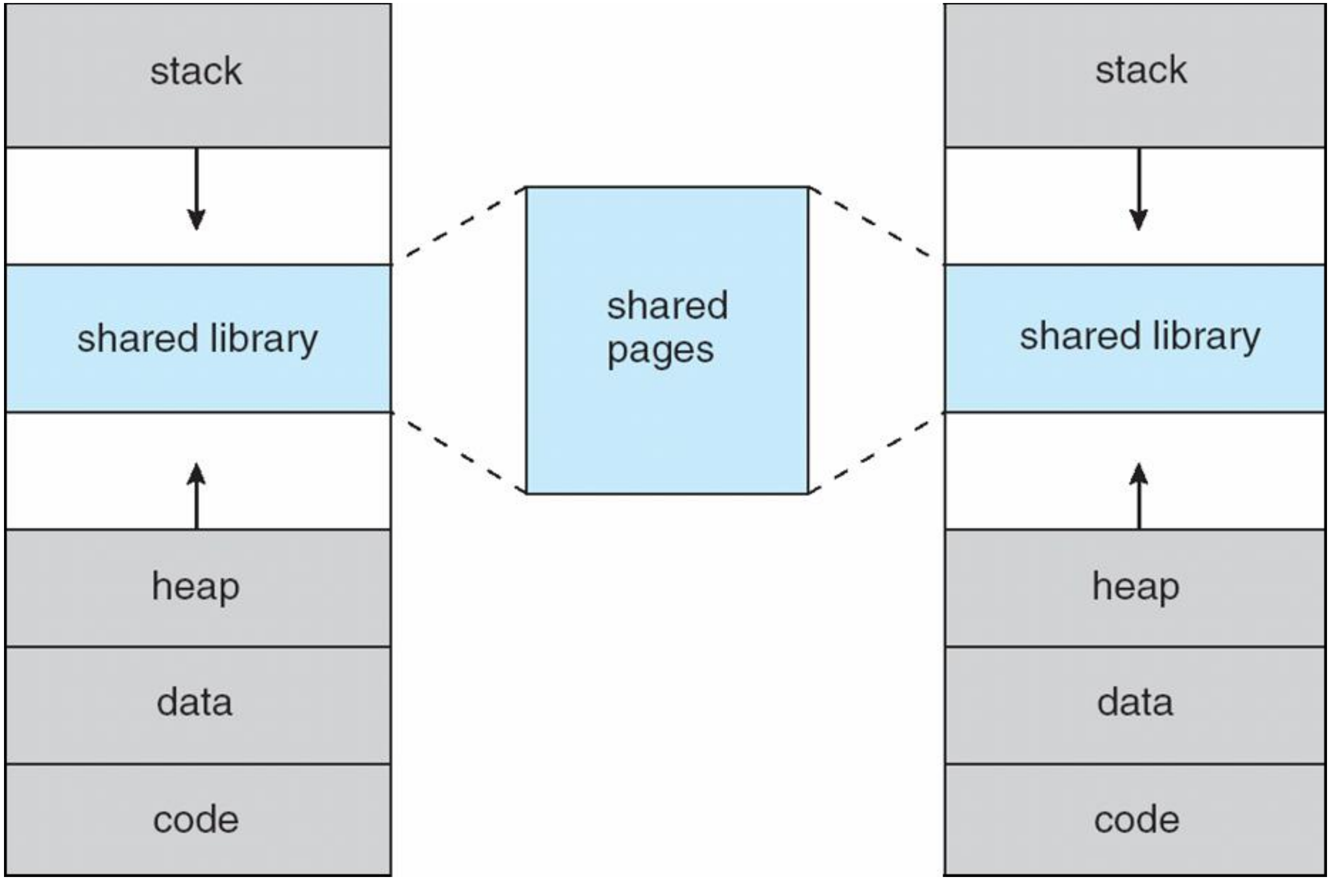


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

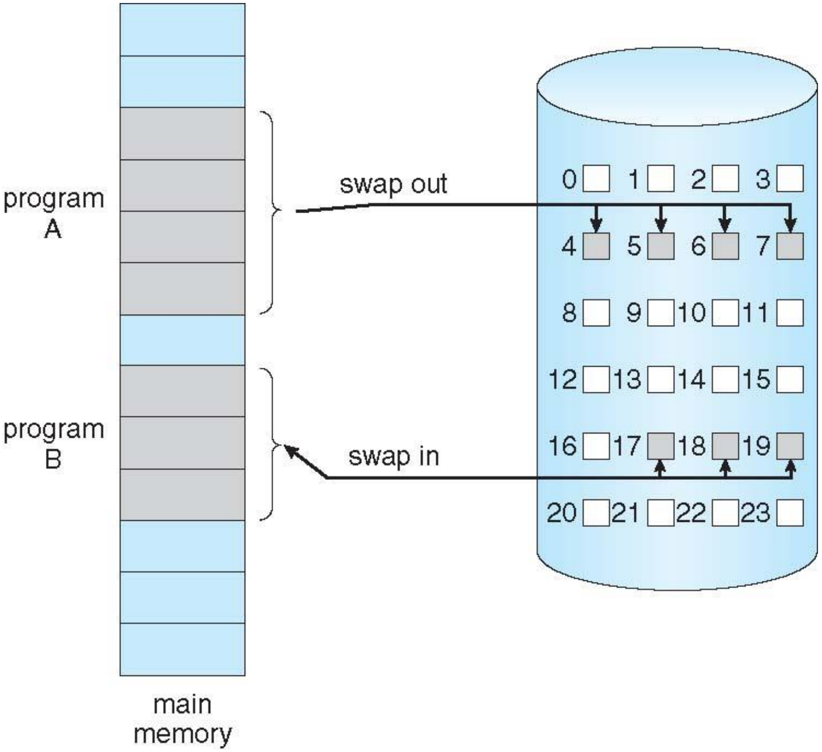


Shared Library Using Virtual Memory



Demand Paging

- To load pages only as they are needed. This technique is known as demand paging.
- Or bring a page into memory only when it is needed
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- Advantages:
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (as diagram on right shows)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory



Demand Paging

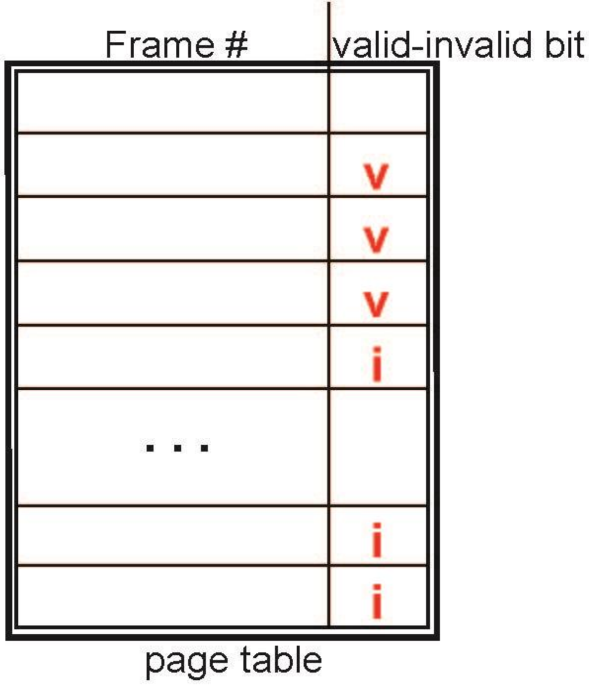
- A lazy swapper never swaps a page into memory unless that page will be needed.
- In the context of a demand-paging system, use of the term “swapper” is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use “pager,” rather than “swapper,” in conection with demand paging

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

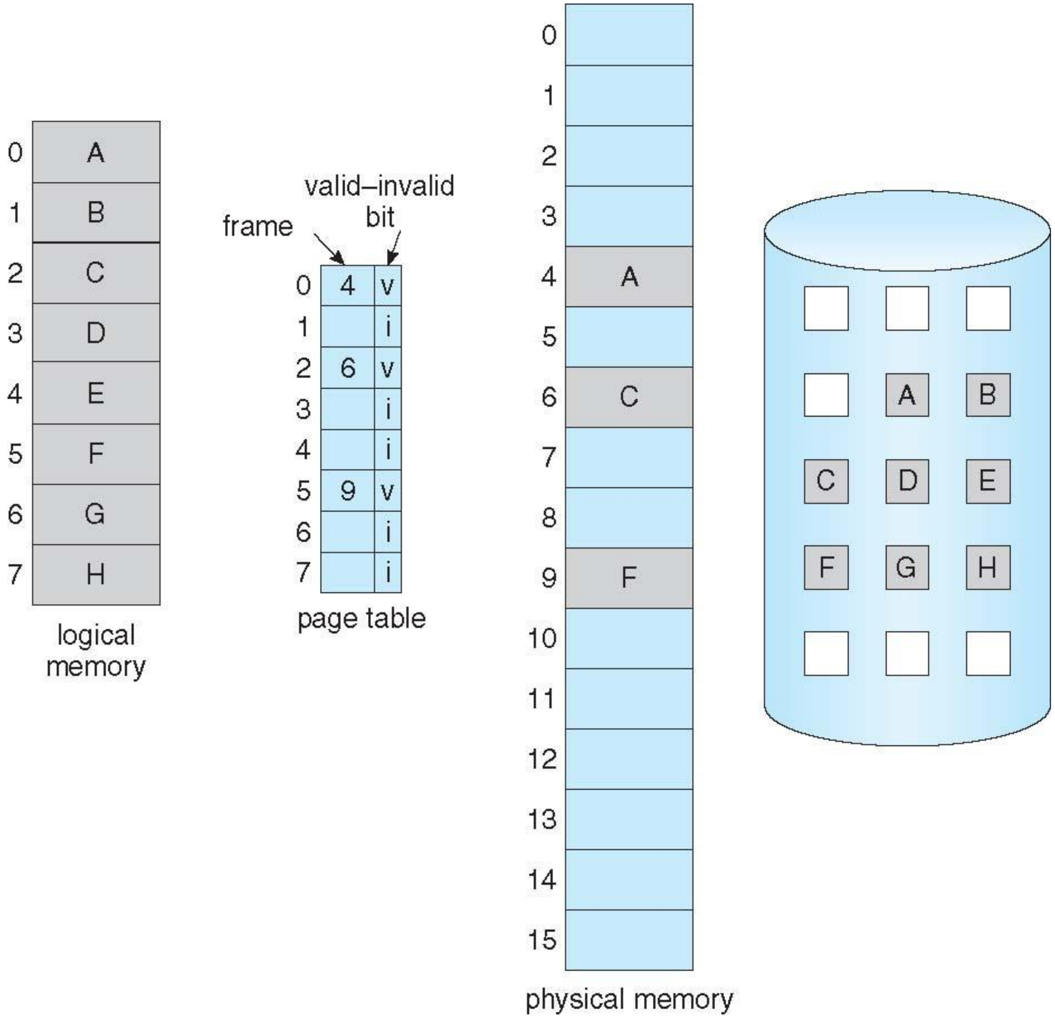


Frame #	valid-invalid bit
	v
	v
	v
	i
. . .	
	i
	i

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

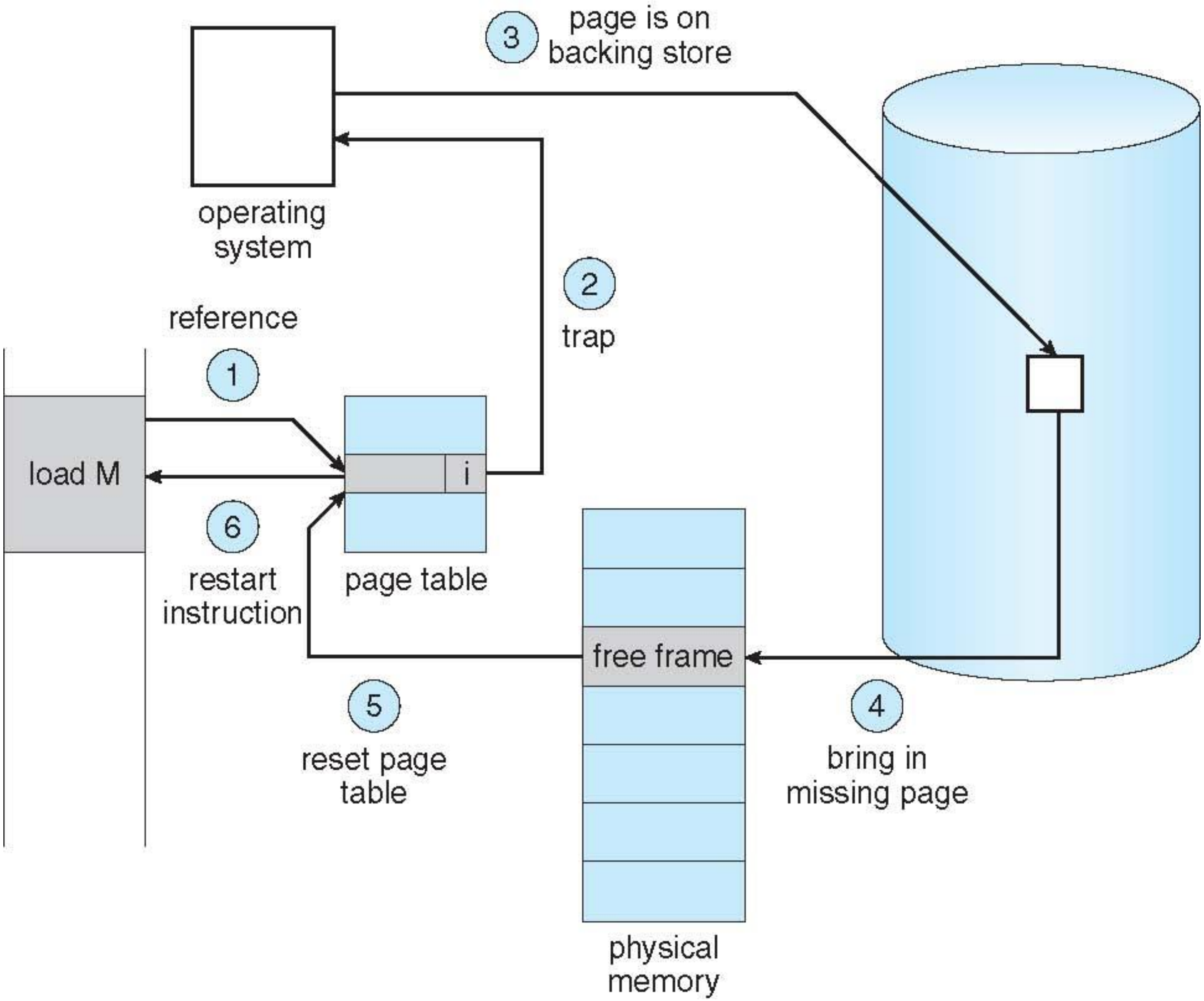
Page Fault

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

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find 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 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**
- Actually,a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:
- 1. Fetch and decode the instruction (ADD).
- 2. Fetch A.
- 3. Fetch B.
- 4. Add A and B.
- 5. Store the sum in C.
- If we have a fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs

Performance of Demand Paging

- Stages in Demand Paging (worse case)

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
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- Three major activities
  1. Service the interrupt – careful coding means just several hundred instructions needed
  2. Read the page – lots of time
  3. Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – *p*) x ma + *p* (page fault time)

page fault time = fault overhead+ swap page out+ swap page in

Page fault (EAT calculation)

- Memory access time = ma = 200 nanoseconds
- Average page-fault service time = **page fault time =** 8 milliseconds
- If one access out of 1,000 causes a page fault, then

  p = 1/1000 = 0.001
- **EAT = (1 – *p*) x ma + *p* (page fault time)**
- EAT = (1 – 0.001) x 200 + 0.001 x (8e -6)
- EAT = 8199.8 nanoseconds
- EAT = 8.199 microseconds

Millisecond = 10 e-3

Microsecond = 10e-6

Nanoseconds = 10e-9

Demand Paging Optimizations

1. Swap space I/O faster than file system I/O even if on the same device
    1. Swap allocated in larger chunks; less management needed than file system. No file lookups and indirect allocation methods
2. Copy entire process image to swap space at process load time
    1. Then page in and out of swap space
    2. Used in older BSD Unix
3. Demand page in from program binary on disk, Demand pages for such files are brought directly from the file system.
    1. Used in Solaris and current BSD
    2. Still need to write to swap space
        1. Pages not associated with a file (like stack and heap) – **anonymous memory**
        2. Pages modified in memory but not yet written back to the file system
4. Mobile systems
    1. Typically, don't support swapping
    2. Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

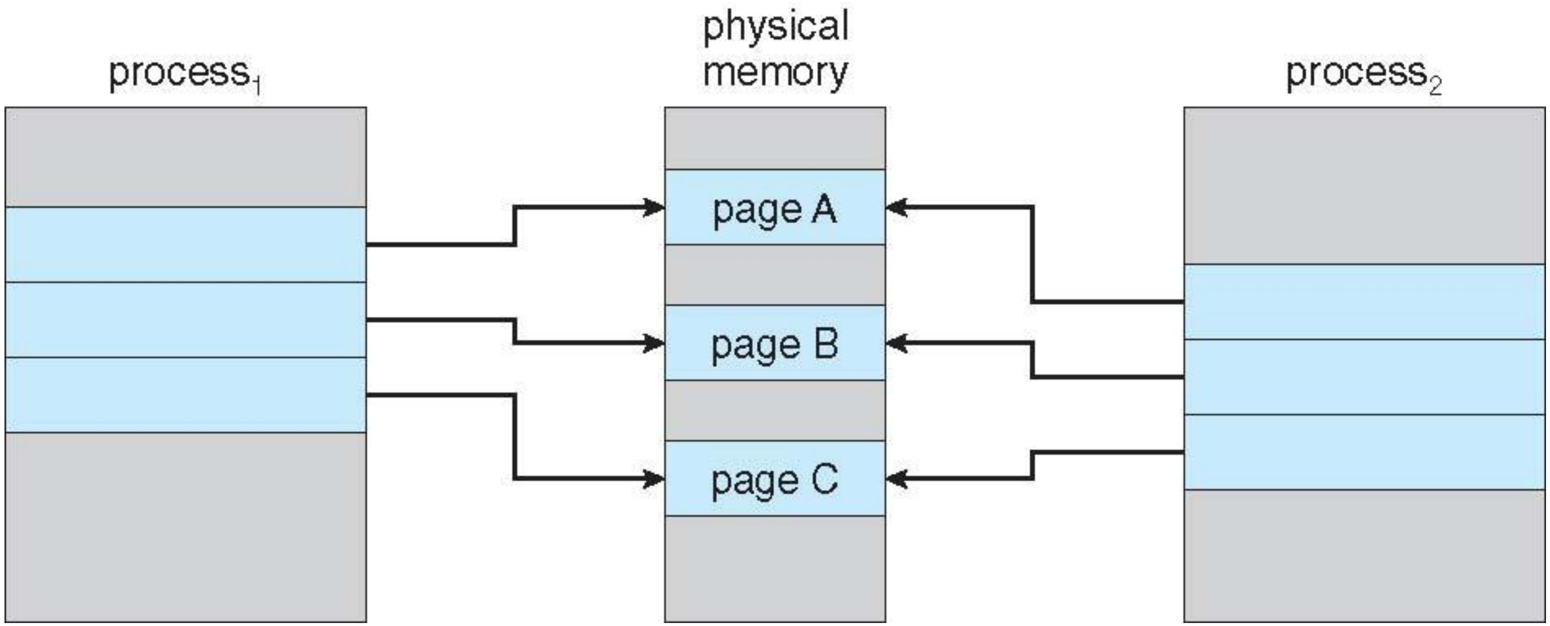
- **Copy-on-Write** (COW) technique allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied.

- COW allows more efficient process creation as only modified pages are copied.

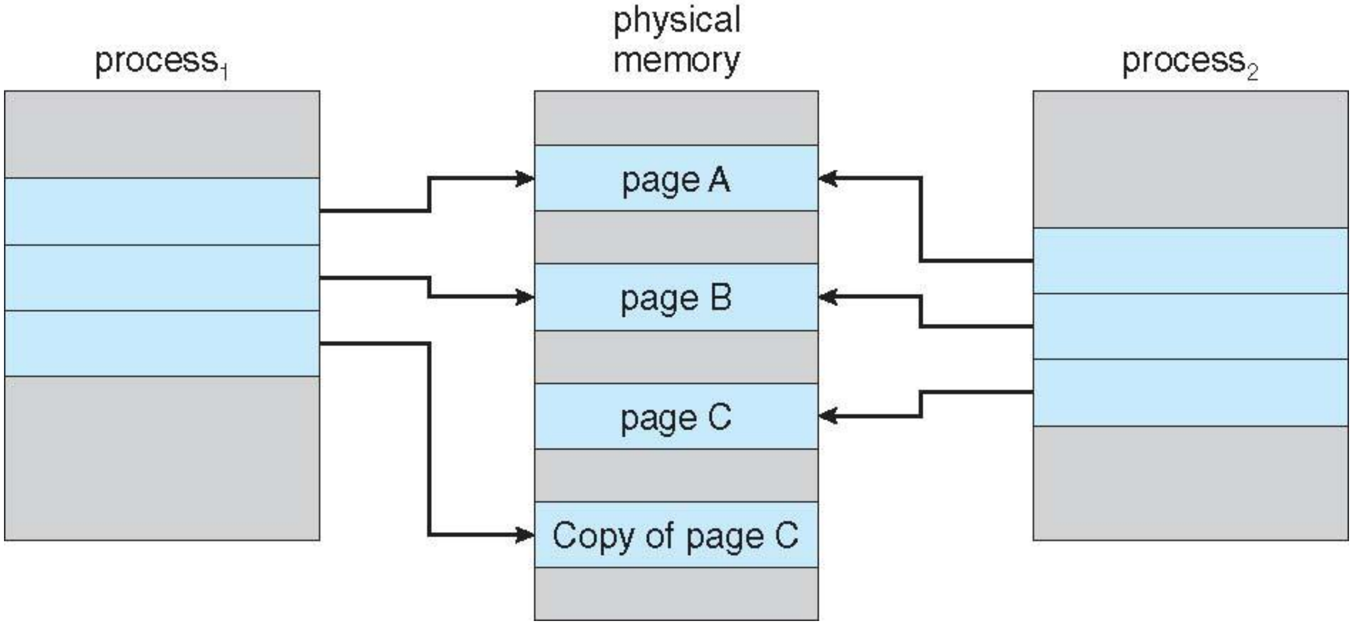
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Operating systems typically allocate these pages using a technique known as zero-fill-on-demand. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

- `vfork()` (for virtual memory fork) variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

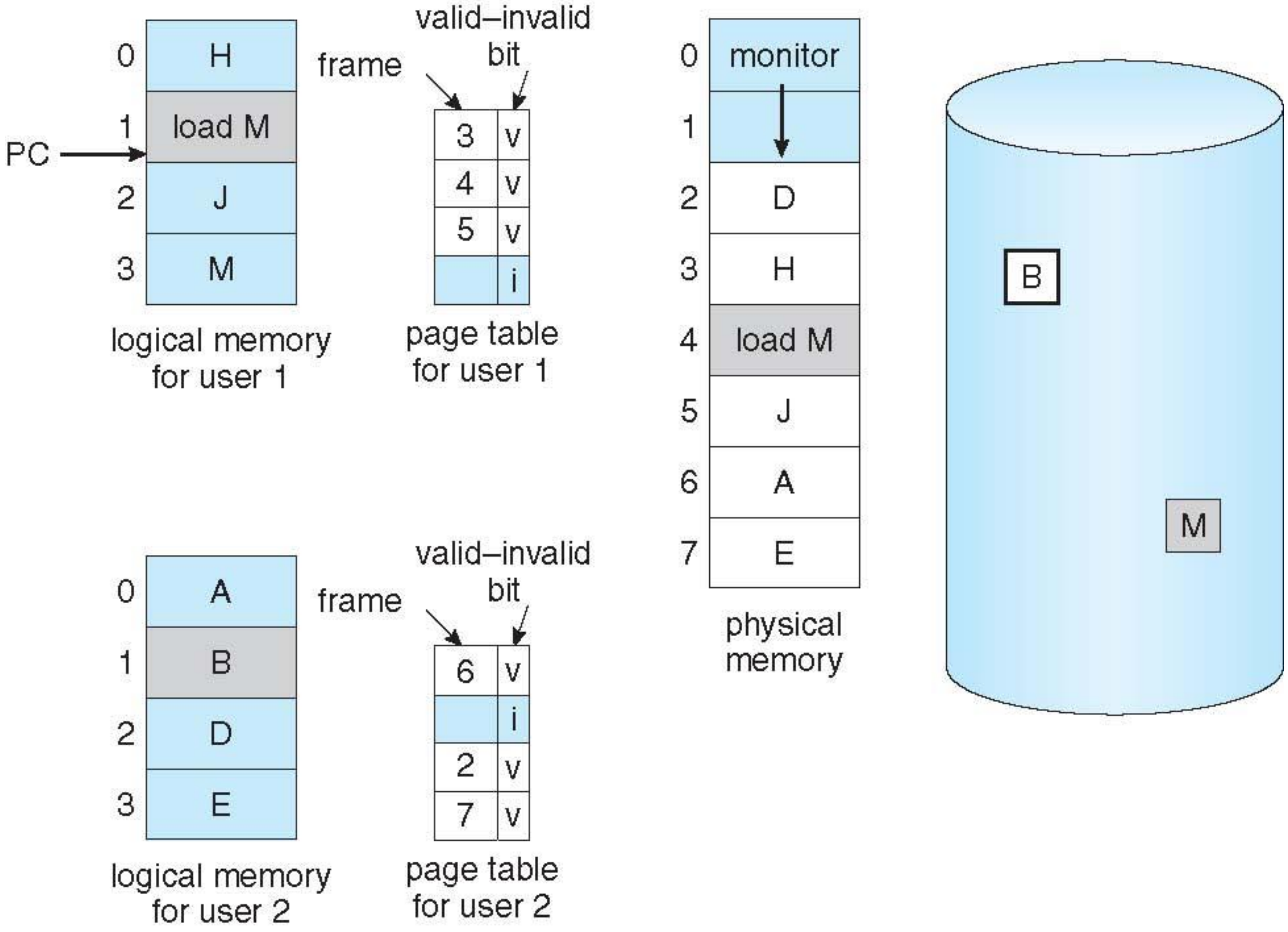
Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Need For Page Replacement



What Happens if There is no Free Frame?

- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times.

Page Replacement

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

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk

Over- Allocation

- If a process of ten pages uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used.
- We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).
- If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare.
- It is possible that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

Modify (dirty) bit

- if no frames are free, two-page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- this overhead by using a modify bit (or dirty bit).
- In this scheme, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware page has been modified.
- When we select a page for replacement, If the bit is set, it means that the page has been modified since it was read in from the disk (swap space). In this case, we must write the page to the disk.
- If the modify bit is not set, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

Basic Page Replacement Steps

1. Find the location of the desired page on disk

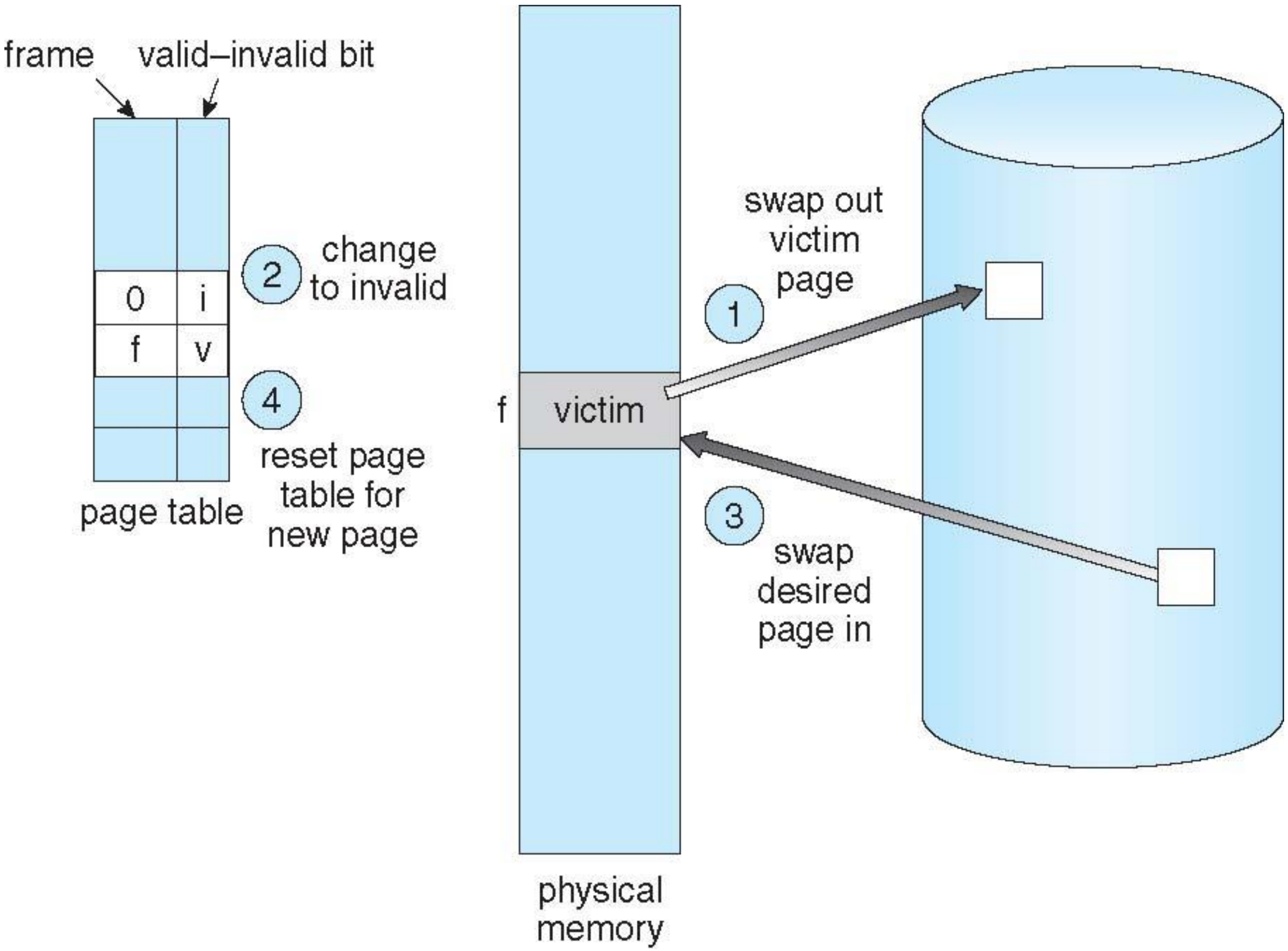
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2-page transfers for page fault – increasing EAT

Page Replacement



Page Replacement

- Page Replacement carried out by:

1. Page-replacement algorithm
2. Frame-allocation algorithm

- **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

Page Replacement Algorithms

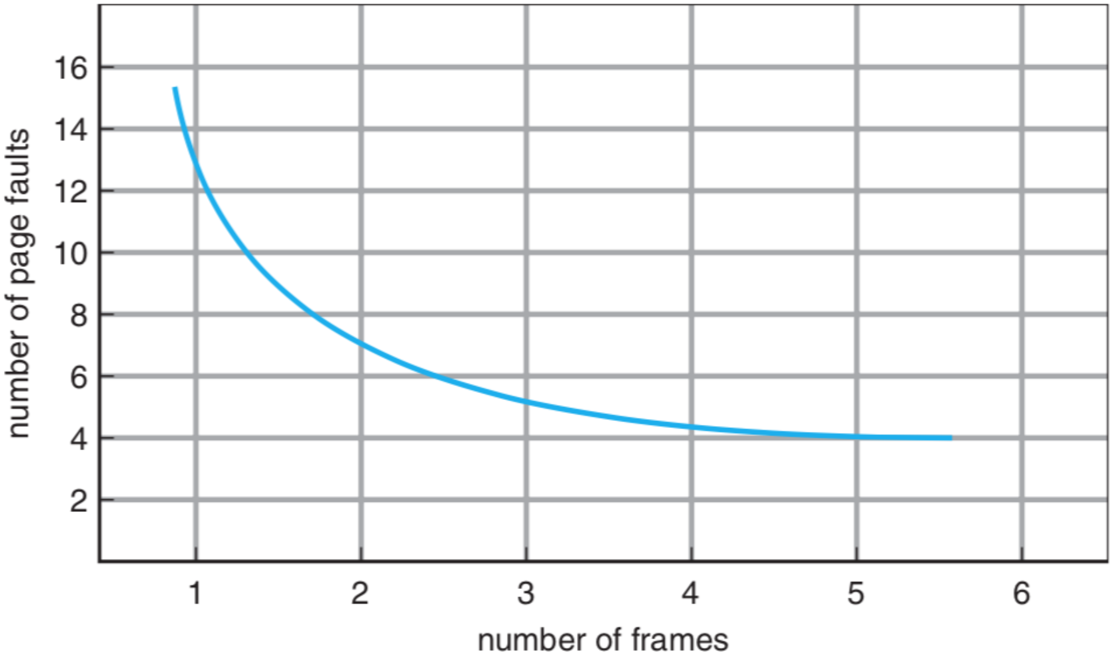
- Basic page replacement algorithms are:

1. FIFO
2. Optimal Solution
3. LRU Algorithm
4. LRU –Approximation

- 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
- In all our examples, the **reference string** of referenced page numbers is

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

ratio	6	0	1	2	0	3	0	5	2	3	0	3	0	3	2	1	2	0	1	6	0	1
f1																						
f2																						
f3																						

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

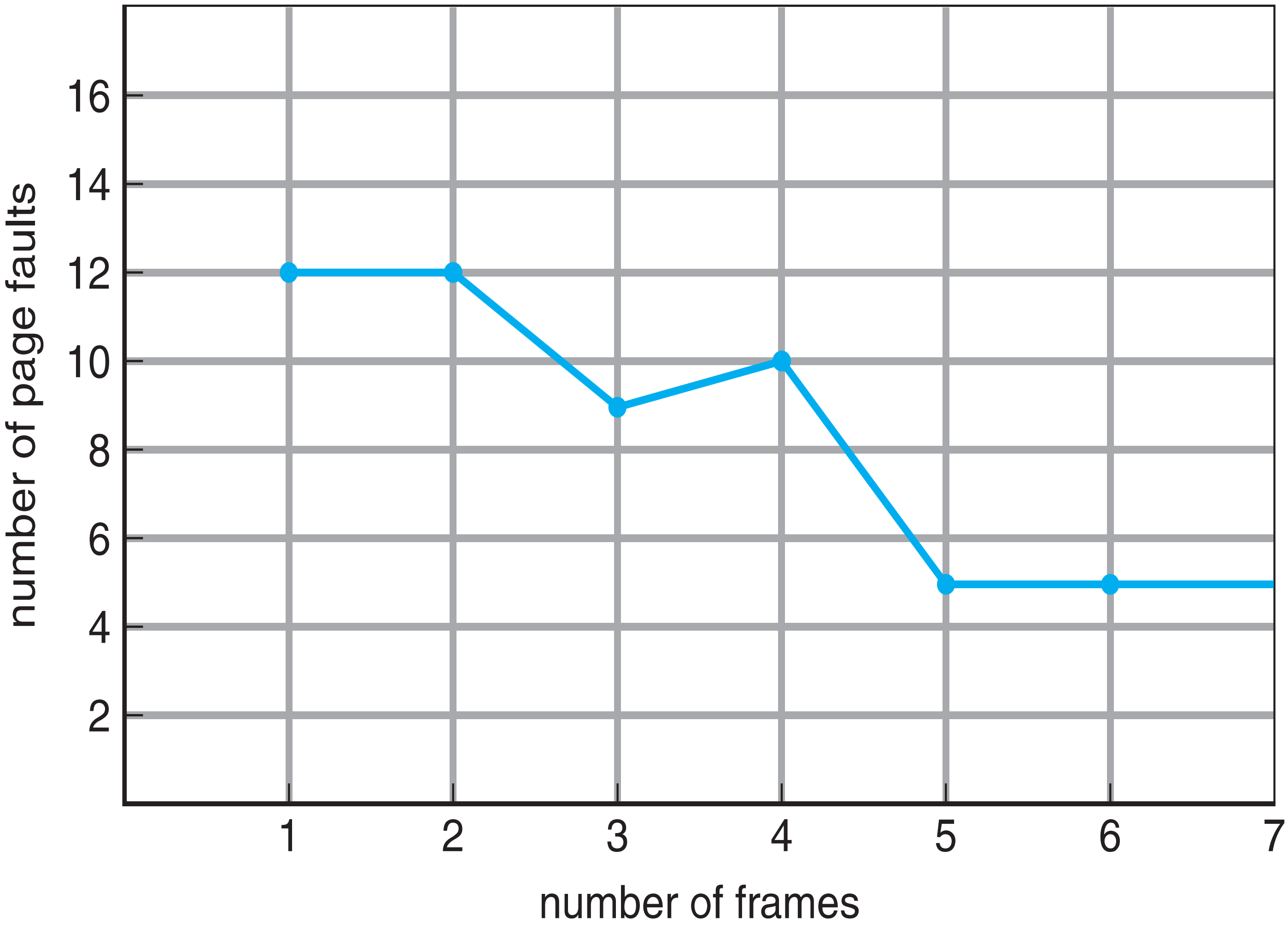
Hit ratio = 7/22*100
=31.81
Fault = 15/22*100 =
68.18

ratio	6	0	1	2	0	3	0	5	2	3	0	3	0	3	2	1	2	0	1	6	0	1
f1	6	6	6	2	2	2	2	5	5	5	0	0	0	0	0	0	0	0	0	6	6	6
f2		0	0	0	0	3	3	3	2	2	2	2	2	2	2	1	1	1	1	1	0	0
f3			1	1	1	1	0	0	0	3	3	3	3	3	3	3	2	2	2	2	2	1
	f	f	f	f	H	f	f	f	f	f	f	H	H	H	H	f	f	H	H	f	f	f

Belady’s Anomaly

- **Belady’s Anomaly**: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady’s anomaly was discovered as a result.

FIFO Illustrating Belady's Anomaly



1,2,3,4,1,2,5,1,2,3,4,5

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

Hit ratio = 12/22*100 =
Fault = 10/22*100 =

ratio	6	0	1	2	0	3	0	5	2	3	0	3	0	3	2	1	2	0	1	6	0	1
f1	6	6	6	6	6	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2
f2		0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6
f3			1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
f4				2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1
	f	f	f	f	H	f	H	f	H	H	f	H	H	H	H	f	f	H	H	f	H	H

Optimal Page Replacement Algorithm

- To Overcome Belady’s anomaly, **optimal page-replacement algorithm introduced**—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady’s anomaly.
- Such an algorithm does exist and has been called OPT or MIN. It is simply this:
- ➢ Replace the page that will not be used for the longest period of time.
- Use of this page-replacement algorithm guarantees the lowest possible page- fault rate for a fixed number of frames.

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

ratio	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1																						
f2																						
f3																						

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Hit ratio = 13/22*100 =
Fault ratio = 9/22*100 =

ratio	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
f2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0
f3			1	1	1	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
	f	f	f	f	H	f	H	f	H	H	f	H	H	H	H	f	H	H	H	f	H	H

Least Recently Used (LRU) Algorithm

- LRU replacement associates with each page the time of that page’s last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- LRU strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

ratio	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1																						
f2																						
f3																						

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Hit ratio = 10/22*100 =54.54
Fault ratio = 12/22*100 =45.45

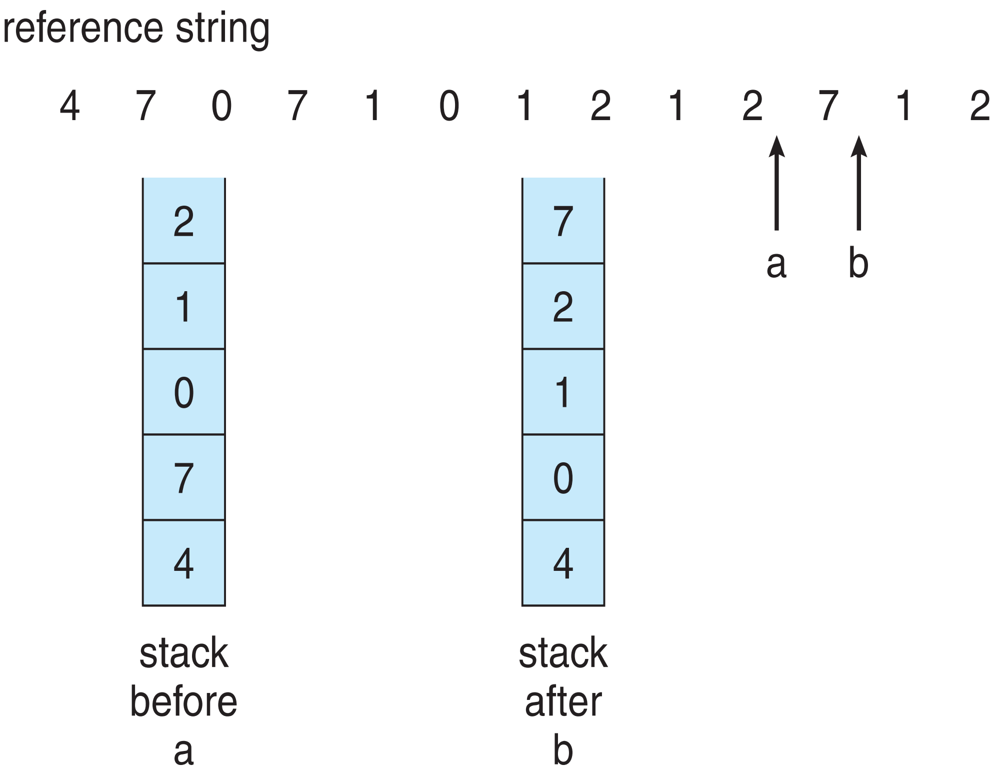
ratio	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	1	1	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	0	0	0	0	0
f3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	f	f	f	f	H	f	H	f	f	f	f	H	H	H	H	f	H	f	H	f	H	H

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter. The clock is incremented for every memory reference.
- Stack implementation
  - Another approach to implementing LRU replacement is to keep a stack of page numbers.
  - Whenever a page is referenced, it is removed from the stack and put on the top. the most recently used page is always at the top of the stack and the least recently used page is always at the bottom .
  - Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer.

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

LRU Implementation using Stack



reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

a b

2
1
0
7
4

stack before a

7
2
1
0
4

stack after b

LRU Approximation Algorithms

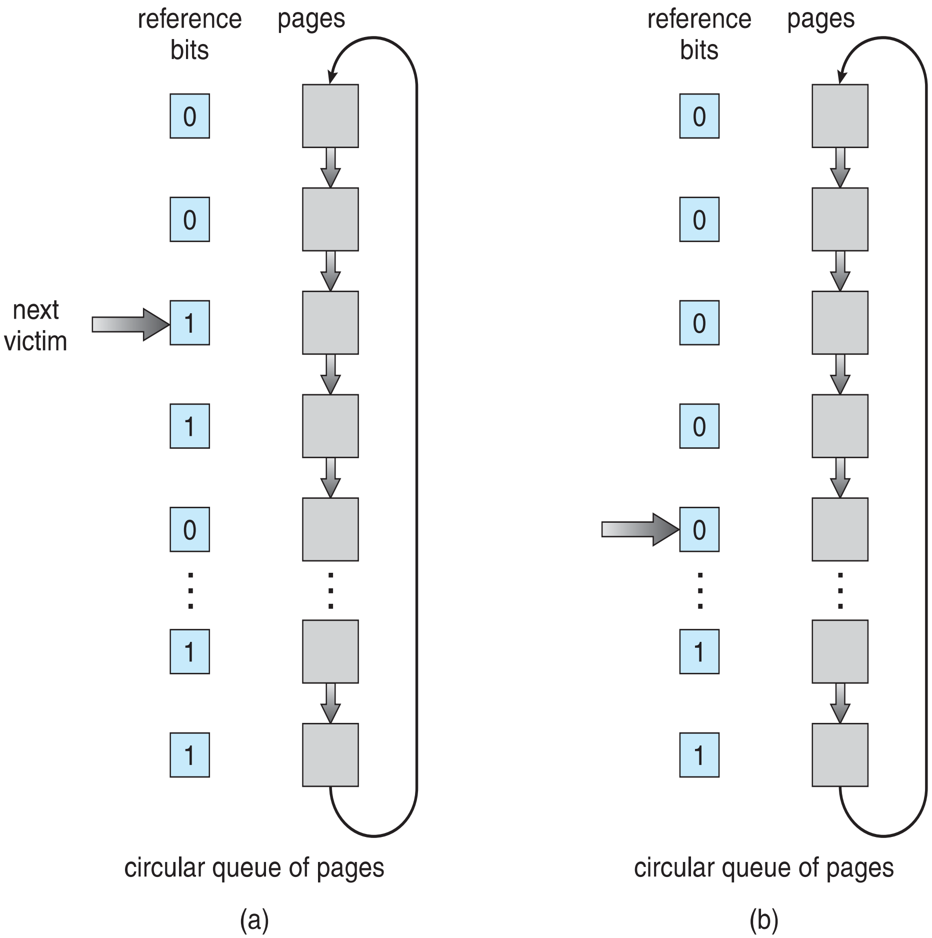
LRU needs special hardware and still slow. There are three LRU Approximation Algorithms:

1. Additional reference bit Algorithm
2. Second Chance Algorithm
3. Enhanced Second Chance Algorithm

LRU Approximation Algorithms

- **Reference bit Algorithm**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally, FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit.
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

Enhanced Second-Chance Algorithm

- Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm.

- Examine the class to which that page belongs and replace the first page encountered in the lowest nonempty class.

- scan the circular queue several times before we find a page to be replaced.

- The major difference between this algorithm and the simpler clock algorithm is:
- Its preference to those pages that have been modified in order to reduce the number of I/Os required.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common

- **Lease Frequently Used** (**LFU**) **Algorithm** replaces page with smallest count

- **Most Frequently Used** (**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

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim.

- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty.

- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All these algorithms have OS guessing about future page access.
- Some applications have better knowledge – i.e., databases.
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications

Allocation of Frames

- Each process needs ***minimum*** number of frames.
- ***Maximum*** of course is total frames in the system.
- Major allocation schemes
  - fixed allocation
  - priority allocation
  - Proportional Allocations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes.

- Keep some as free frame buffer pool

  - Equal Allocation= Number of frames / number of processes
  - EA= 100/ 5 = 20 frames /process

Proportional Allocation

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

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

$$S = \sum s_i$$

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

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

Process2 = 1 =59,62 frames, P=5
Process1 = 5=5 frames, P=1

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

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

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
  - But greater throughput so more common

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

Reclaiming Pages

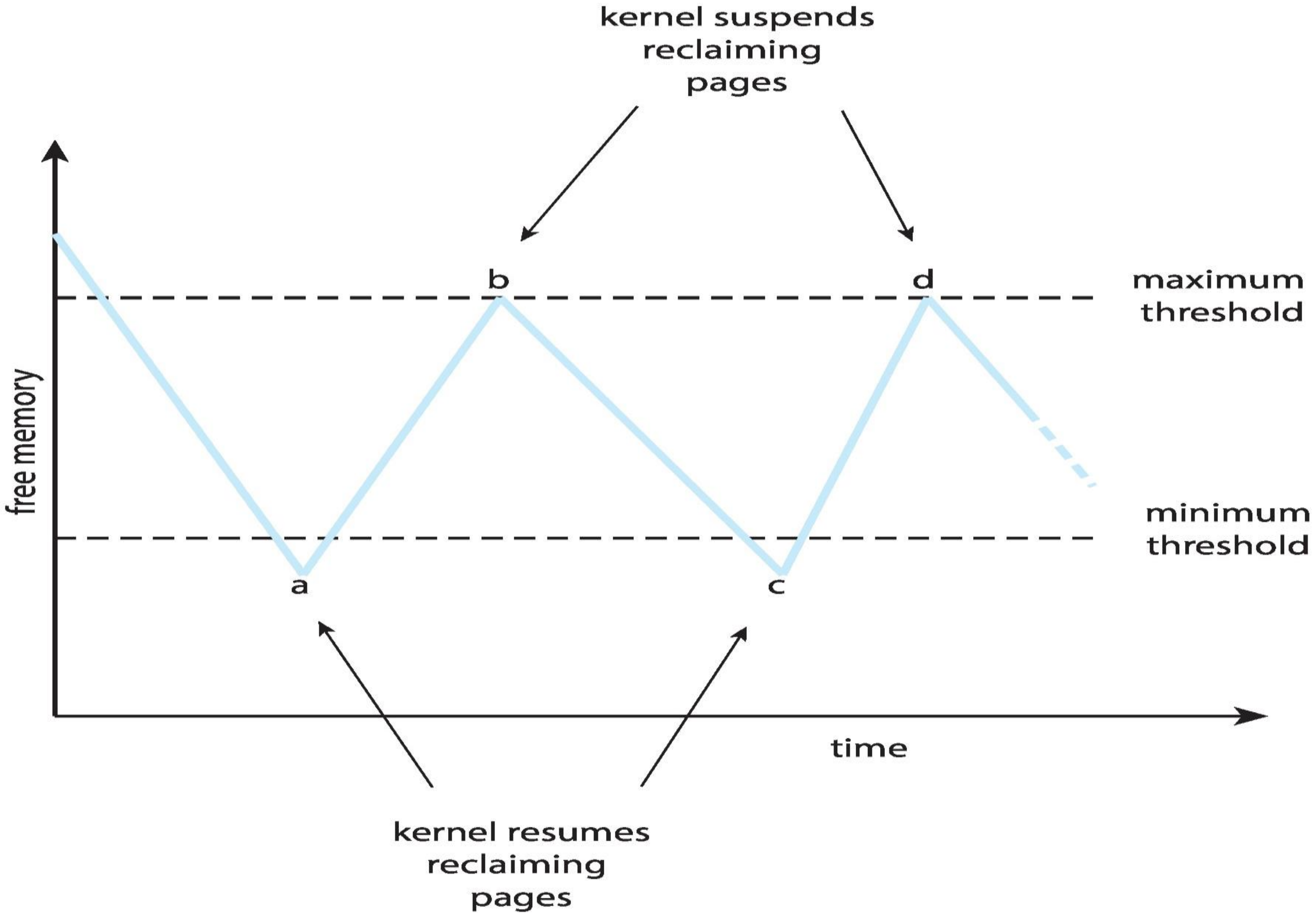
- A strategy to implement global page-replacement policy.

- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement.

- Page replacement is triggered when the list falls below a certain threshold.

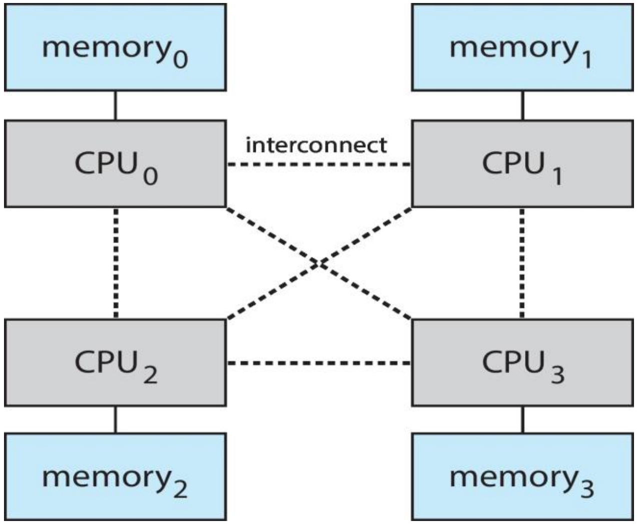
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example


kernel suspends reclaiming pages
free memory
b
d
maximum threshold
minimum threshold
a
c
time
kernel resumes reclaiming pages

Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



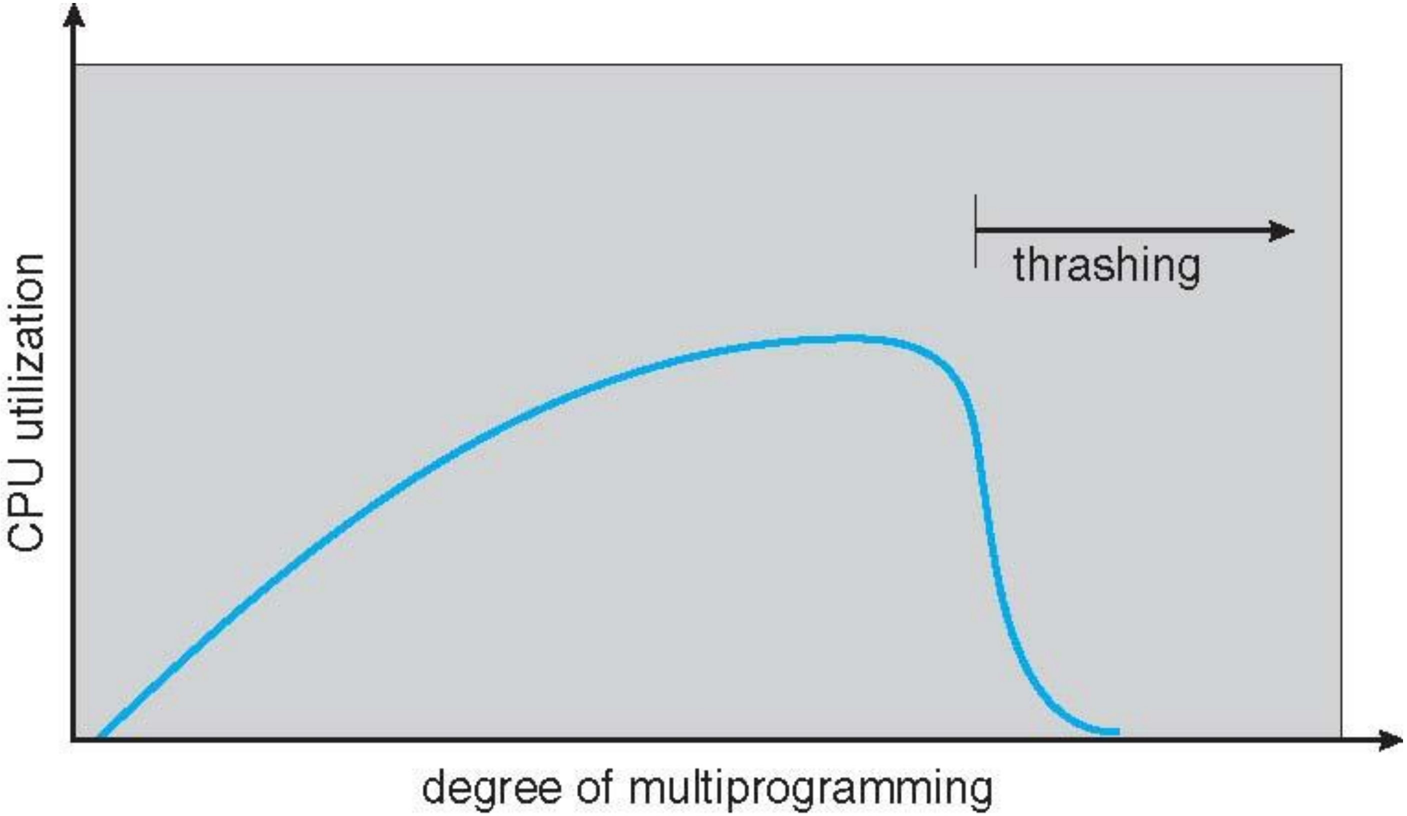
Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Thrashing

- A process is busy swapping pages in and out, this high paging activity is called **Thrashing.**
- A process is thrashing if it is spending more time paging than executing.
- If a process does not have “enough” pages, 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 (Cont.)



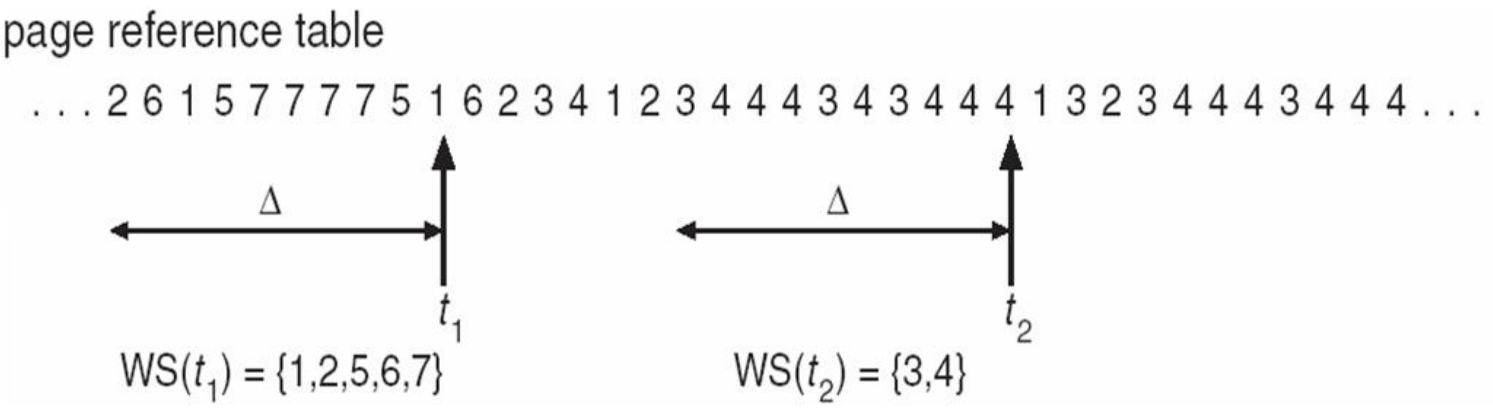
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model= set of pages used actively together
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set 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 $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes



Keeping Track of the Working Set

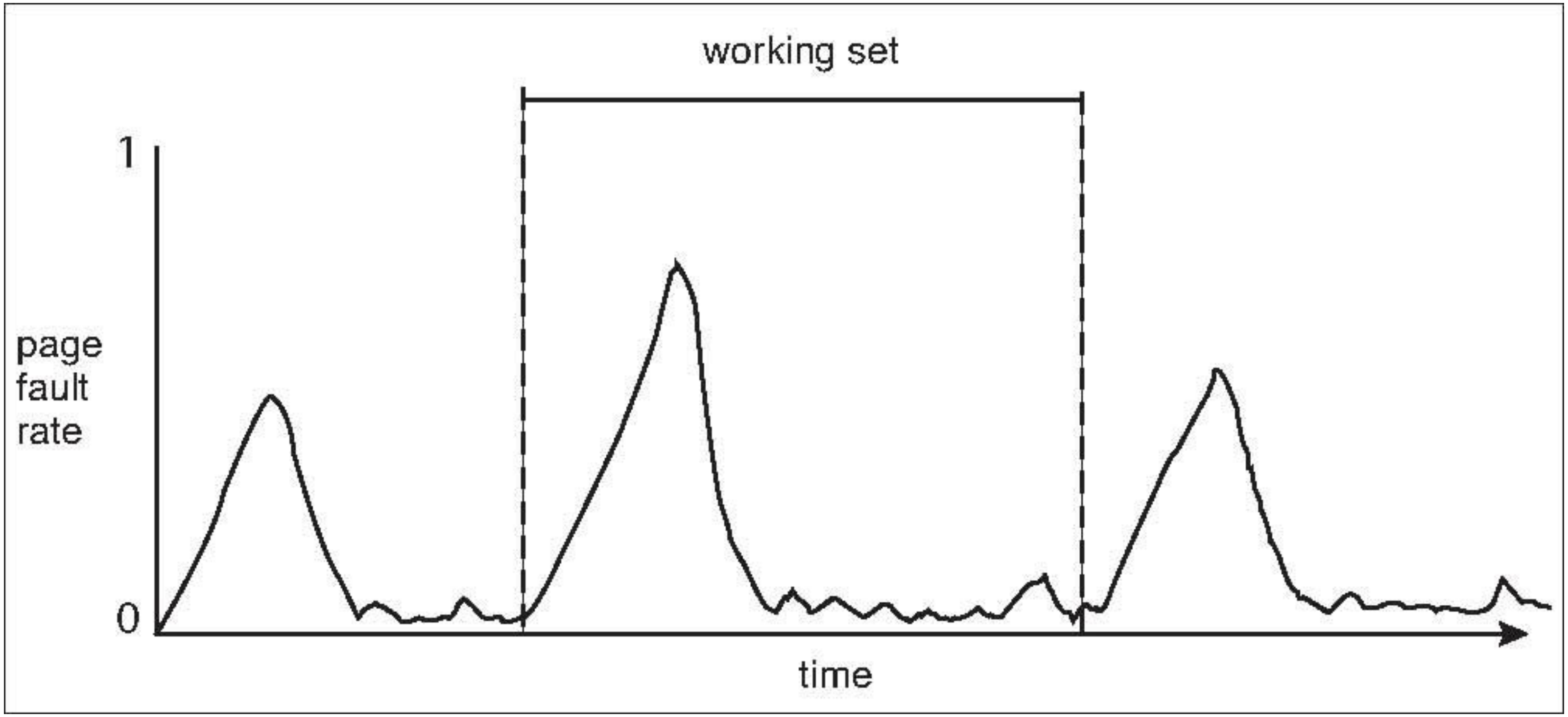
- Approximate with interval timer + a reference bit

- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000-time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000-time units

Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Allocating Kernel Memory

- Treated differently from user memory

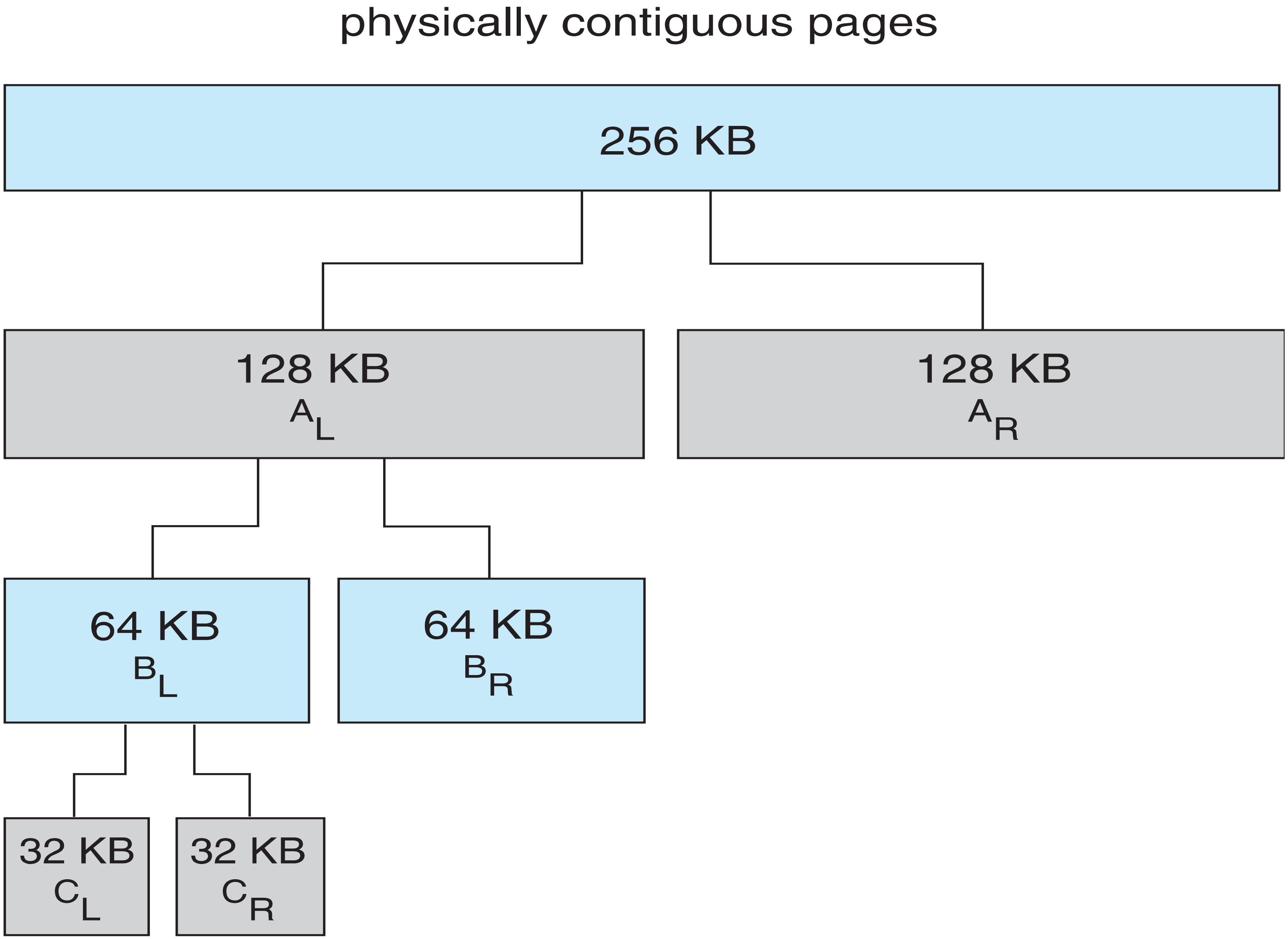
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e., for device I/O

    - 1. Buddy System Allocator
    - 2. Slab Allocator

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 appropriately sized chunk available
- For example, assume 256KB chunk available, kernel requests **21KB**
  - Split into $A_{L\ \text{and}}$ 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**

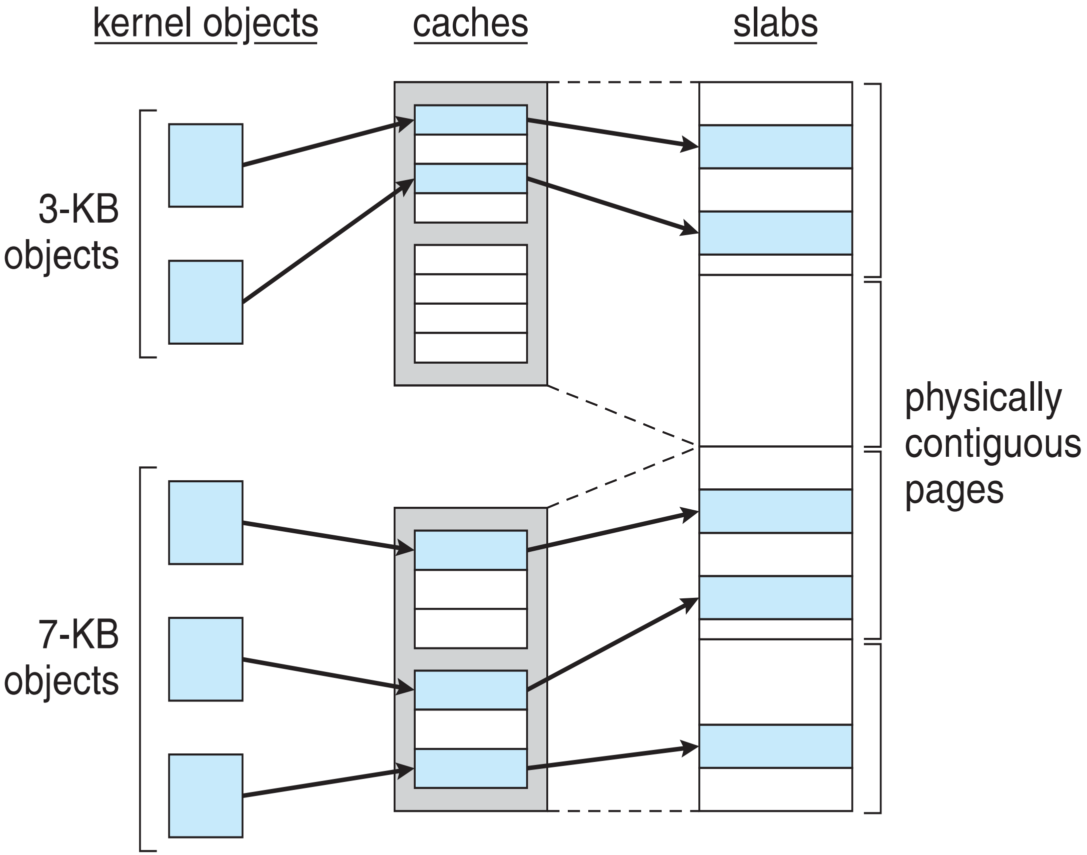
Buddy System Allocator



Slab Allocator

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example, process descriptor is of type `struct task_struct`
- Approx. 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- 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

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or 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 *α* of the pages is used
    - Is cost of ***s * α*** save pages faults > or < than the cost of prepaging
      s * (1- α) unnecessary pages?
    - ***α*** near zero ⇒ prepaging loses

Other Issues – Page Size

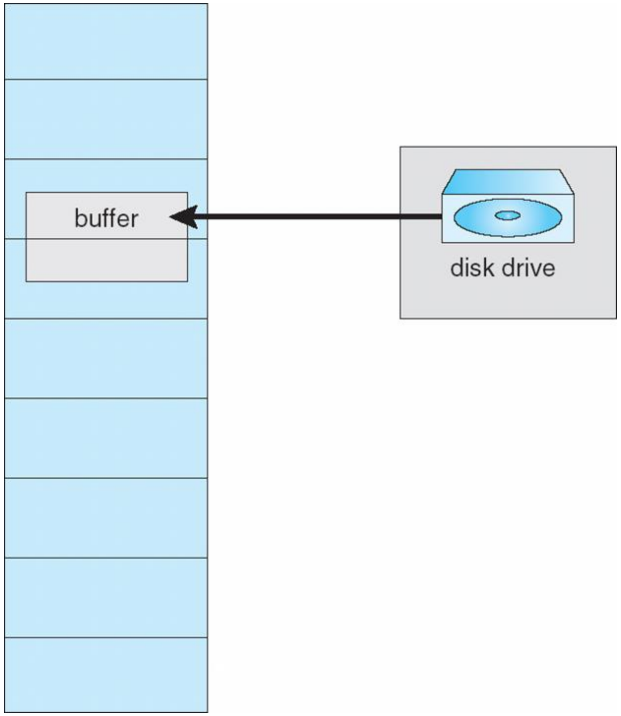
- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise, there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



Thank You!