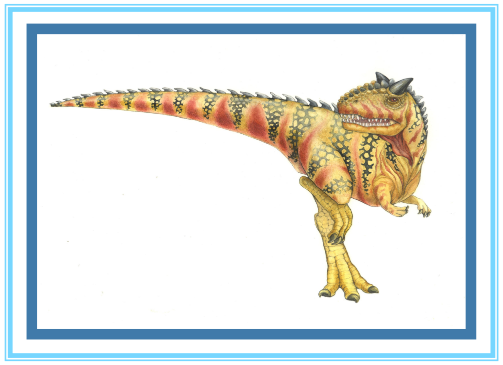
Chapter 9: Virtual Memory



Chapter 9: Virtual Memory

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

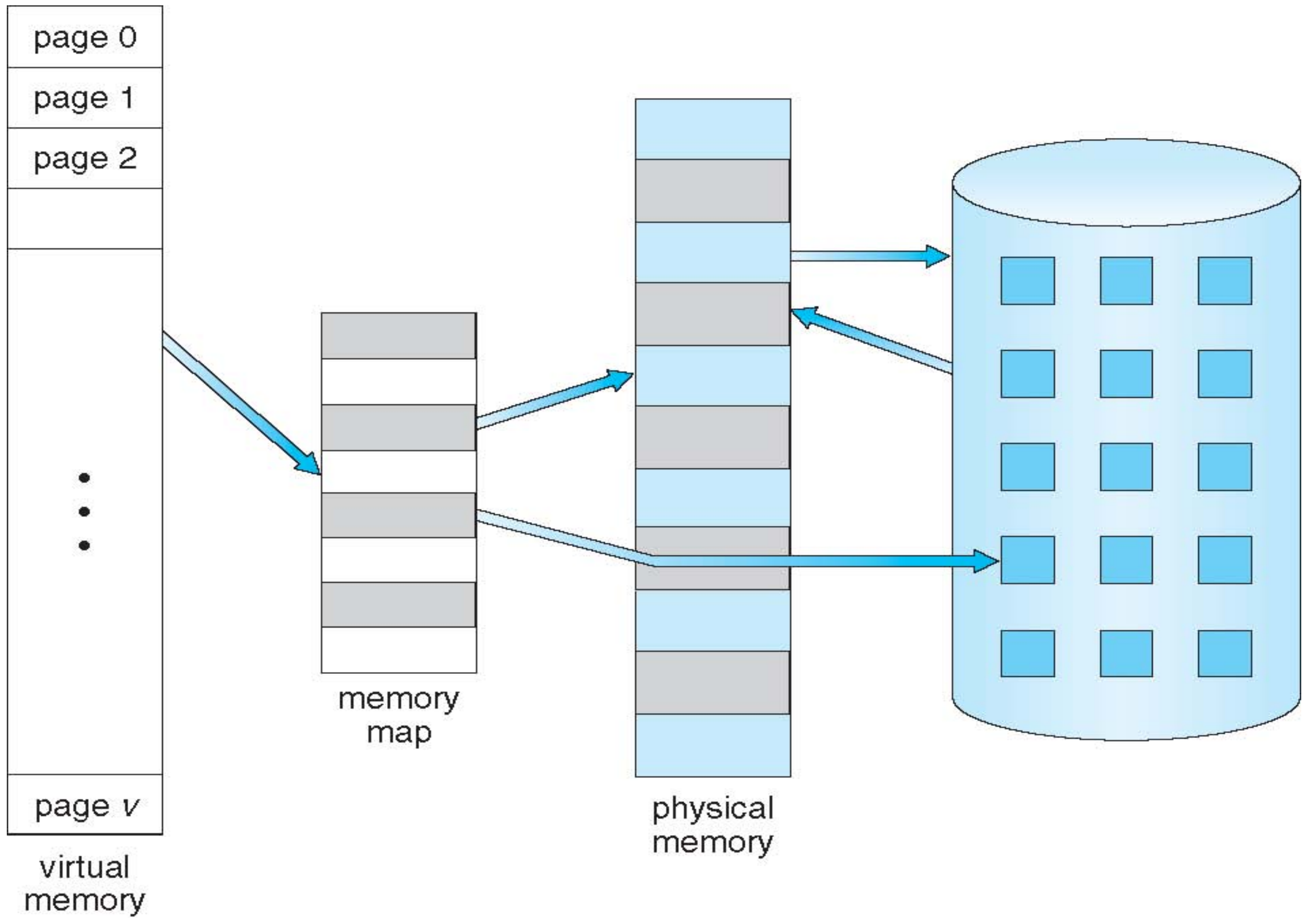
Background

- 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
  - Program and programs could be larger than physical memory

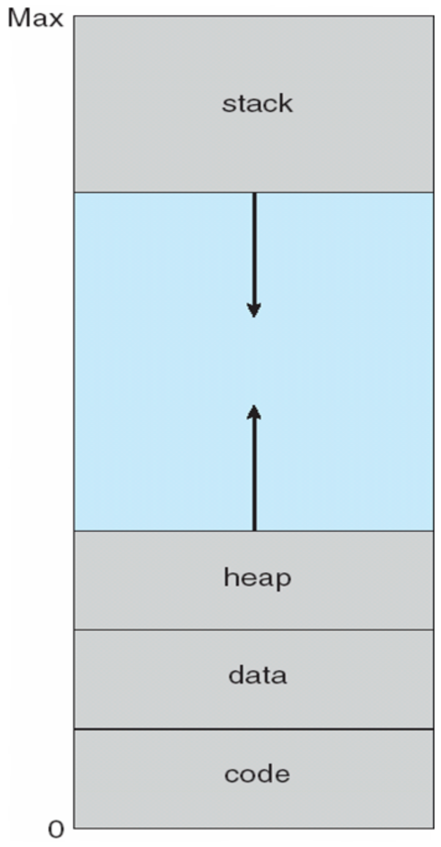
Background

- **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
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



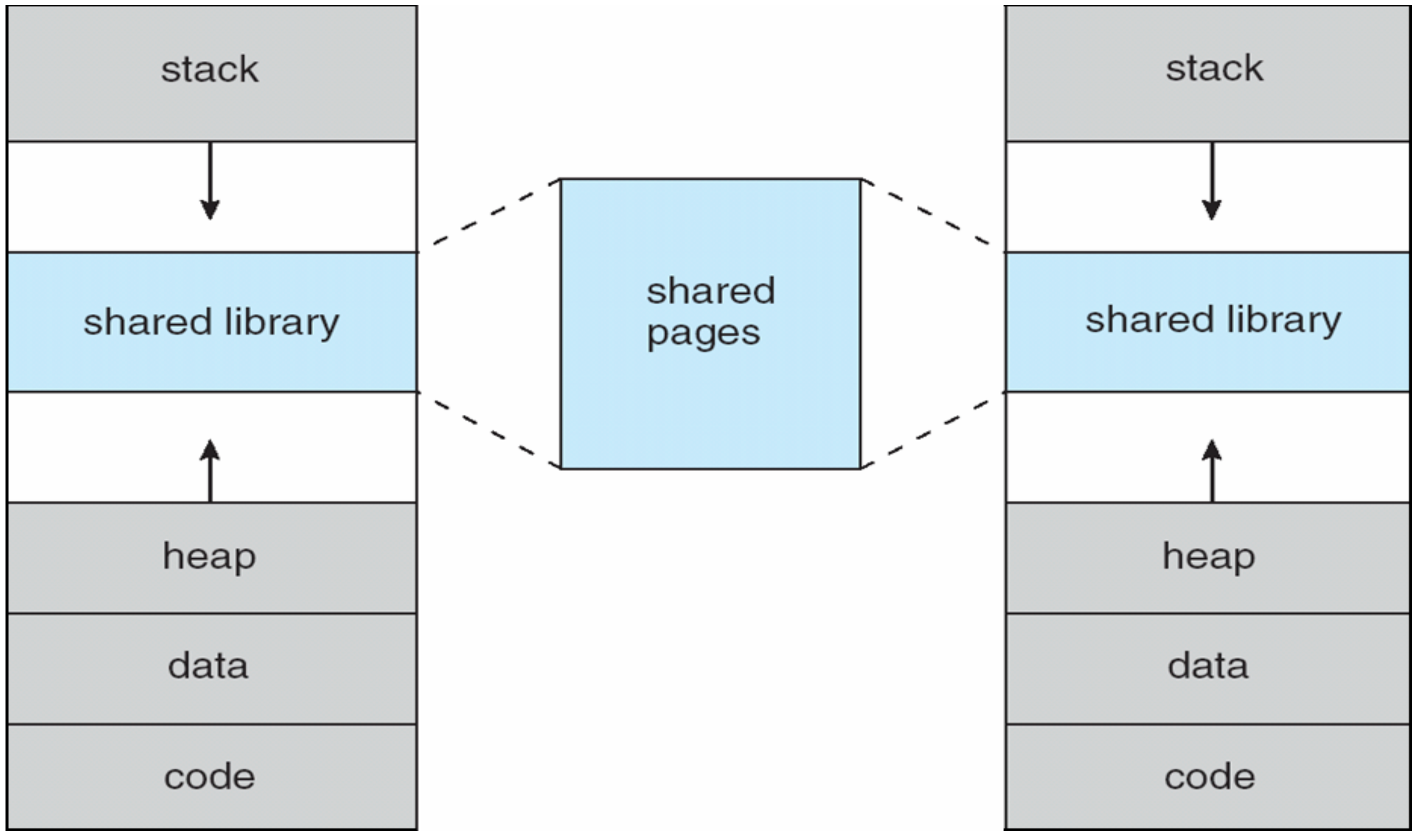
Virtual-address Space



Virtual Address Space

- 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



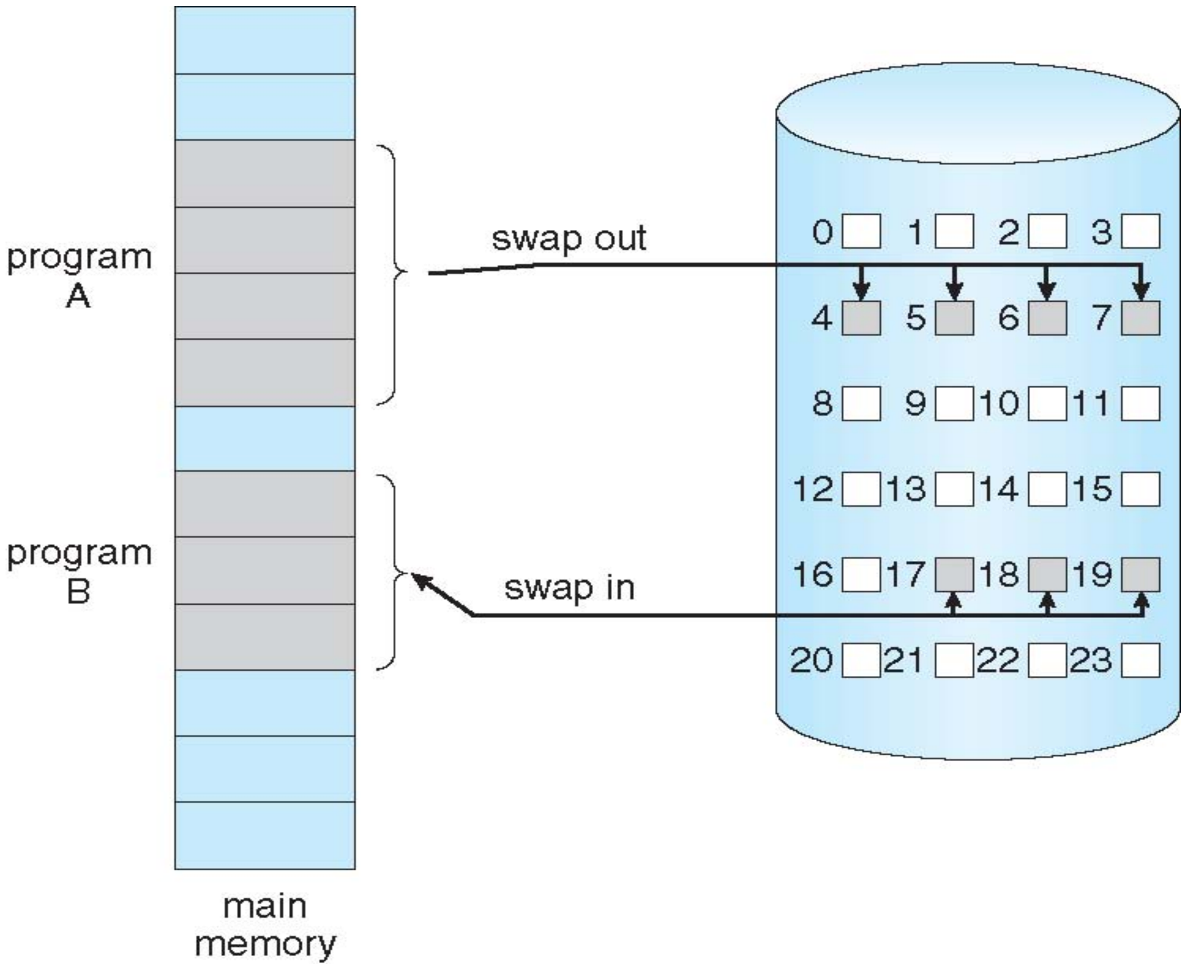
Demand Paging

- 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

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

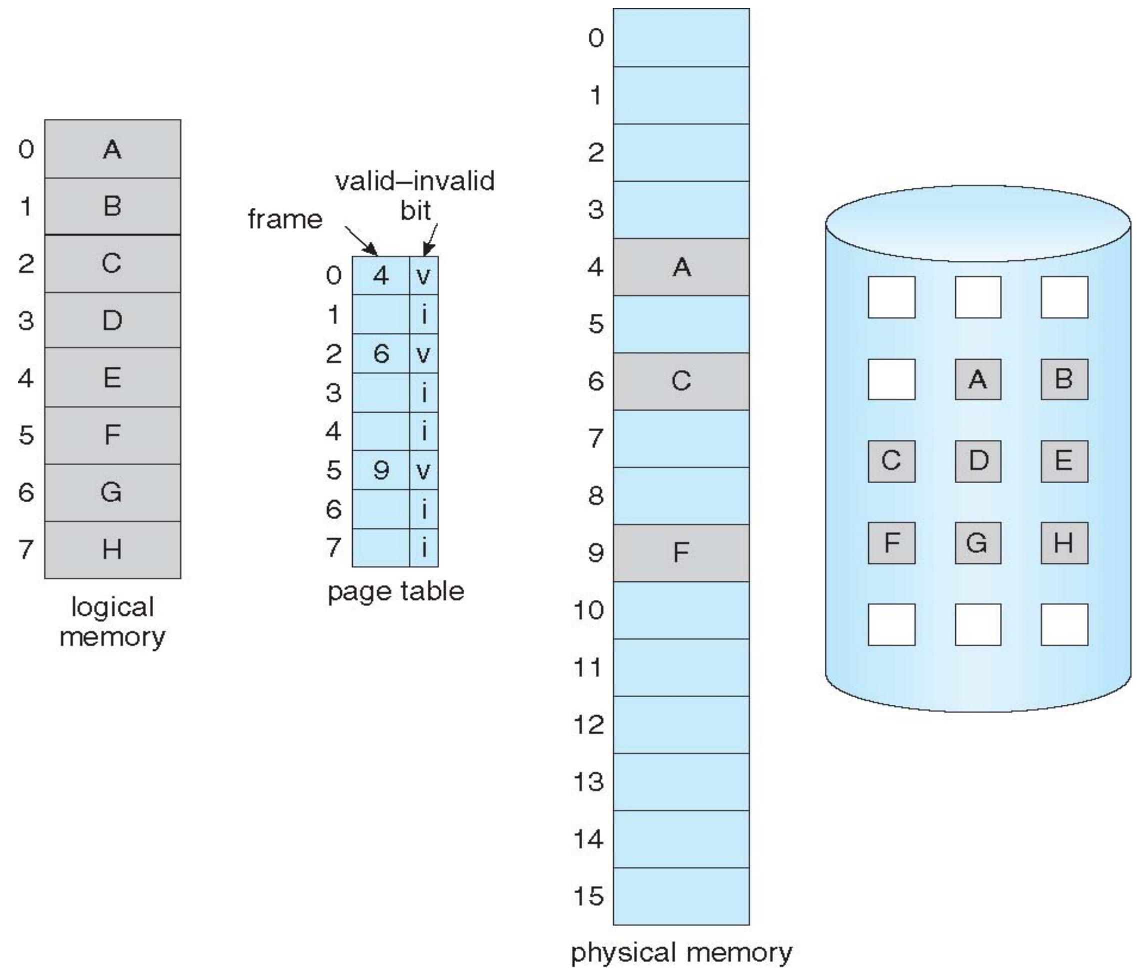
- With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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

page table

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

Page Table When Some Pages Are Not in Main Memory

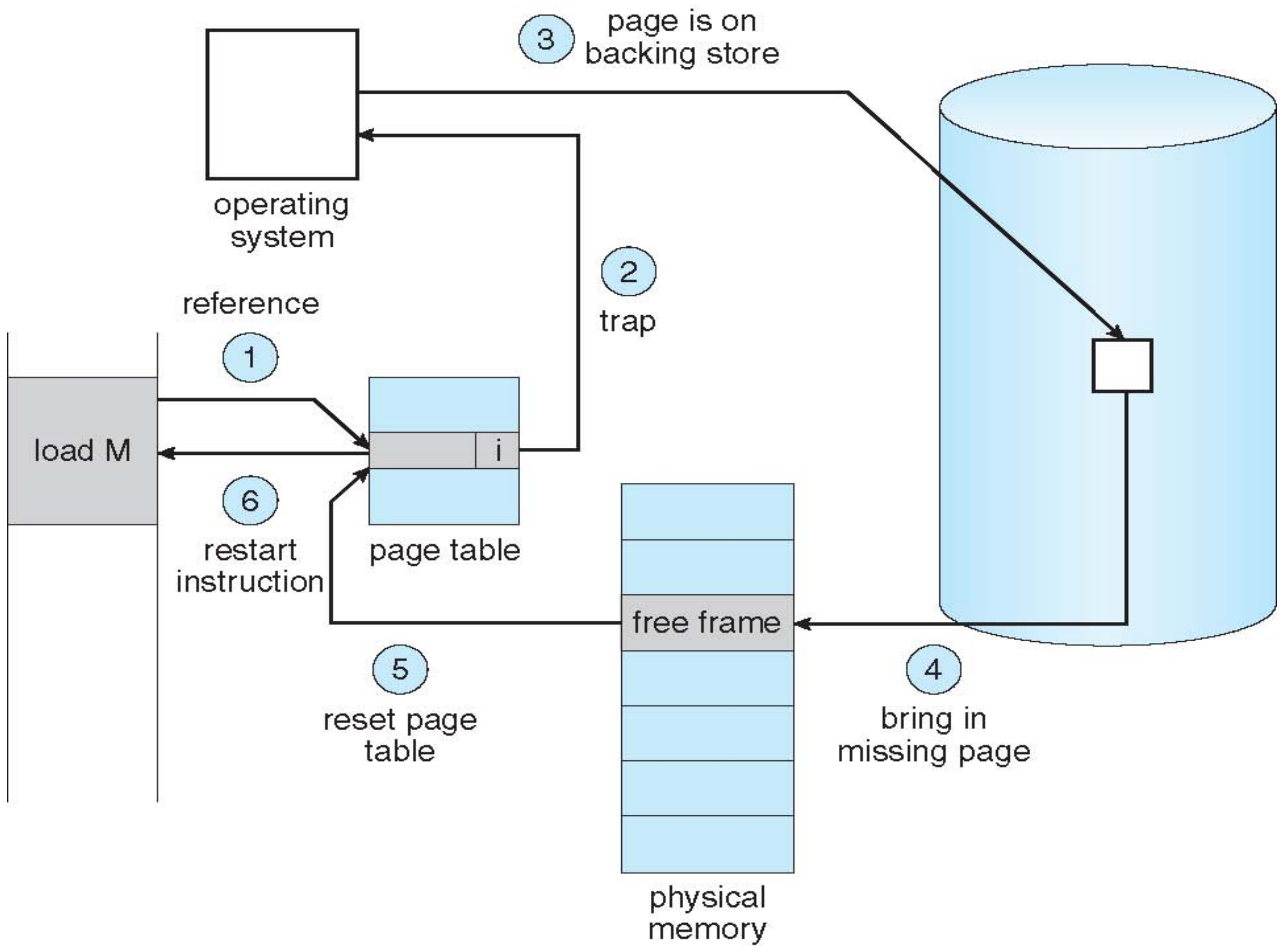


Page Fault

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

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find an empty 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

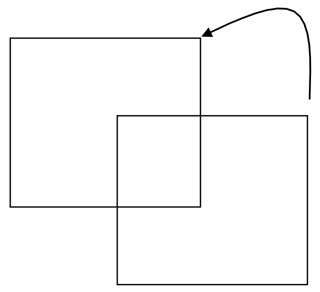


Aspects of Demand Paging

- 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
  - 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 an instruction that could access several different locations
  - block move



  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?
    - different solutions: check bounds first, ensuring pages are valid, or use temporary buffers, ...

Performance of Demand Paging

- Stages in Demand Paging

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.)

- 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)

$$\begin{aligned} \text{EAT} = {} & (1 - p) \times \text{memory access} \\ & + p \times (\text{page fault overhead} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead}) \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds (8,000,000 nanoseconds)
- EAT = $(1 – p)$ x 200 + p x (8 milliseconds)

  = $(1 – p)$ x 200 + p x 8,000,000

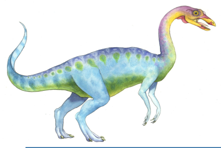
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds

  This is a slowdown by a factor of 40!
- If 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 Optimizations

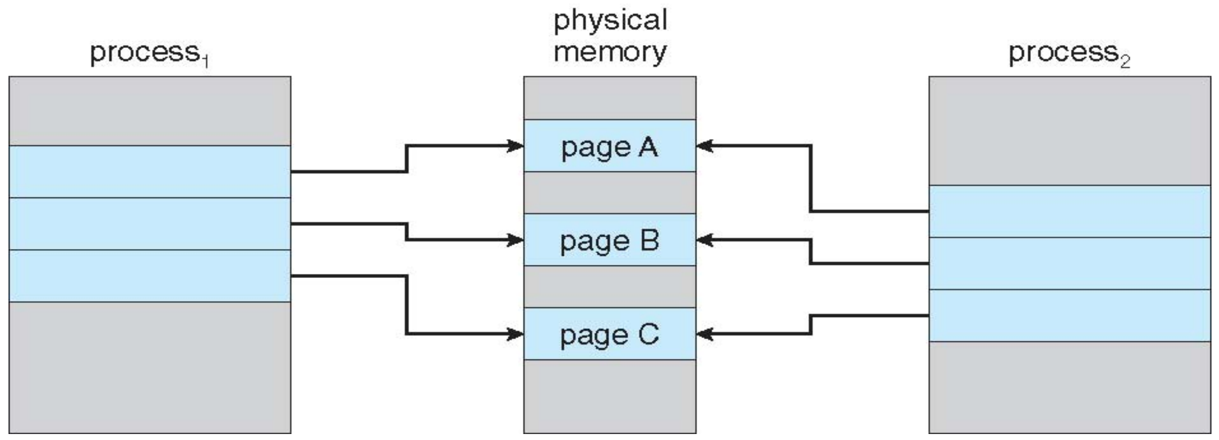
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
- For dynamic memory (heap and stack of a process), swap space–**anonymous memory**--cannot be optimized as above...

Copy-on-Write

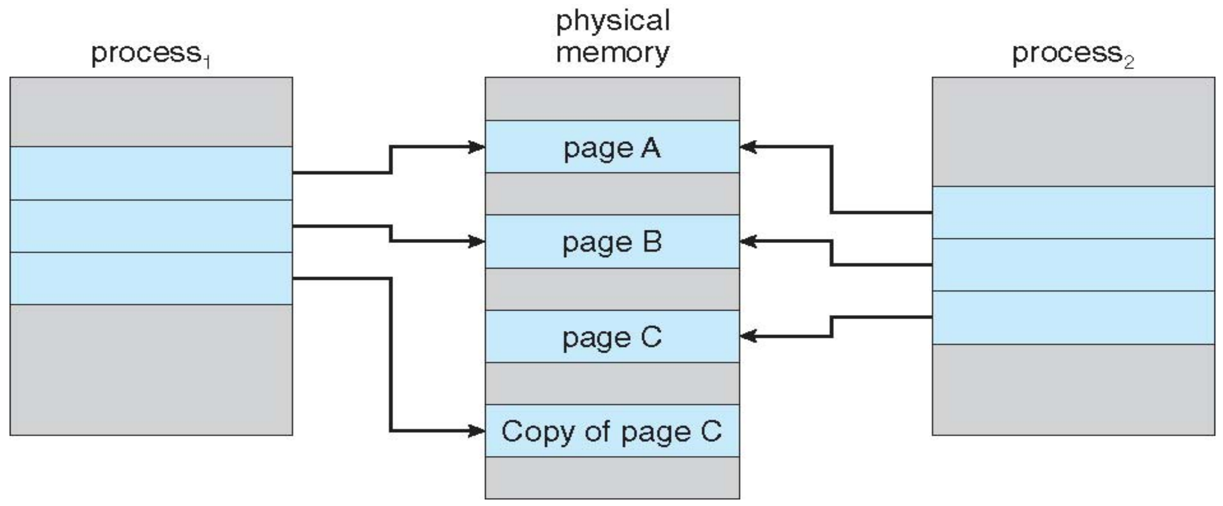
- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
  - e.g., shared code pages do not need to be copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - zero-out a page before allocating it erases all its content
- `vfork()` variation on `fork():` system call has parent suspended and child using address space of parent
  - Designed to have child call `exec()`
  - Very efficient
  - Must not modify the parent's address space

COW: Process 1 Modifies Page C

before modification of page C by process 1



after modification



What Happens if There is no Free Frames?

- Assume all frames are used up by process pages
- Frames are also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each process?

- Strategies:
  - terminate the process (but user will not be happy)
  - process swap out and all its frames are now available
  - page replacement

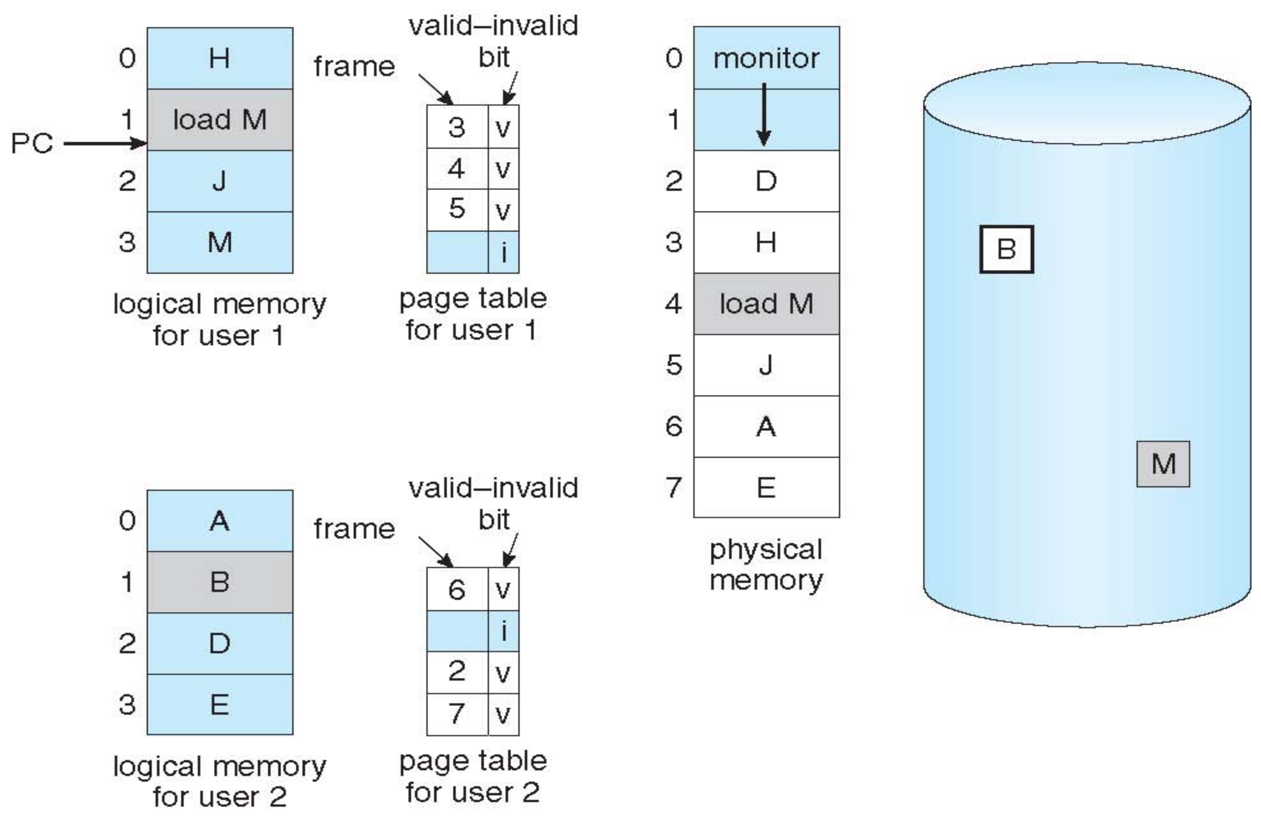
- Page replacement – find some page in memory, but not really in use, and page it out
  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- **modify (dirty) bit** indicates if a page/frame has been modified by writing to it
  - used to reduce overhead of page transfers – only modified pages are written back to disk (similar strategy for read-only pages)
- 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

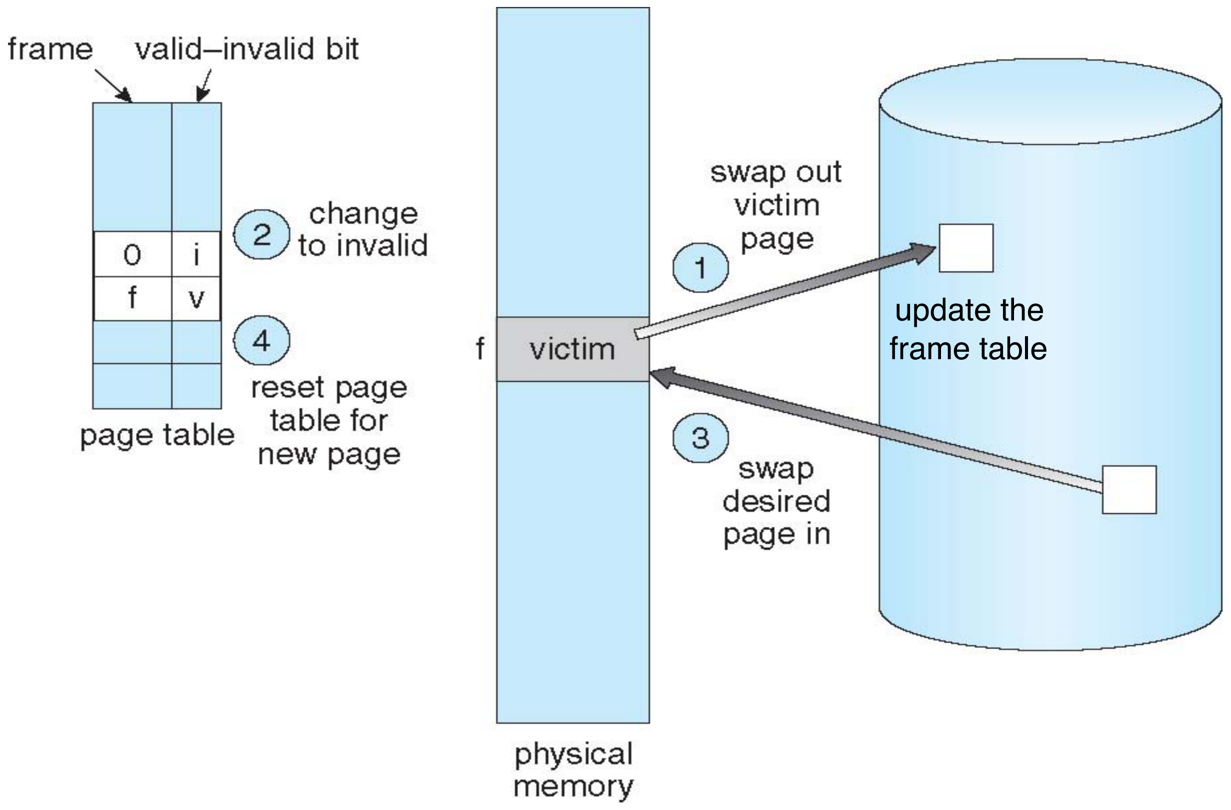
2. Find a free frame:
    1. If there is a free frame, use it
    2. If there is no free frame, use a page replacement algorithm to select a **victim frame**, write victim frame to disk if marked as *dirty*, and update page and frame tables

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

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

Note now potentially **two** 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** wants
  - lowest page-fault rate on both first access and re-access

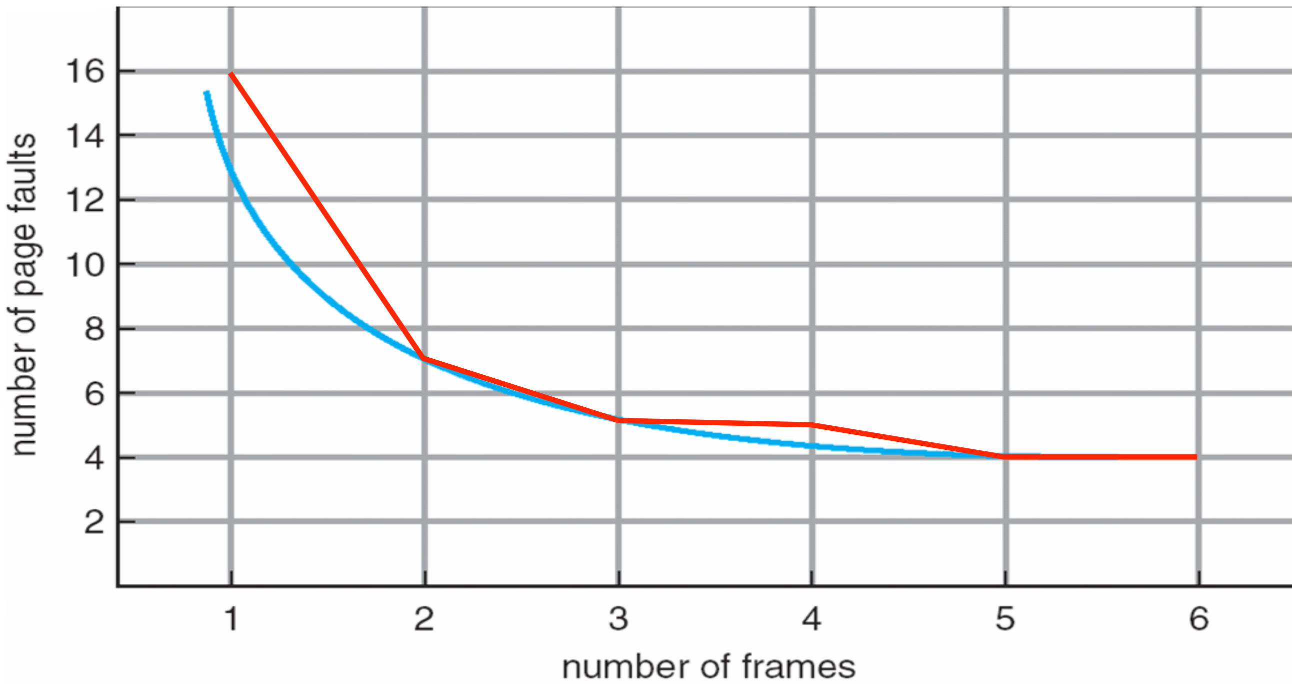
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that (cleaned up) string, i.e.,
  - String is just a series of page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault

- In all our examples, next, the reference string is

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

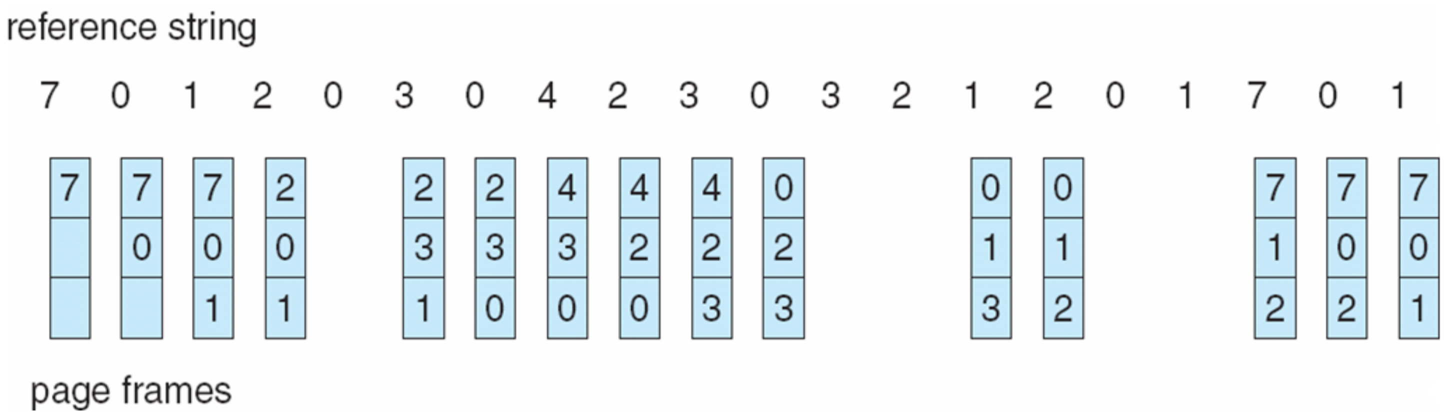
Graph of Page Faults vs. Number of Frames

General expected shape for the curve



First-In-First-Out (FIFO) Algorithm

- Give an age to each page, and remove the oldest page
  - Just use a FIFO queue
- 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) and FIFO replacement

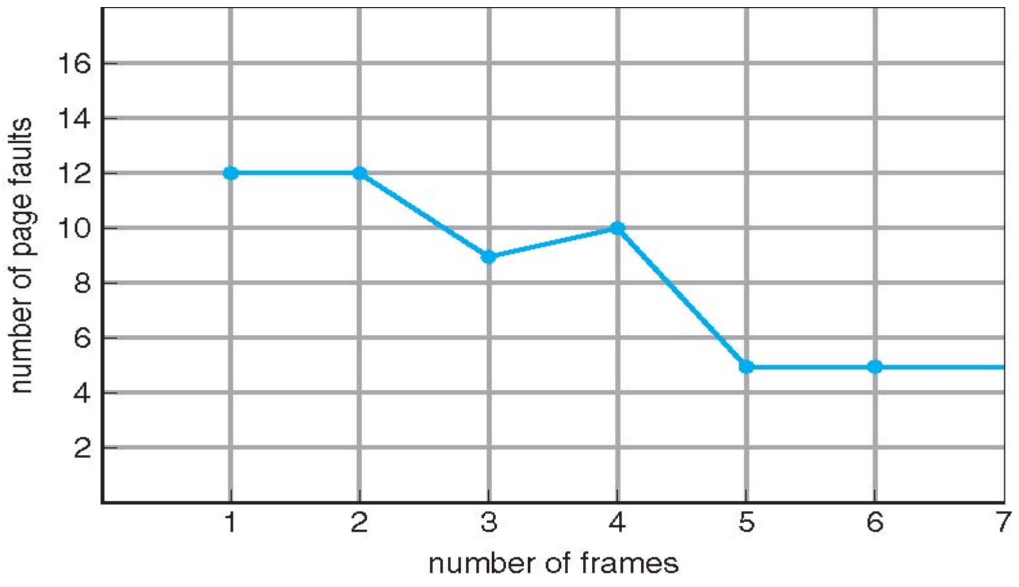


- The reference string results in 15 page faults

- Easy to code, but not always a good replacement strategy
  - a variable used constantly could have its page replaced regularly because it becomes the oldest regularly

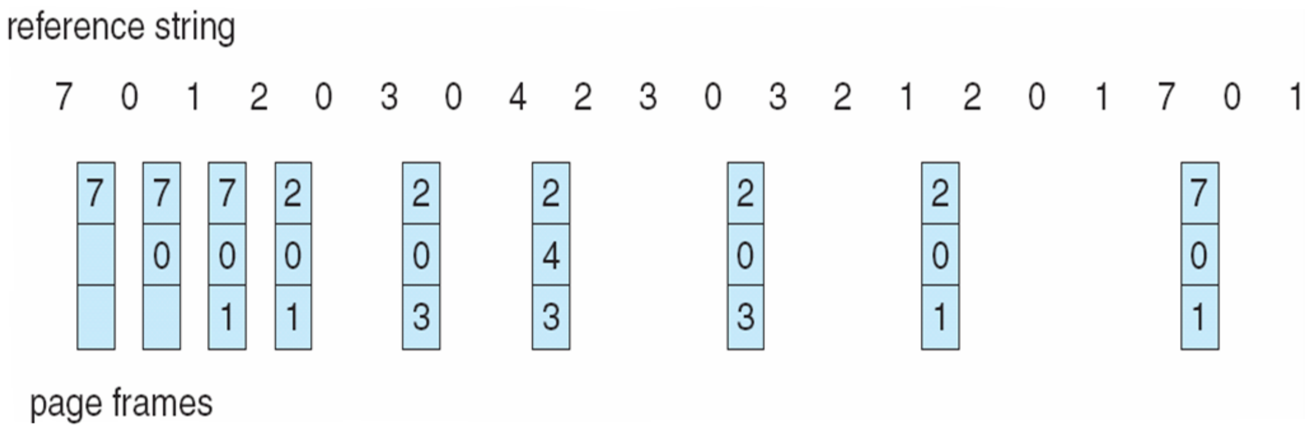
First-In-First-Out (FIFO) Algorithm

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - While adding more frames tend to diminish the page fault rate, sometimes, adding more frames maycause more page faults! This is called the **Belady's anomaly**



Optimal Algorithm

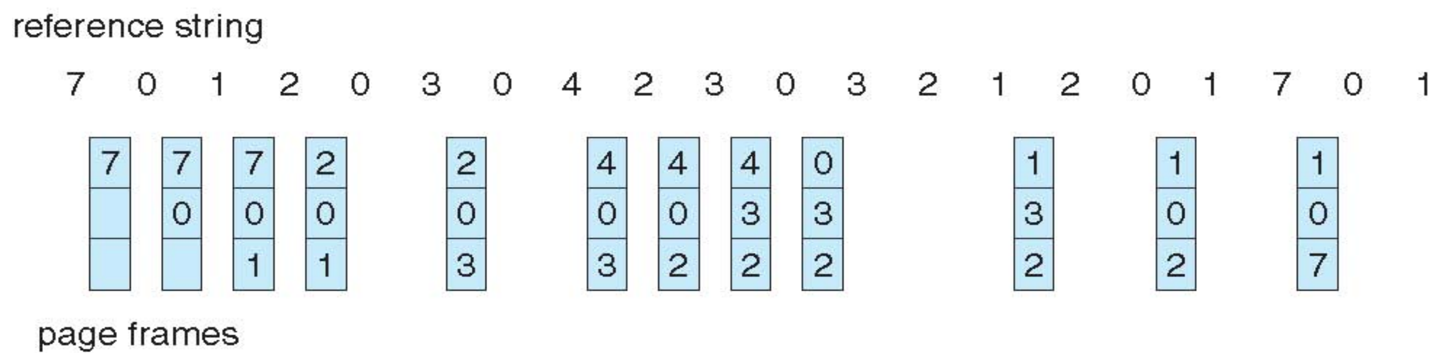
- Replace page that will not be used for longest period of time
  - 9 is optimal for our reference string example



- How do you know this?
  - Cannot read the future
- The optimal algorithm (OPT) is used for measuring how well an 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 (LRU)
- Associate time of last use with each page



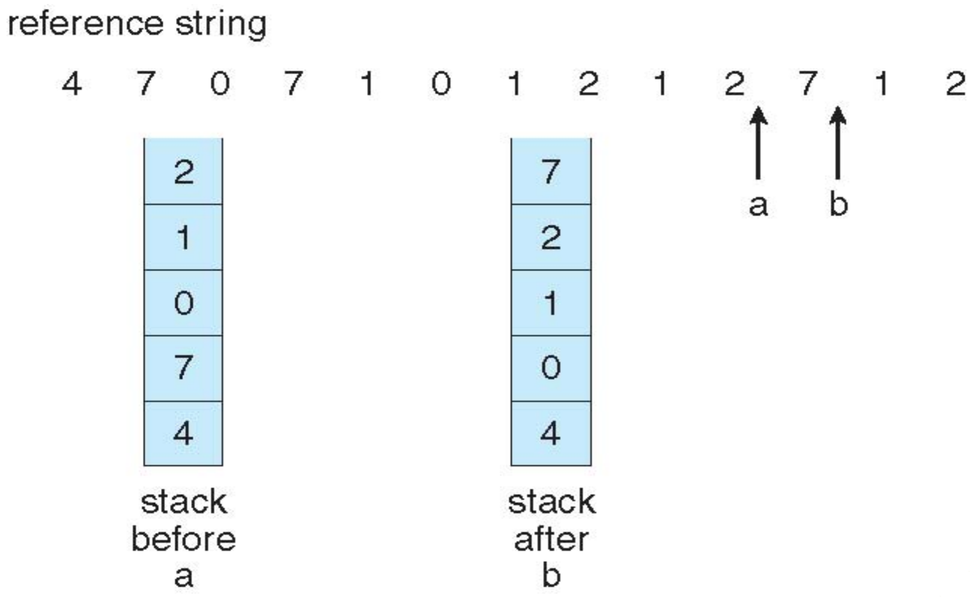
- 12 faults – better than FIFO (15) but worse than OPT (9)
- Generally LRU is a good algorithm and frequently used
- LRU and OPT are cases of **stack algorithms** which do not exhibit Belady's anomaly
- Both algorithms require hardware support
- But the difficulty lies in its implementation (with counter or stack)

LRU Algorithm - Counter

- Every page entry has a counter
- Every time a page is referenced through this entry, copy the clock into the counter
- When a page needs to be replaced, look at the counters to find smallest value
  - search through table needed
  - must write counter to memory
  - be aware of clock overflow

LRU Algorithm - Stack

- Keep a stack of page numbers in a double-link list
- Page referenced:
  - move it to the top
  - requires 6 pointers to be changed
- Each update is more expensive
- No search for replacement



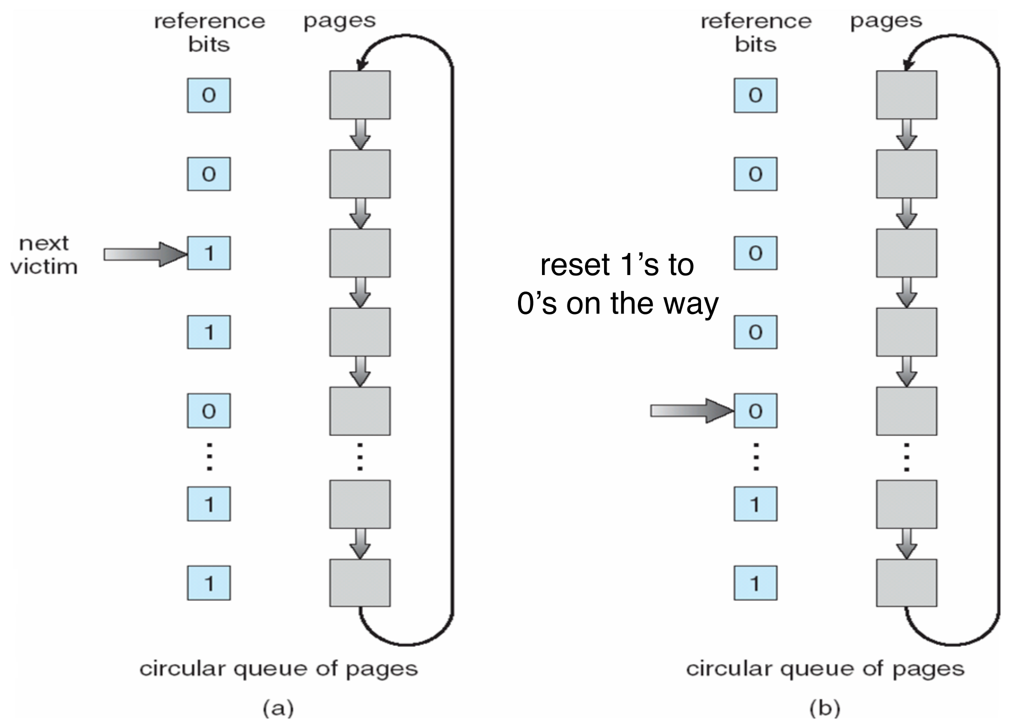
LRU Approximation Algorithms

- LRU needs special hardware (often not provided) and it is still slow
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced, bit is set to 1
  - Replace any frame with reference bit = 0 (if one exists)
    - We do not know the order, however
- Additional-reference-bits
  - Similar as reference bit, but every 100msec, OS right-shifts bits in reference **byte**
    - lowest bit becomes the leftmost bit, second-lowest is discarded
  - Lowest number page is our victim
  - Since pages with same number form a category of victims, the victim could also be chosen in FIFO among pages with same value

LRU Approximation Algorithms

- **Second-chance algorithm**
  - Similar to a FIFO algorithm, plus hardware-provided reference bit
  - Also known as **Clock** algorithm
  - If page to be replaced has
    - reference bit = 0 then replace the page
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory for the next round
      - replace next page with bit = 1, subject to same rules
  - Degenerates to FIFO if all reference bits are 0
  - Could pair up the (reference, modify) bits
    - (0,0) not referenced nor modified: best victim
    - (0,1) not referenced but modified: not great because will need to write it to disk
    - (1,0) referenced but not modified: could be used again
    - (1,1) referenced and modified: worst victim

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm** (least frequently used): replaces page with smallest count
  - but if a page is referenced a lot at the beginning, it will always remain there
  - could use a right-shift strategy to diminish the counter
- **MFU Algorithm** (most frequently used): replaces page with largest count
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither of these algorithms approximate well OPT, they are costly, and not used much

Page-Buffering Algorithms

- Keep a pool of free frames
  - A frame is available when needed, not found at page fault time
  - Read page into free frame, select victim frame to evict, and add it to free pool
  - When convenient, evict victim to disk
- Possibly, keep list of modified pages
  - When backing store is 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 if it is still in the pool of free frames
  - Generally useful to reduce penalty if wrong victim frame selected (e.g., with FIFO replacement)

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – e.g., 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 given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.

Allocation of Frames

- Each process needs a *minimum* number of frames, in function of the design of its set of instructions
  - indirection in some instructions requires potentially more pages
- Example: 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*
- *Maximum* is of course the total number of frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

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
  - But a small process will not need some frames while large one might not operate well
- Proportional allocation – Allocate according to the size of process

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

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

  - Dynamic as degree of multiprogramming, process sizes change
  - Can also use proportion according to various priorities

Global vs. Local Allocation

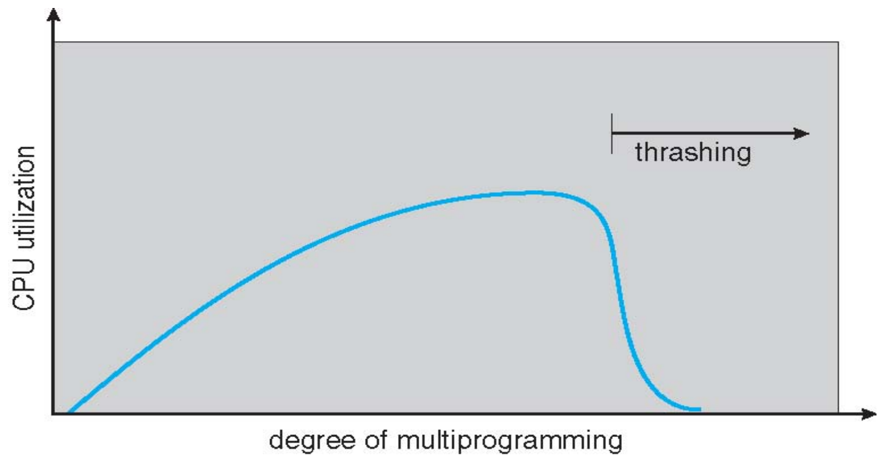
- If process P_i generates a page fault
  - select for replacement one of its own frames
  - select for replacement a frame from a process with lower priority number
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly from one run to another
  - 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

Non-Uniform Memory Access

- So far we considered all memory accesses takes an equal time
- Many systems are NUMA – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- 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
    - When possible, schedule all threads of a process and allocate all memory for that process within the lgroup
    - Otherwise, use an lgroup that is “closer” in latency

Thrashing

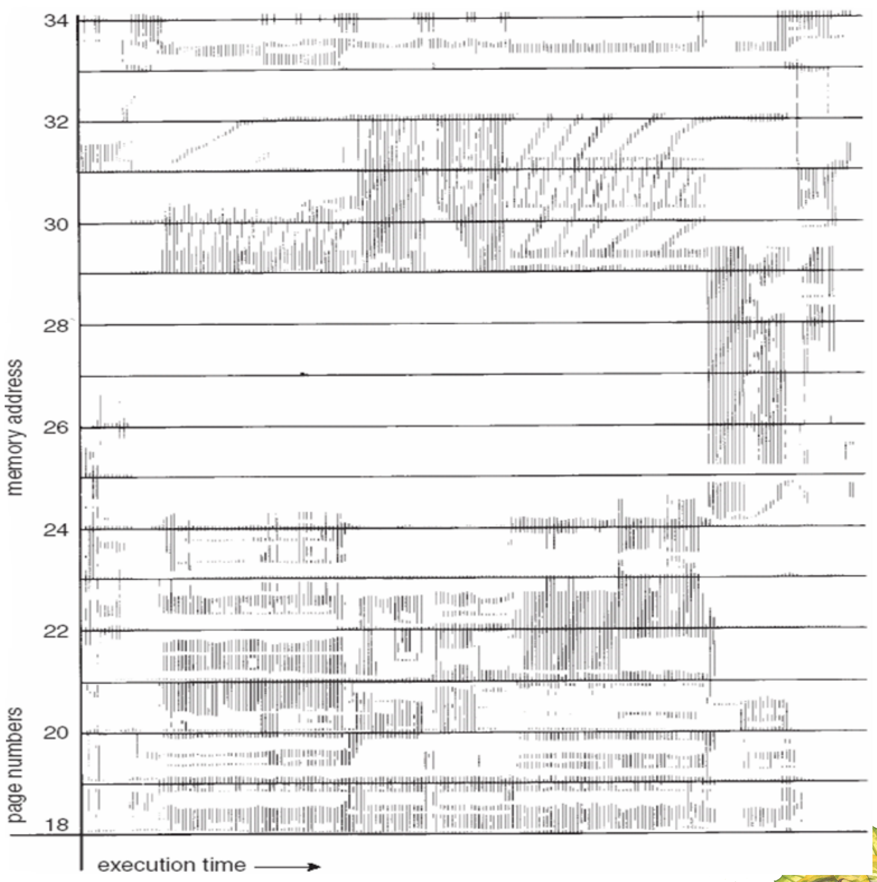
- If a process does not have "enough" pages, its page fault rate can become very high
  - Page fault to get page
  - Replace existing frame
  - But quickly needs the replaced frame back
  - This leads to:
    - Low CPU utilization
    - OS thinks that it should increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out (i.e., spends more time paging than executing)



Demand Paging and Thrashing

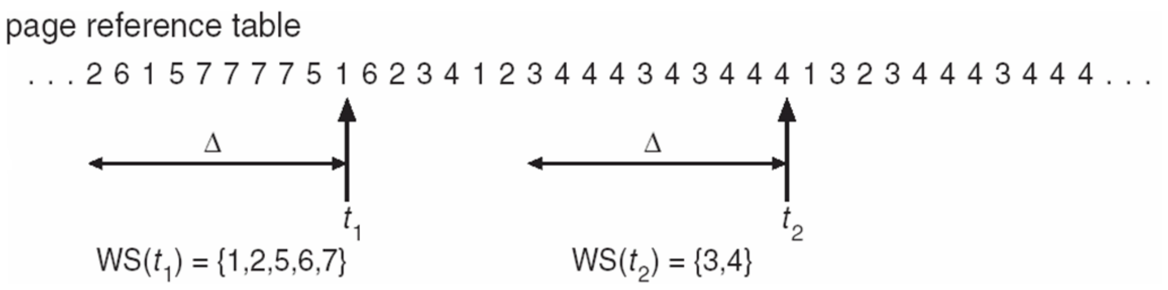
- Demand paging works because of the **locality model**
  - Process under execution migrates from one locality to another (e.g., function call and its local variables)
  - Localities may overlap
  - Locality might return
- Thrashing occurs when the sum of the size for locality > allocated memory size
  - Can limit impacts of thrashing by using local replacement or priority page replacement

Memory-reference locality pattern



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, it will not encompass entire locality
  - if Δ too large, it will encompass several localities
  - if $\Delta = \infty$, it will encompass entire program



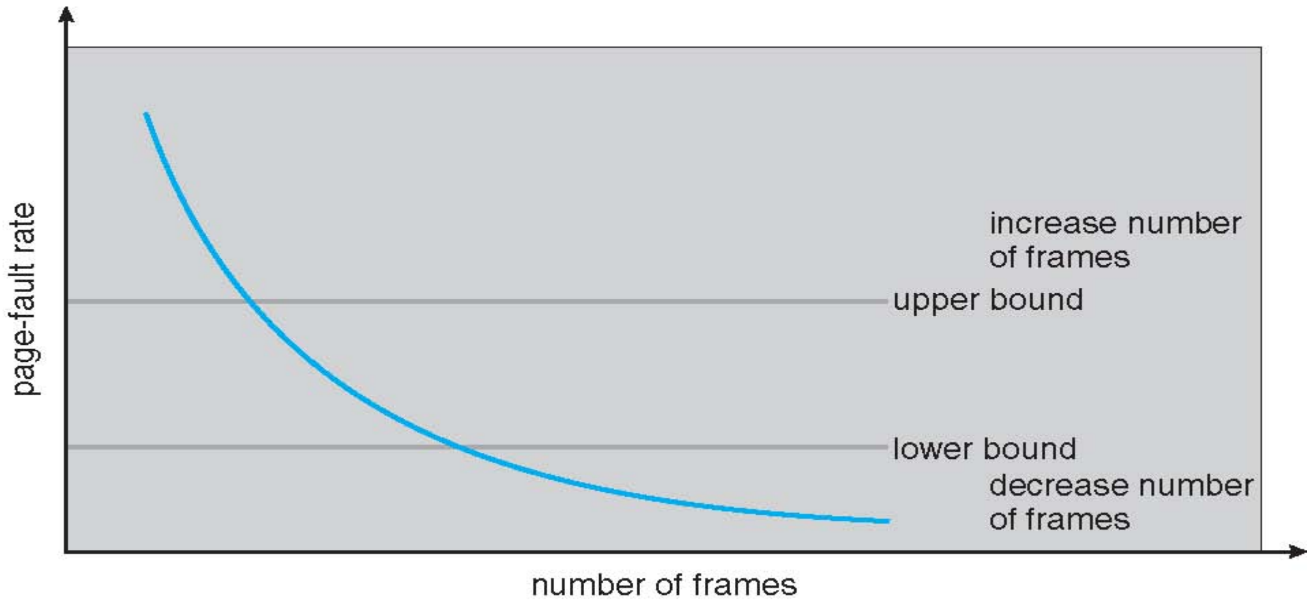
- $D = \Sigma\ WSS_i \equiv$ total demand of frames
  - Approximation of locality
- if $D > m$ (total number of frames), thrashing : then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5,000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts, copy in memory all reference bits, and resets their to 0
  - If one of the bits in memory = 1, then the page is in working set
- Inaccurate because we do not know when within the 5,000 time units the page was referenced
- Improvement = 10 bits and interrupt every 1000 time units, but more costly to handle

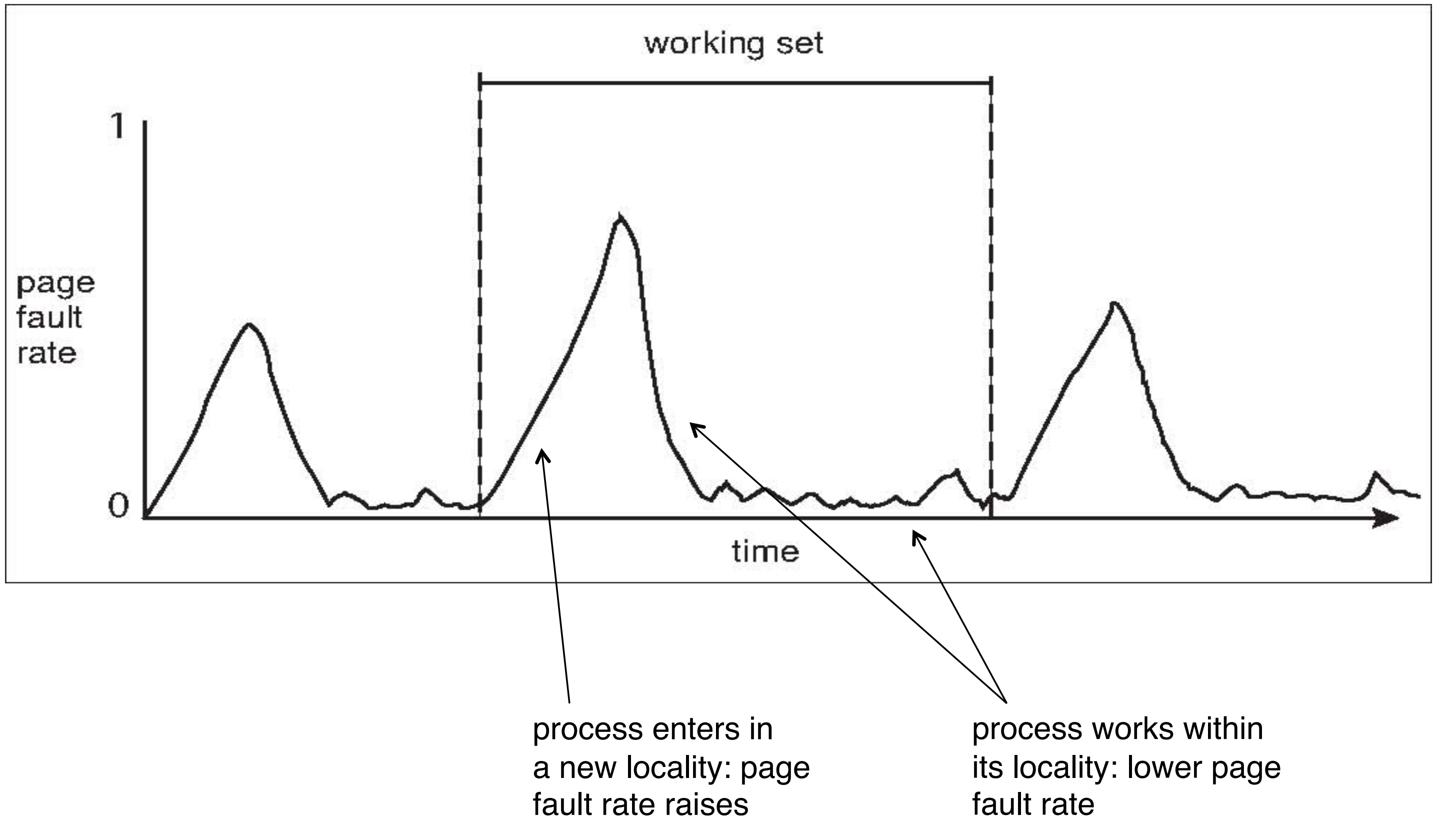
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

Typical behavior of a working set with respect to page fault rate



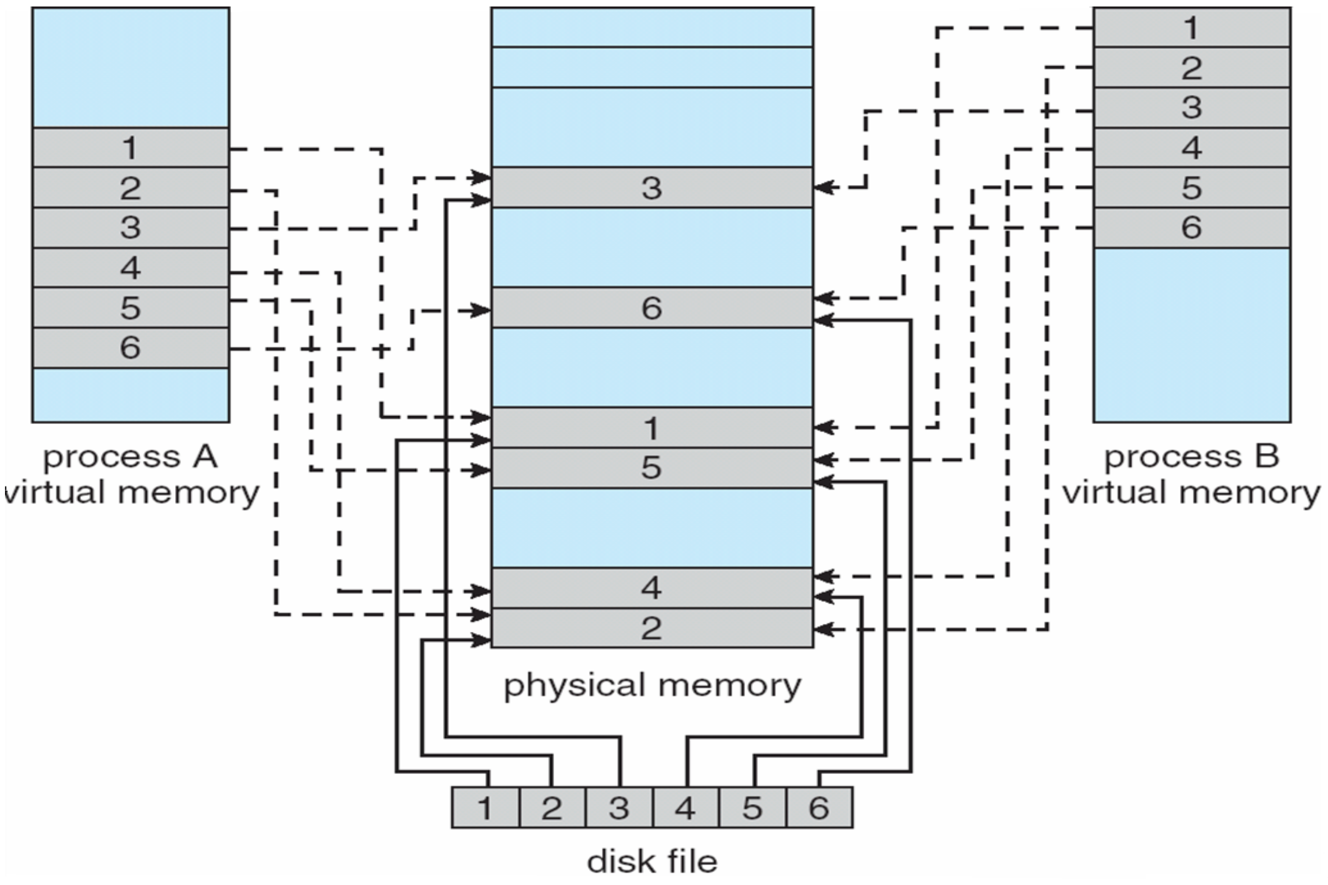
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially 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
- Simplifies and speeds up file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- The written data make it to disk:
  - Periodically and/or at file `close()` time
  - For example, when the pager scans for dirty pages

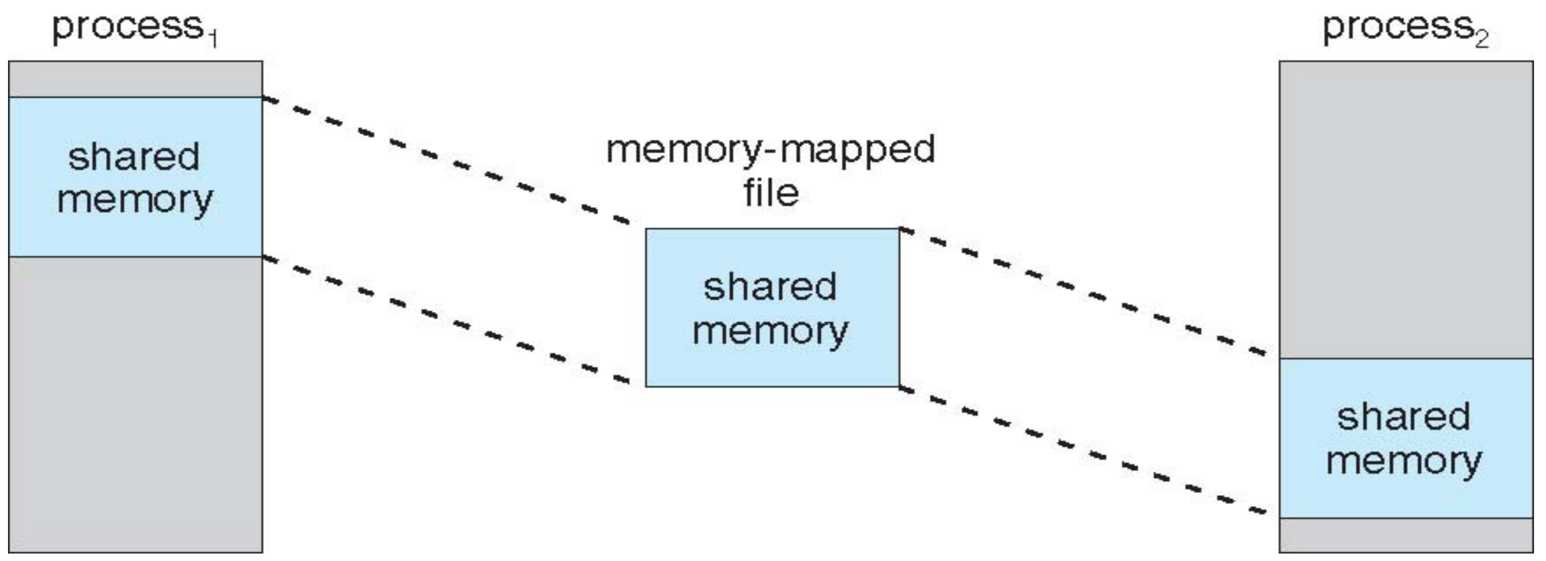
Memory-Mapped File Technique for all I/O

- Some OSes use memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
  - But map file into **kernel** address space
  - Process still does `read()` and `write()`
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory Mapped Files



Memory-Mapped Shared Memory in Windows

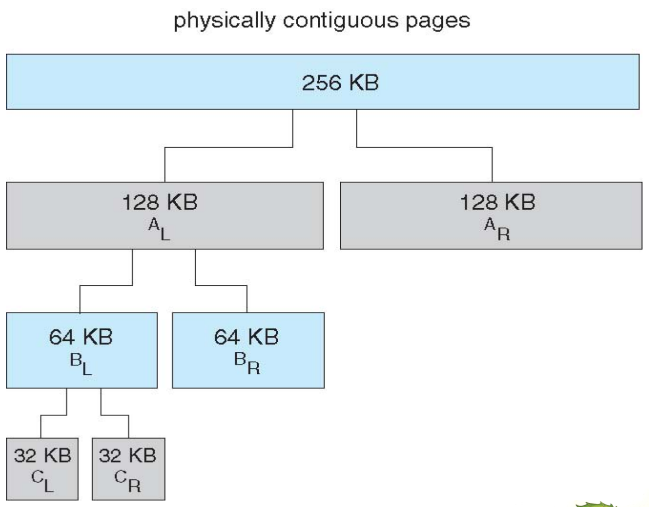


Allocating Kernel Memory

- 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

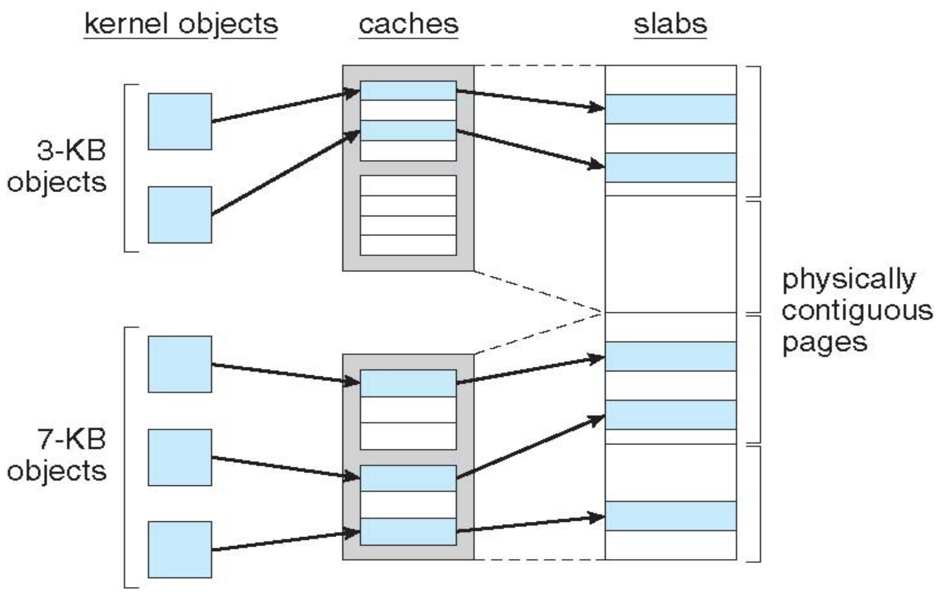
Buddy System

- 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



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, a new slab is allocated
- Benefits include no fragmentation, fast memory request satisfaction



Other Issues – Prepaging

- To reduce the large number of page faults that occur at process (re-)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 are used
  - Is cost of *s* x *α* save pages faults more or less than the cost of prepaging *s* x *(1- α)* unnecessary pages*?*
  - *α* near zero ⇒ prepaging loses
  - *α* near one ⇒ prepaging might win

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 (smaller size)
  - Page table size (larger size)
  - **Resolution** (smaller size)
  - I/O overhead (larger size)
  - Number of page faults (larger size)
  - Locality (smaller size)
  - TLB size and effectiveness (larger size)
- Page size is always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing page size 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
  - Recent trend with multiple page sizes and software-managed TLBs

Other Issues – Program Structure

- `Int[128,128] data;`
- Each row is stored in one page of 128 `Int`
- Program 1 (zeroes one entry per page at a time)

```
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 (zeroes all entries per page at a time)

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

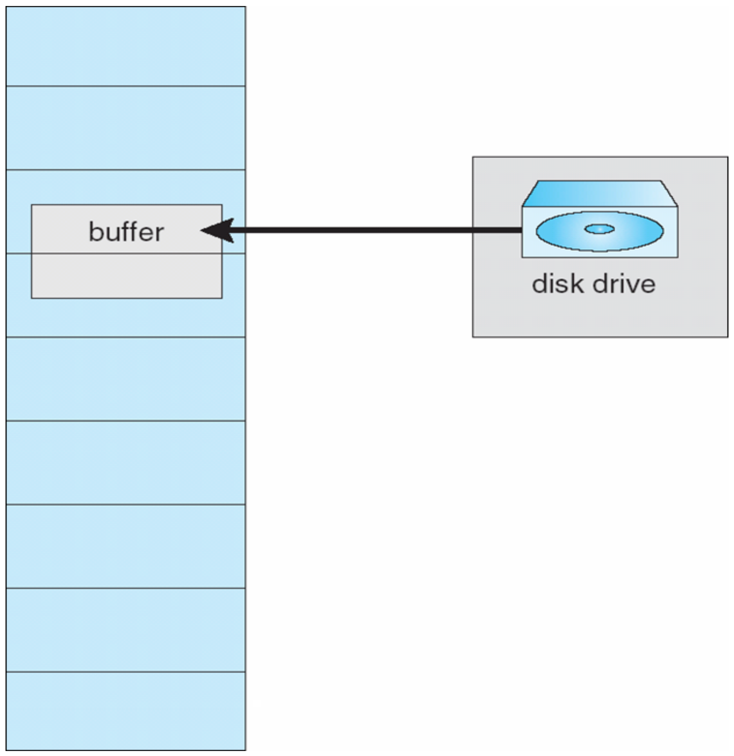
  128 page faults

- Try to exploit locality (even though it is only one aspect of efficiency), e.g.,
  - stack has good locality
  - hash table is much less local
- Compiler could generate read-only / reentrant code, avoid routine overlapping page boundaries, cluster routines that call each other many times, etc.

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
- Locking must be used with care, because bugs in user programs or even in kernel programs could have major impacts on an OS efficiency

An example of locking mechanism



Operating System Examples

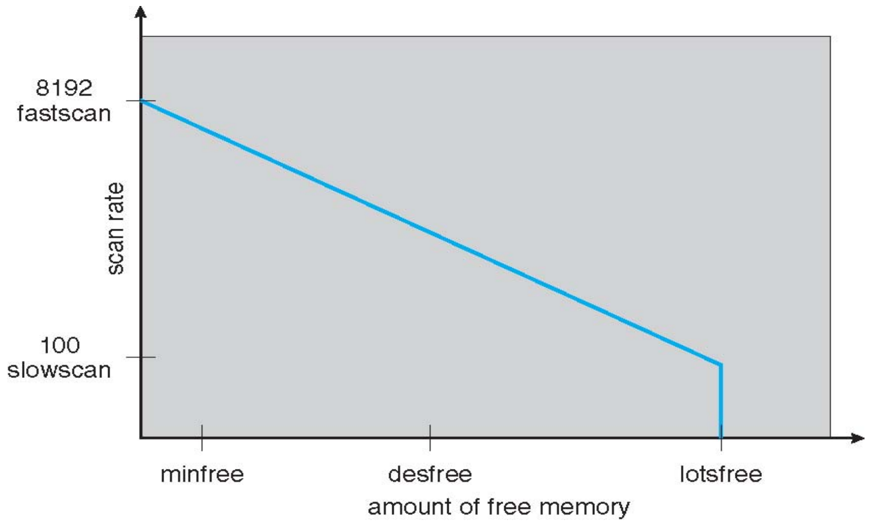
- Windows XP
- Solaris

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 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 scans pages using 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
- Priority paging gives priority to process code pages



End of Chapter 9