

Lecture 10: Virtual Memory



Slides Sources

- These slides are originated by the slides from the book
 - <https://www.os-book.com/OS10/slide-dir/index.html>
- Slides are adapted and extended by Sashko Ristov
 - Chapter 10

Semester Outline

- Introduction
- Processes
- Threads
- Interprocess Communication (IPC)
- Process Synchronization
- Scheduling
- Deadlocks
- Memory Management
- **Virtual Memory**
- Multicomputers and Multiprocessors
- I/O & Storage
- Virtualization Concepts
- SW/HW Virtualization

Lecture 10: Virtual Memory

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

Background

Background

- Code needs to be in memory to execute, but **entire program rarely used**
 - Error code, unusual routines, large data structures (e.g., arrays)
- **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

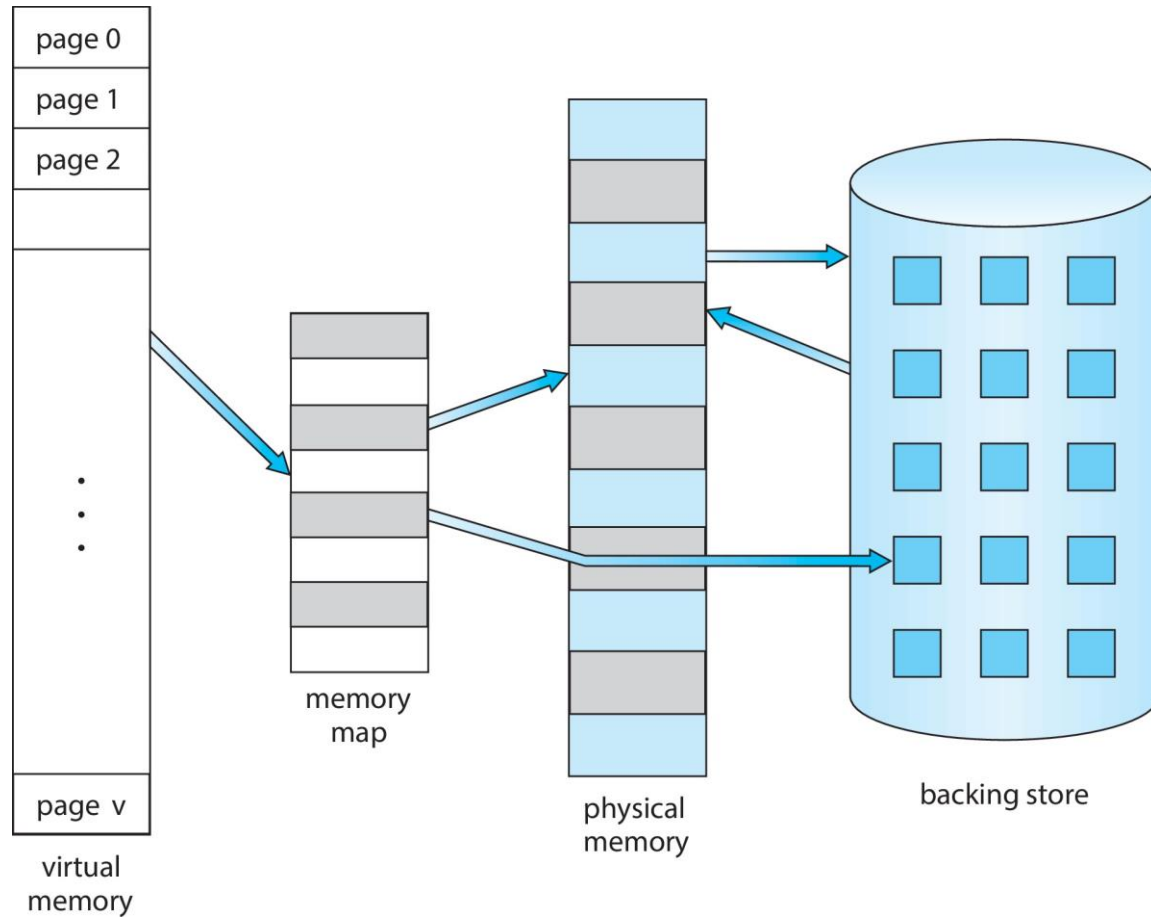
Virtual memory

- **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

Virtual memory (Cont.)

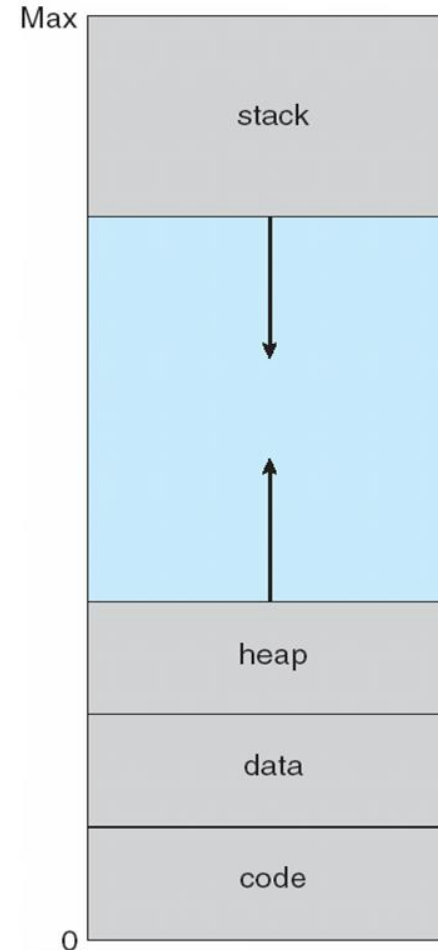
- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- **Virtual memory** can be implemented via:
 - Demand **paging**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

Virtual Memory That is Larger Than Physical Memory

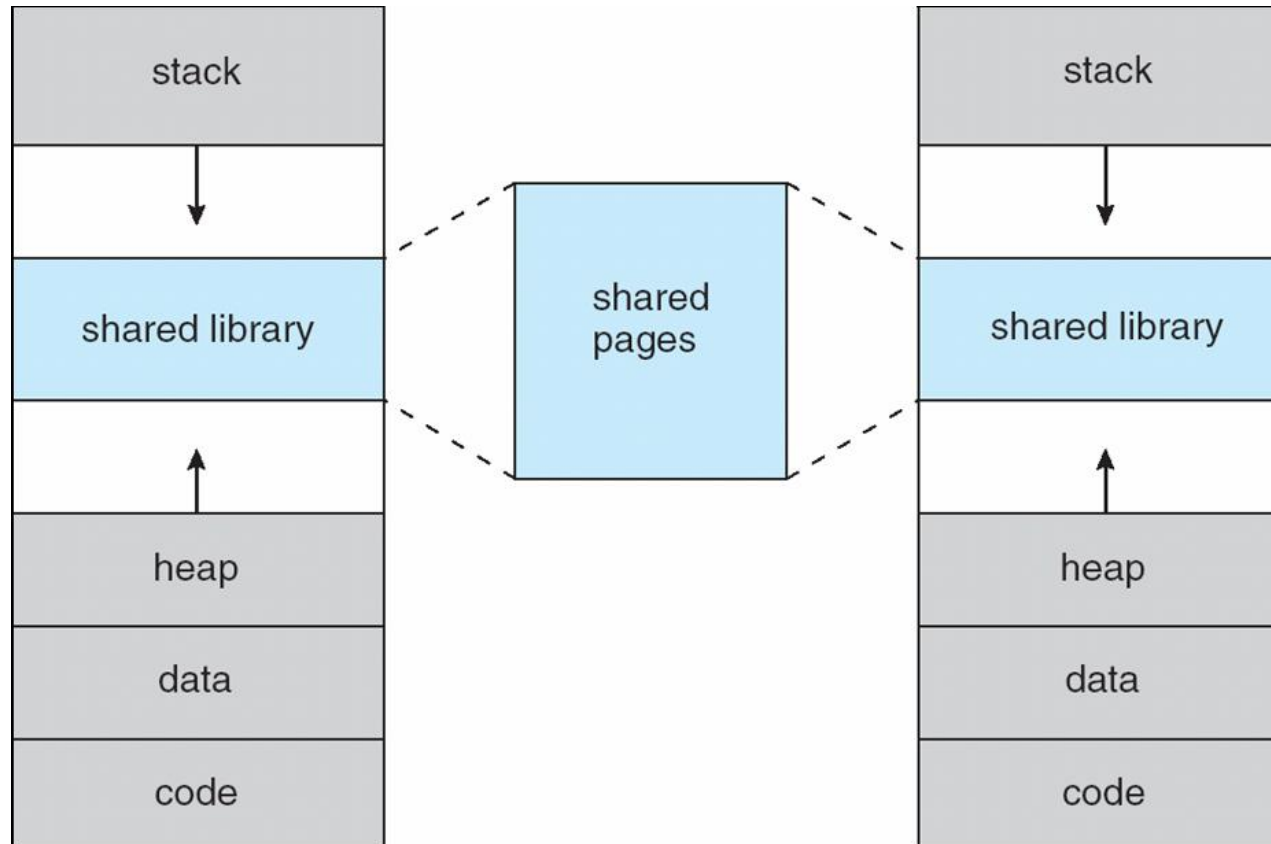


Virtual-address Space

- 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



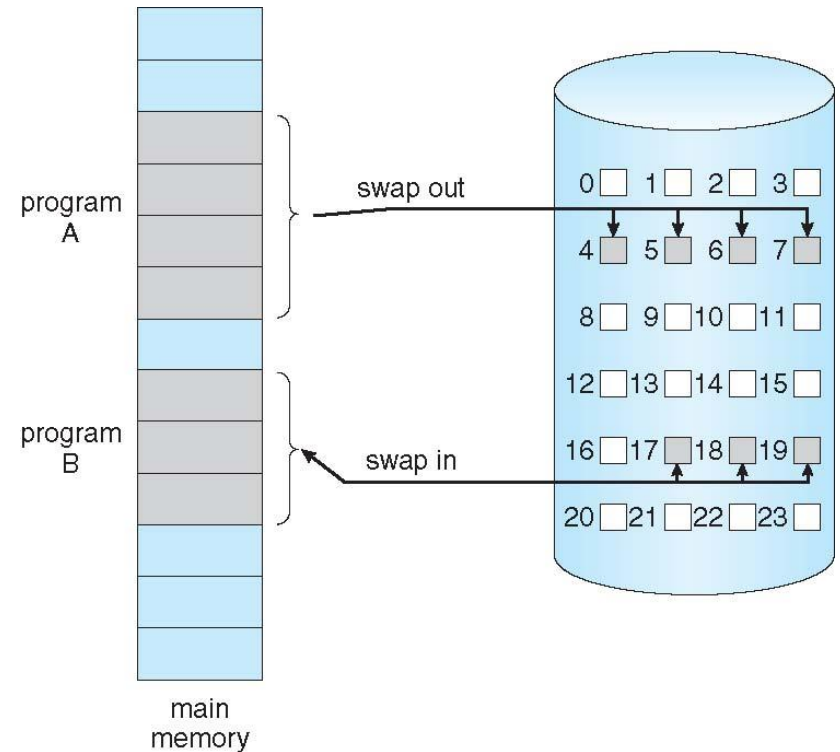
Shared Library Using Virtual Memory



Demand Paging

Demand Paging

- Could bring entire process into memory at load time
- Or **bring a page into memory only when it is needed**
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)



Valid-Invalid Bit

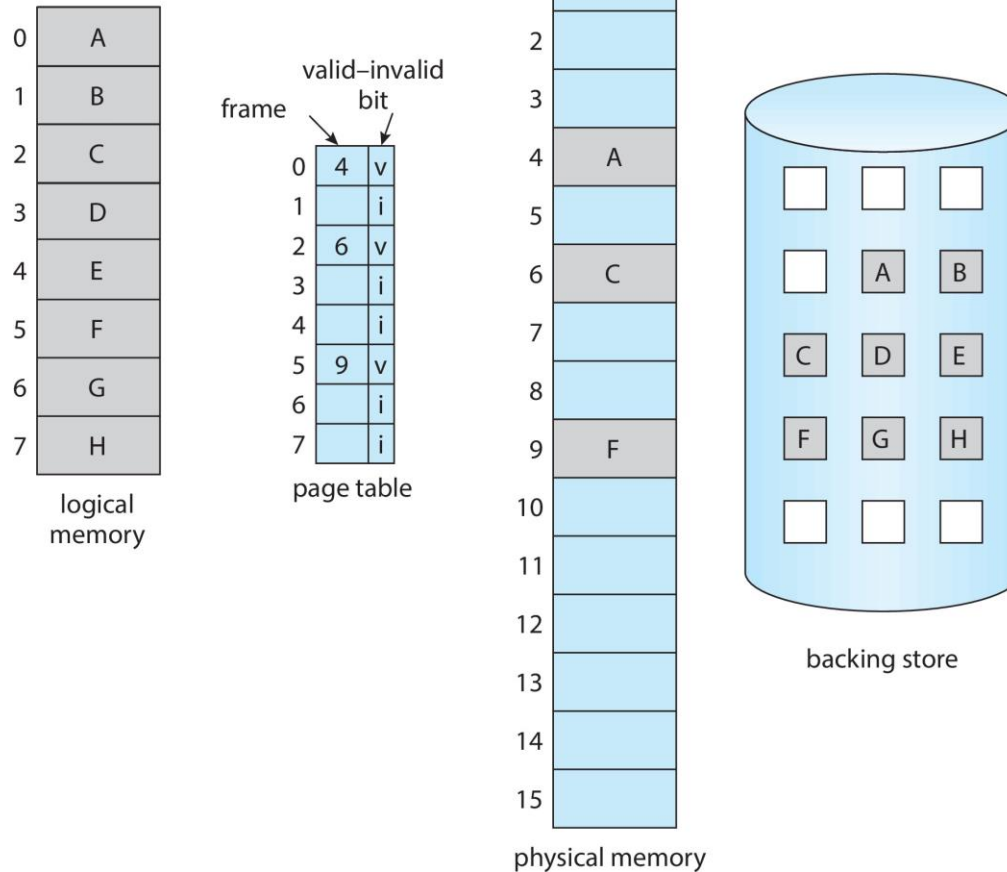
- With each page table entry a valid–invalid bit is associated (**v** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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** \Rightarrow page fault

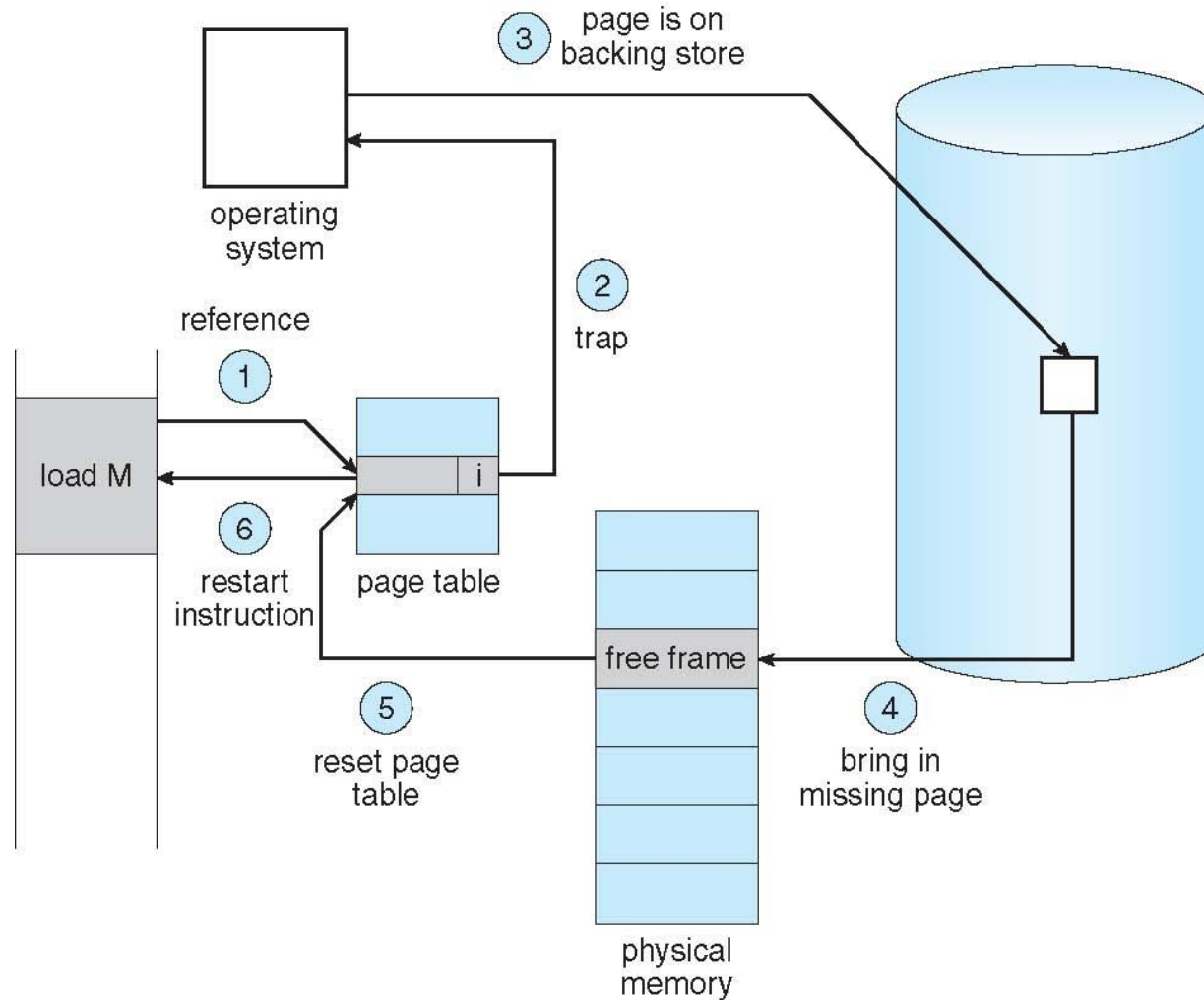
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
 - **Page fault**
2. Operating system **looks** at **another table** to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
3. **Find free** frame
4. **Swap** page into frame via scheduled disk operation
5. **Reset** tables to indicate page now in memory
Set validation bit = **v**
6. **Restart** the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)



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

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- The operating system typically allocates free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

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:
 - a) Wait in a queue for this device until the read request is serviced
 - b) Wait for the device seek and/or latency time
 - c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

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

- Three major activities
 - **Service the interrupt** – careful coding means just several hundred instructions needed
 - **Read the page** – lots of time
 - **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 = $(1 - p)$ x memory access
 - + p (page fault overhead
 - + swap page out
 - + swap page in)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000$
 $= 200 + p \times 7,999,800$
- 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 want performance degradation < 10 percent
 - $220 > 200 + 7,999,800 \times p$
 $20 > 7,999,800 \times p$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses

Q/A

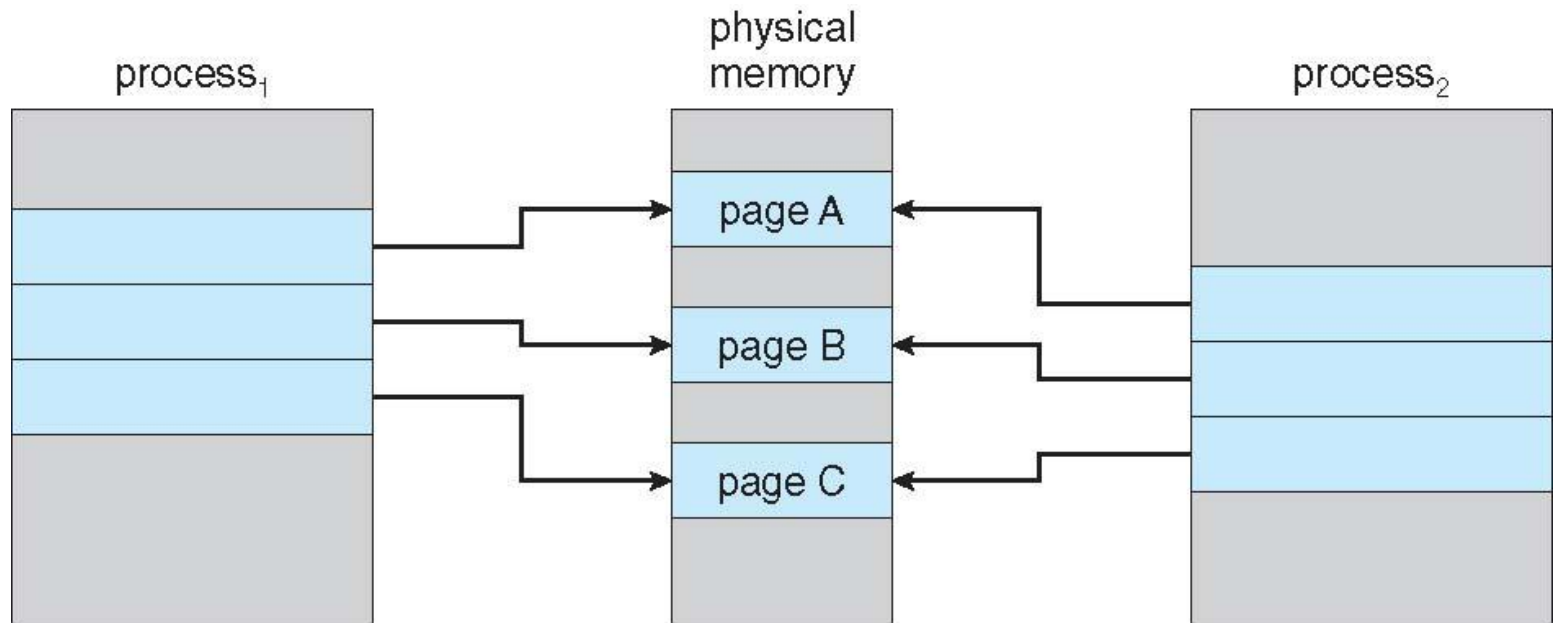


Copy-on-Write

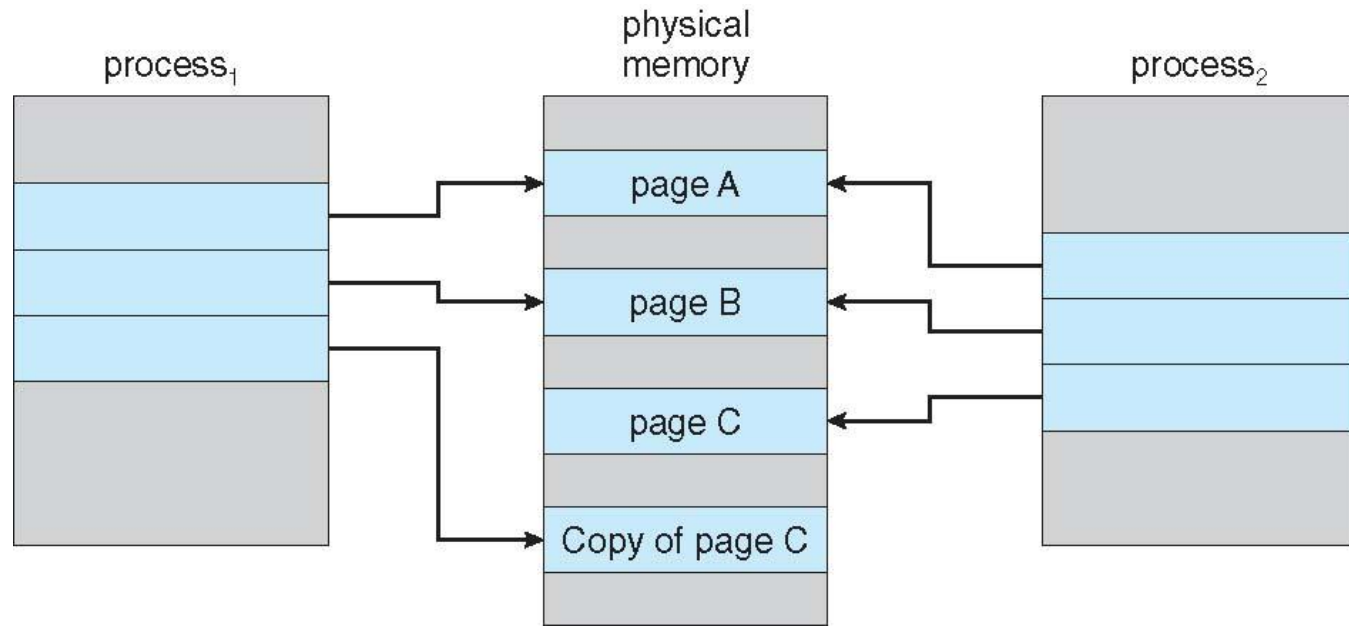
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then the page is 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
 - Pool should always have free frames for fast demand page execution
 - ▶ Don't want to have to free a frame as well as other processing on page fault

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if There is no Free Frame?

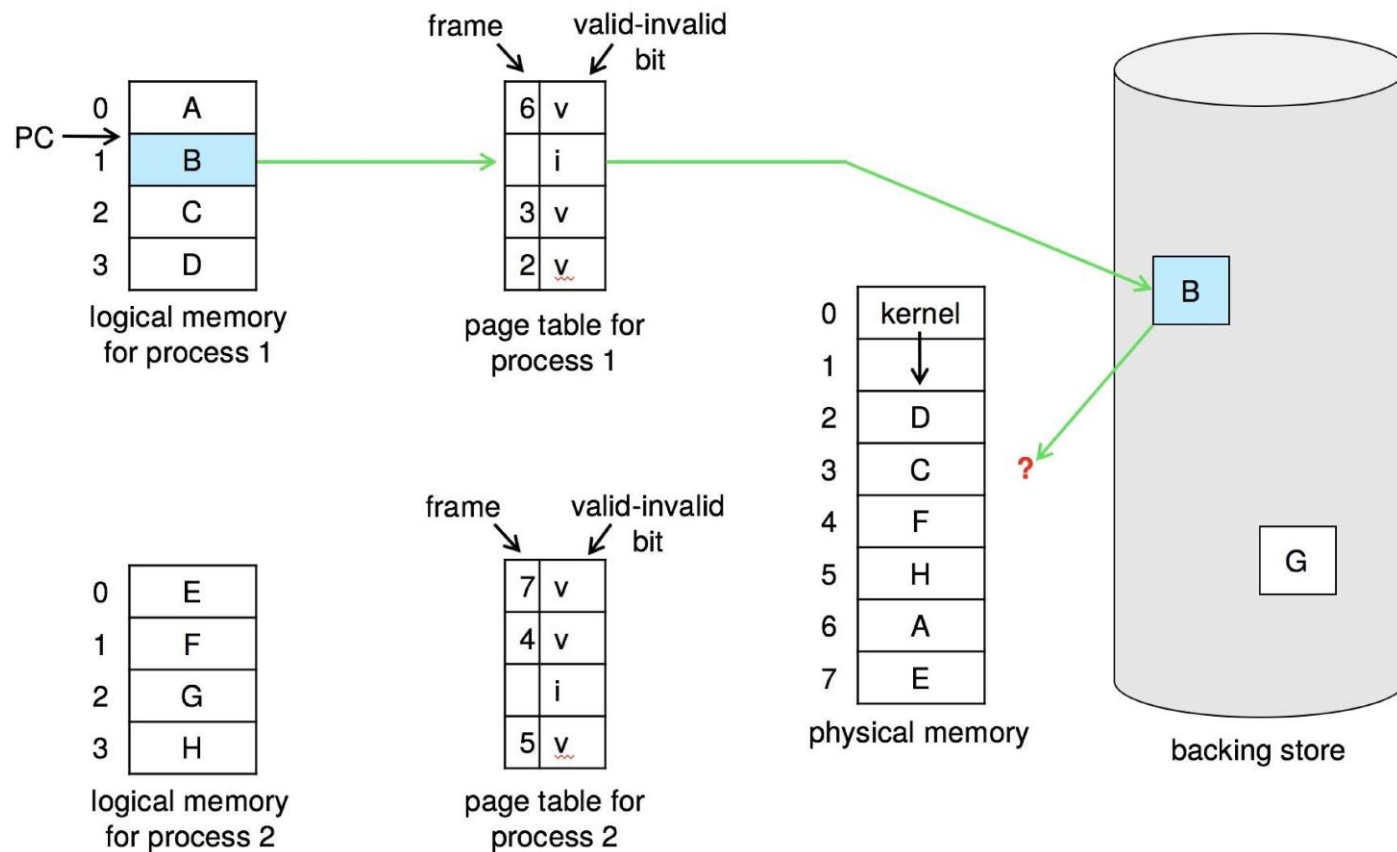
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

Page Replacement

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

Need For Page Replacement

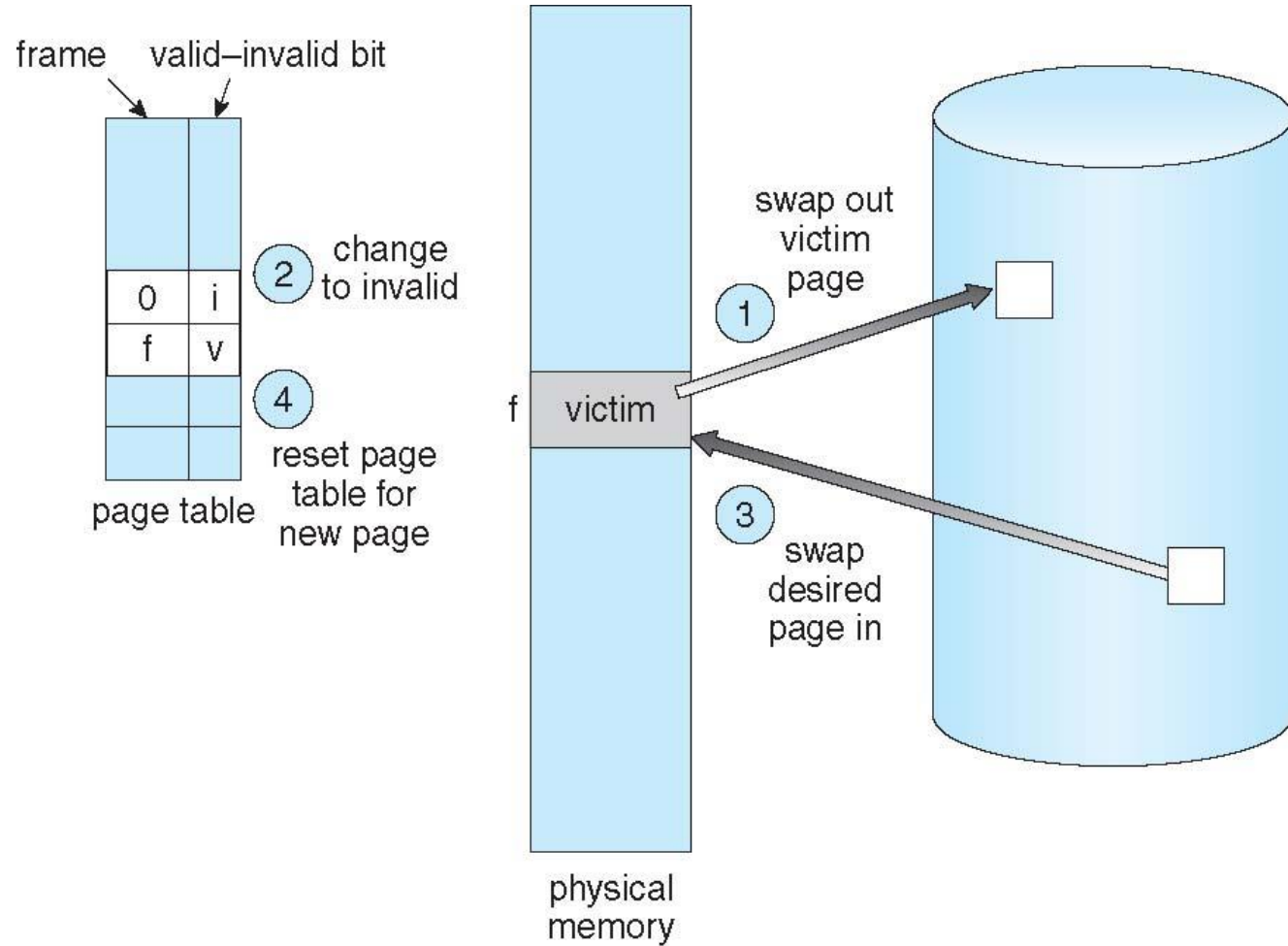


Basic Page Replacement

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 (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

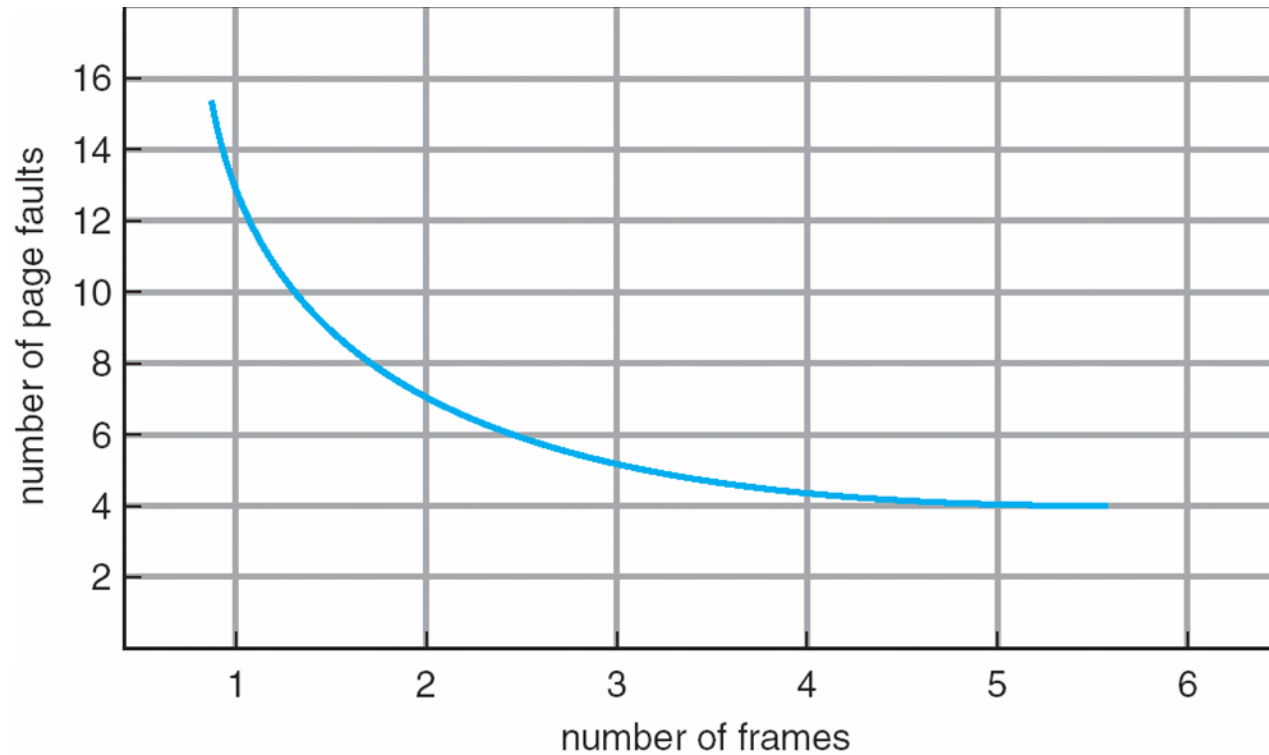
Page Replacement



Page and Frame Replacement Algorithms

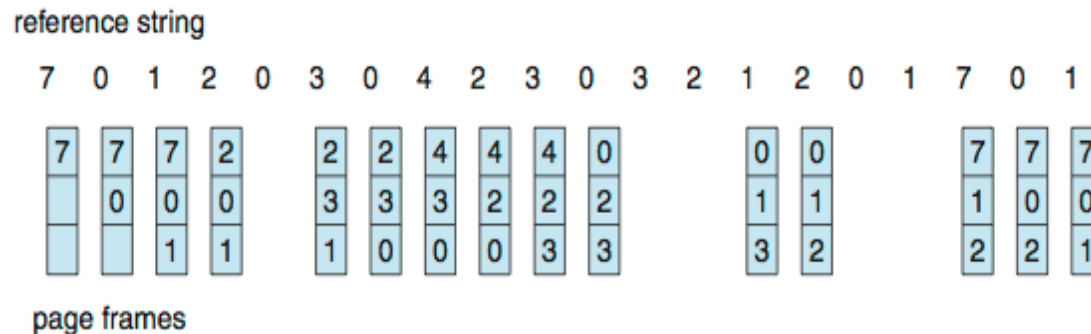
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) 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
- In all our examples, the **reference string** of referenced page numbers is
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus the Number of Frames



First-In-First-Out (FIFO) Algorithm

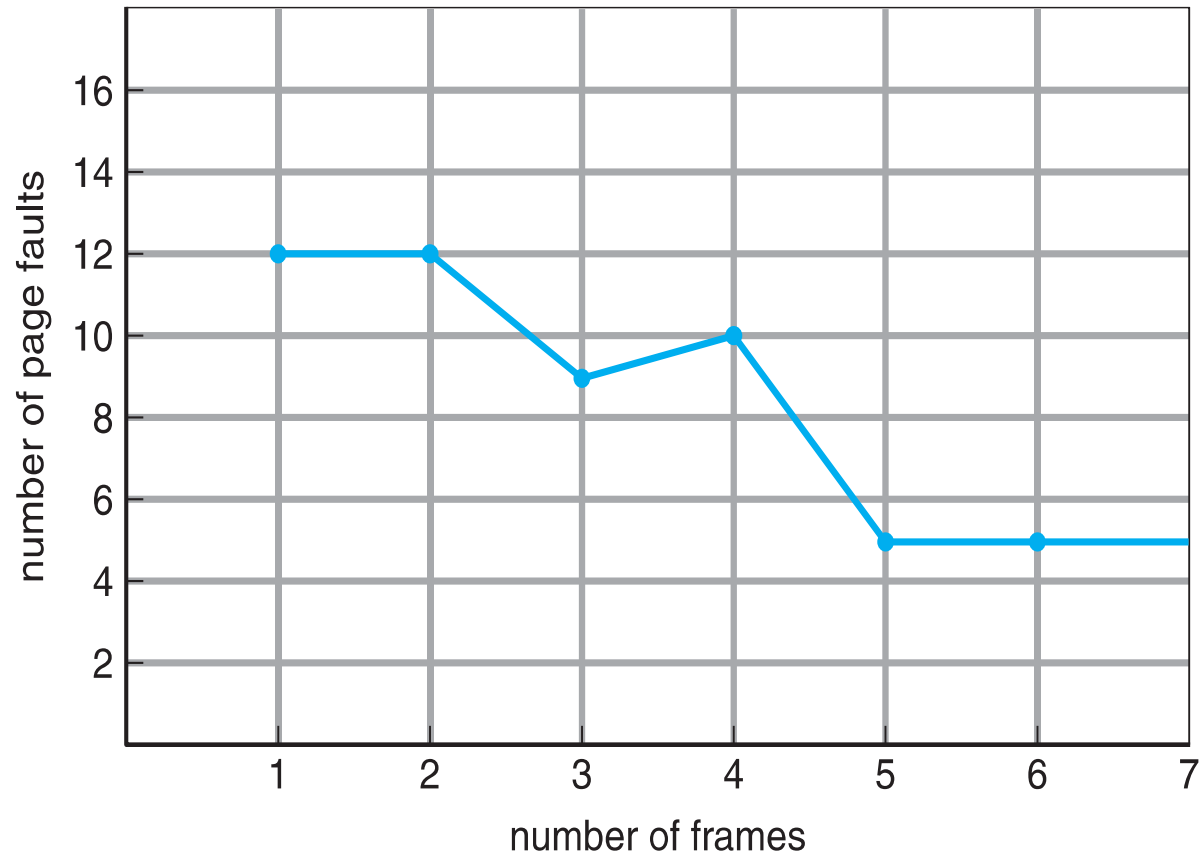
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

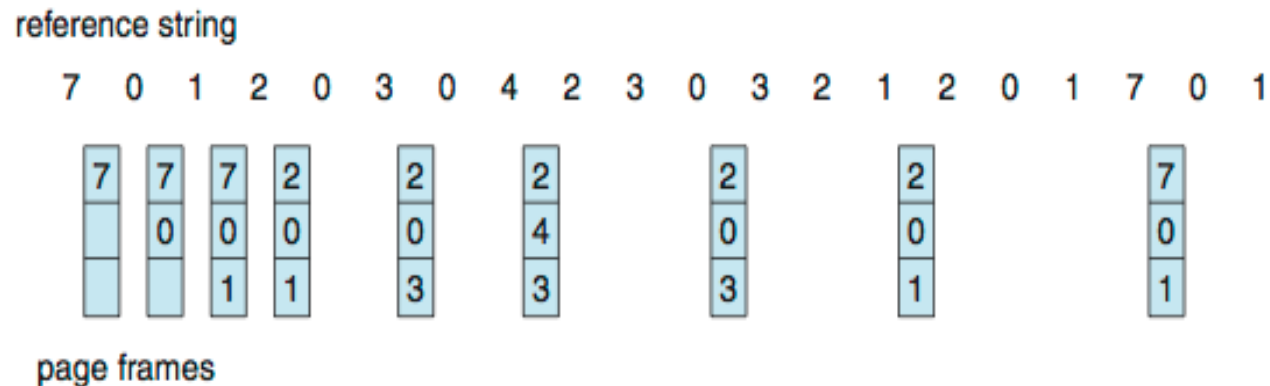
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames (4 compared to 3) can cause more page faults!
 - **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue

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

reference string

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

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

page frames

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

LRU Algorithm (Cont.)

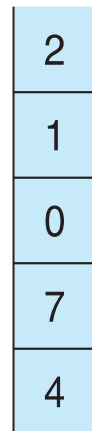
- **Counter** implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value
 - ▶ Search through table needed
- **Stack** implementation
 - Keep a stack 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 Algorithm (Cont.)

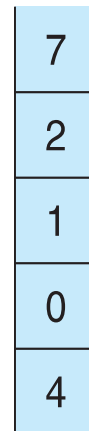
- LRU and OPT are cases of **stack algorithms** that **don't have Belady's Anomaly**
- Use of A Stack to Record Most Recent Page References

reference string

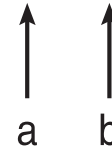
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



LRU Approximation Algorithms

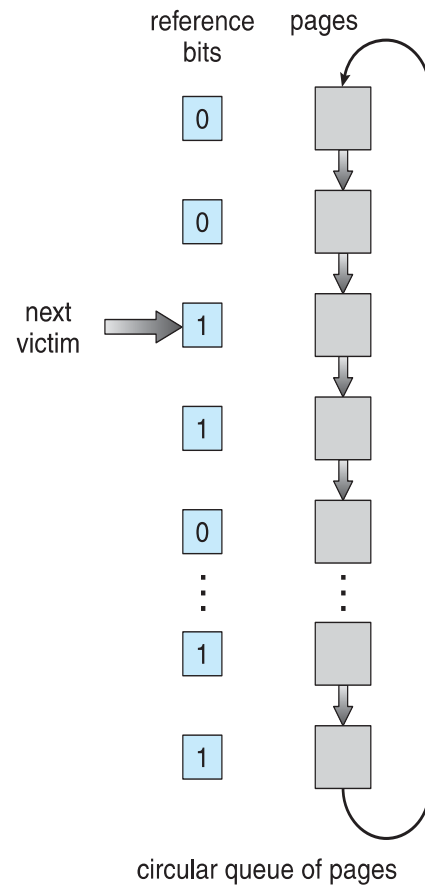
- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order, however

LRU Approximation Algorithms (cont.)

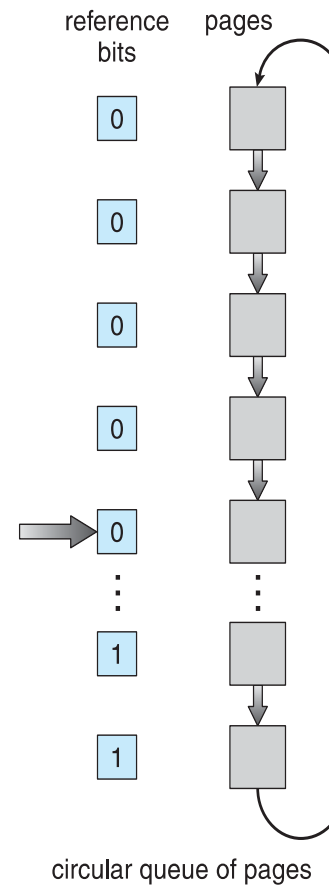
■ Second-chance algorithm

- 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 Algorithm



(a)



(b)

Enhanced Second-Chance Algorithm

- Improve algorithm by using **reference bit** and **modify bit** (if available) in concert
- Take ordered pair (reference, modify):
 - (0, 0) neither recently used nor modified – best page to replace
 - (0, 1) not recently used but modified – not quite as good, must write out before replacement
 - (1, 0) recently used but clean – probably will be used again soon
 - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
 - Might need to search circular queue several times

Q/A



Allocation of Frames

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations between *minimum* and *maximum*

Fixed Allocation

- **Equal** allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - 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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$

Global vs. Local Allocation

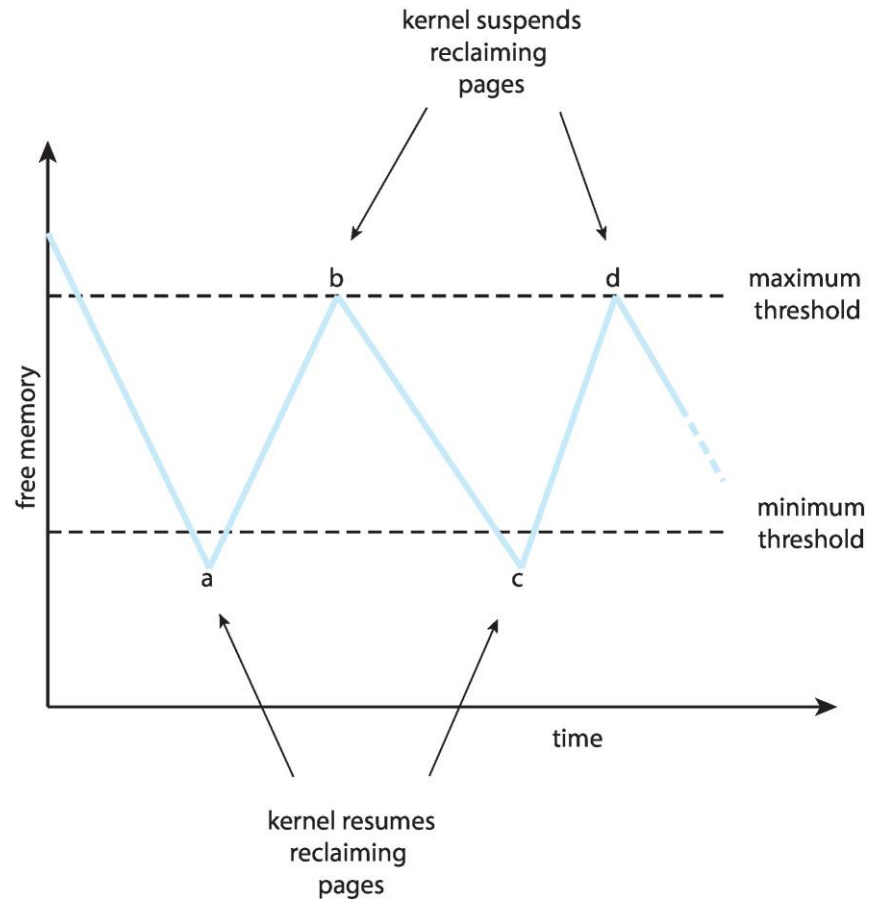
- **Global replacement** – process selects a replacement frame from the set of all frames; **one process can take a frame from another**
 - But then process execution time can vary greatly
 - But greater throughput so more common

- **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



Memory Compression

Memory Compression

- An alternative to paging is **memory compression**.
 - Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames

free-frame list

head → 7 → 2 → 9 → 21 → 27 → 16

modified frame list

head → 15 → 3 → 35 → 26

- Assume that this number of free frames falls below a certain threshold that triggers page replacement.
- The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list.
 - It first places these frames on a modified-frame list.
- Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list.

Memory Compression (Cont.)

- Let 15, 3, and 35 are to be compressed. They are compressed in the first free page – 7.
 - Then they are placed in the free-frame list.
- If now some of pages 15, 3, or 35 is referenced
 - a page fault occurs
 - compressed frame 7 is restored
 - ▶ Restoring the three pages in memory

free-frame list

head → [2] → [9] → [21] → [27] → [16] → [15] → [3] → [35]

modified frame list

head → [26]

compressed frame list

head → [7]

Q/A



Allocating Kernel Memory

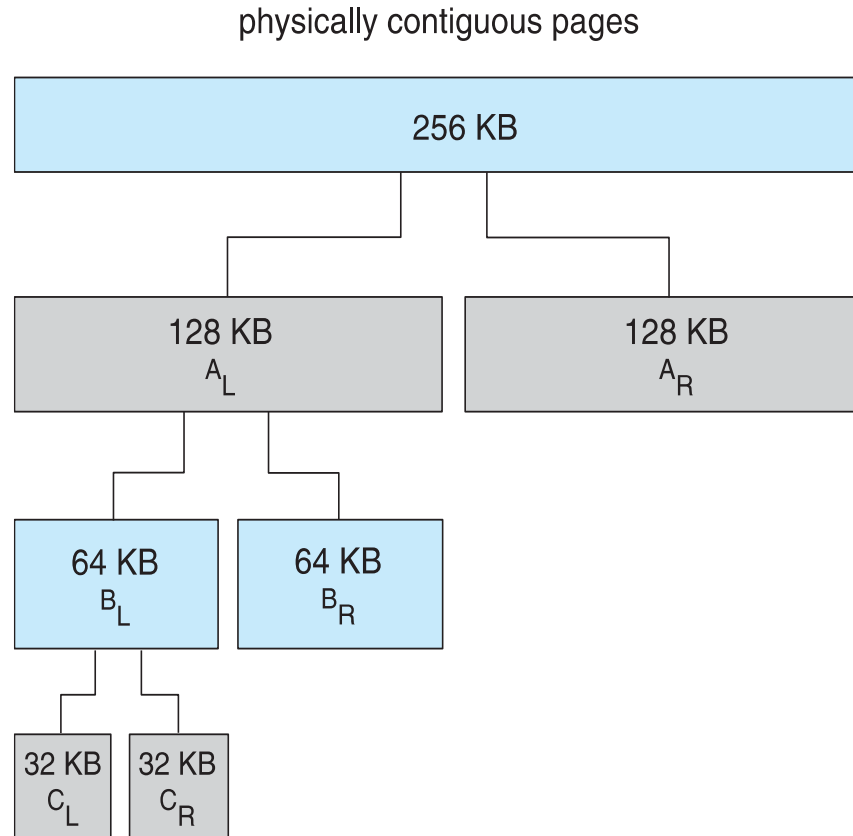
Allocating Kernel Memory

- 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

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
- 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

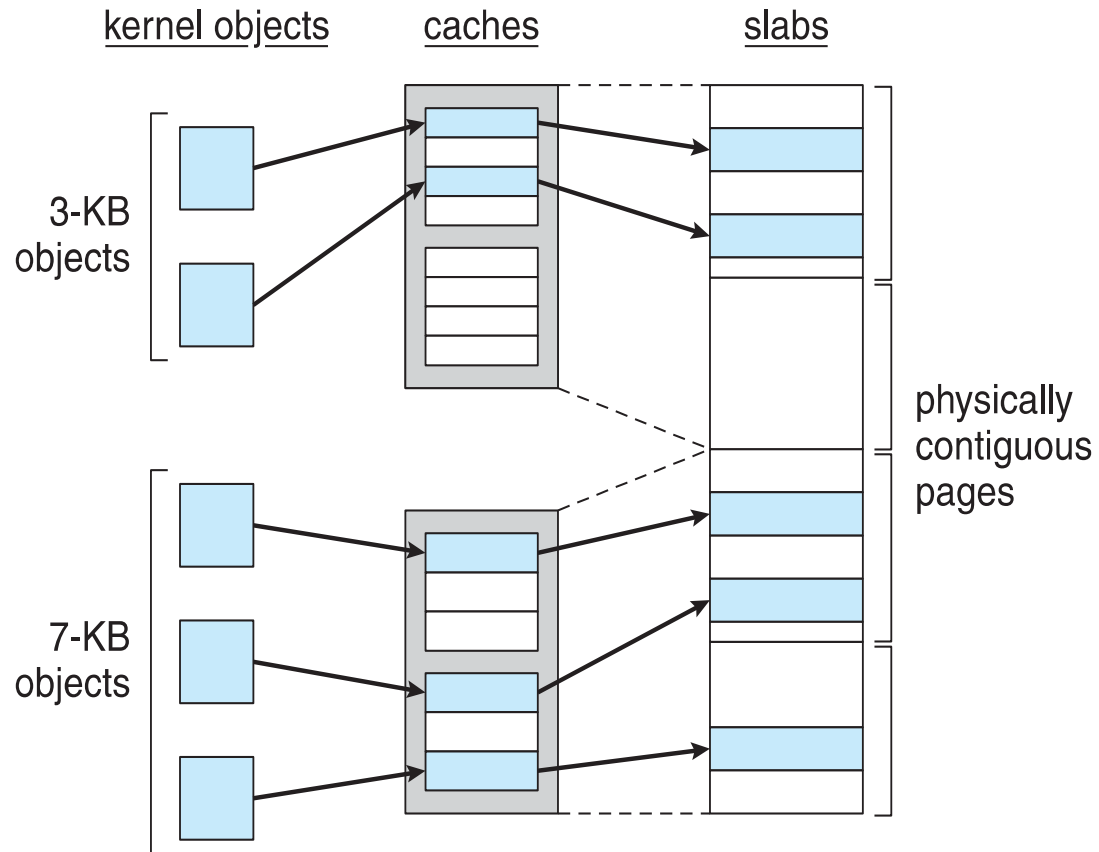
Buddy System Allocator



Slab Allocator

- Alternate strategy
- **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
 - If no empty slabs, new slab allocated
- 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
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

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 $s * \alpha$ save pages faults $>$ or $<$ than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
 - α near zero \Rightarrow prepaging loses

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

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- 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

Program Structure

- Program structure

- `int[128,128] data;`
- Each row is stored in one page
- 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

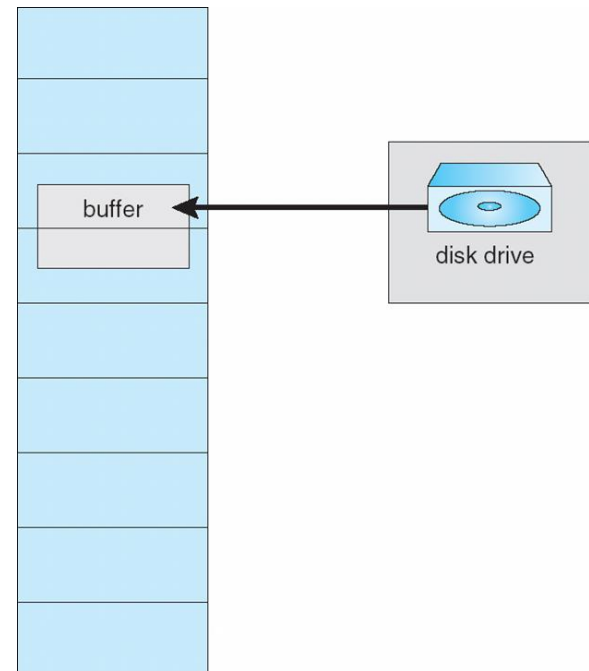
- Program 2

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

128 page faults

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



Q/A

