Virtual memory

Last modified: 04.05.2021

Contents

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

Background (Cont.)

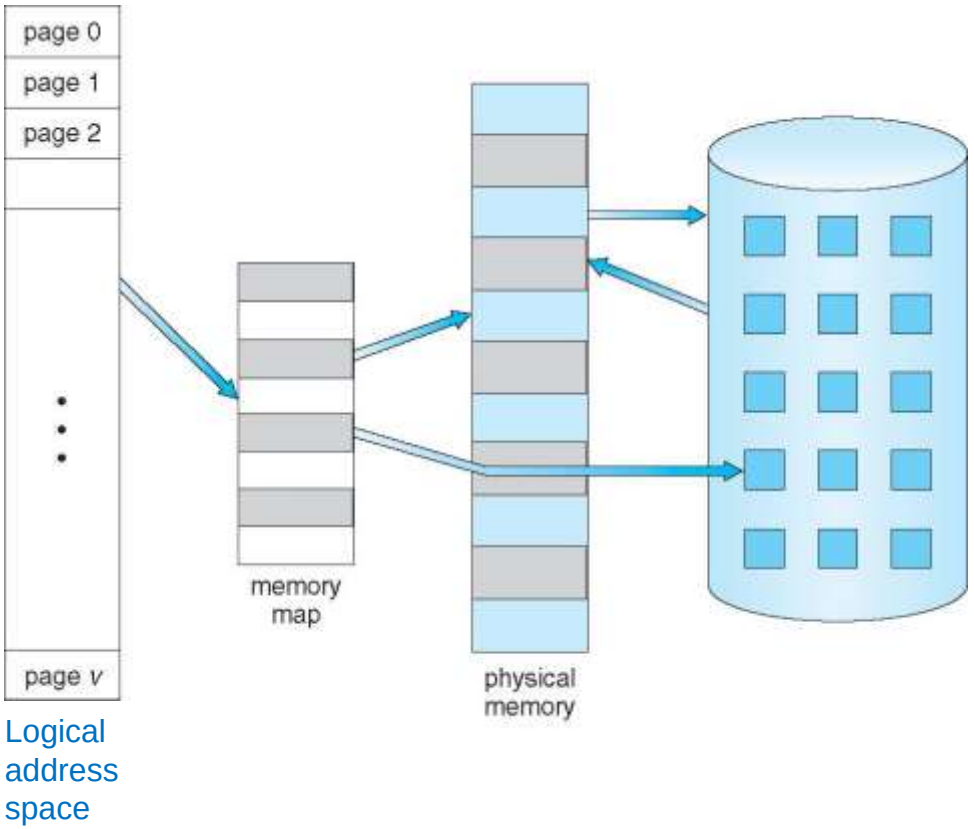
Virtual memory – separation of user logical memory from physical memory

- 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

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Background (Cont.)

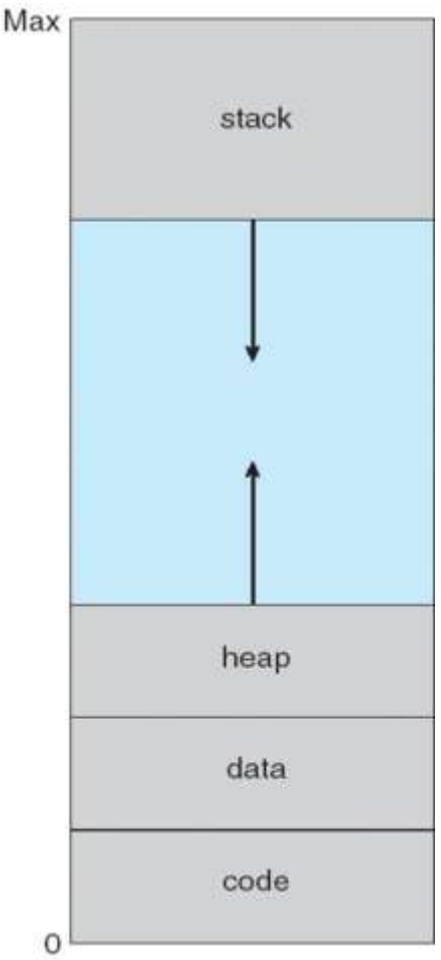
- **Virtual address space** – logical view of how process is stored in memory
  - Usually linear addressing: contiguous addresses from 0 until end of space
    An alternative addressing: segment+offset
  - Physical memory is organized in blocks
  - MMU must map logical addresses to physical



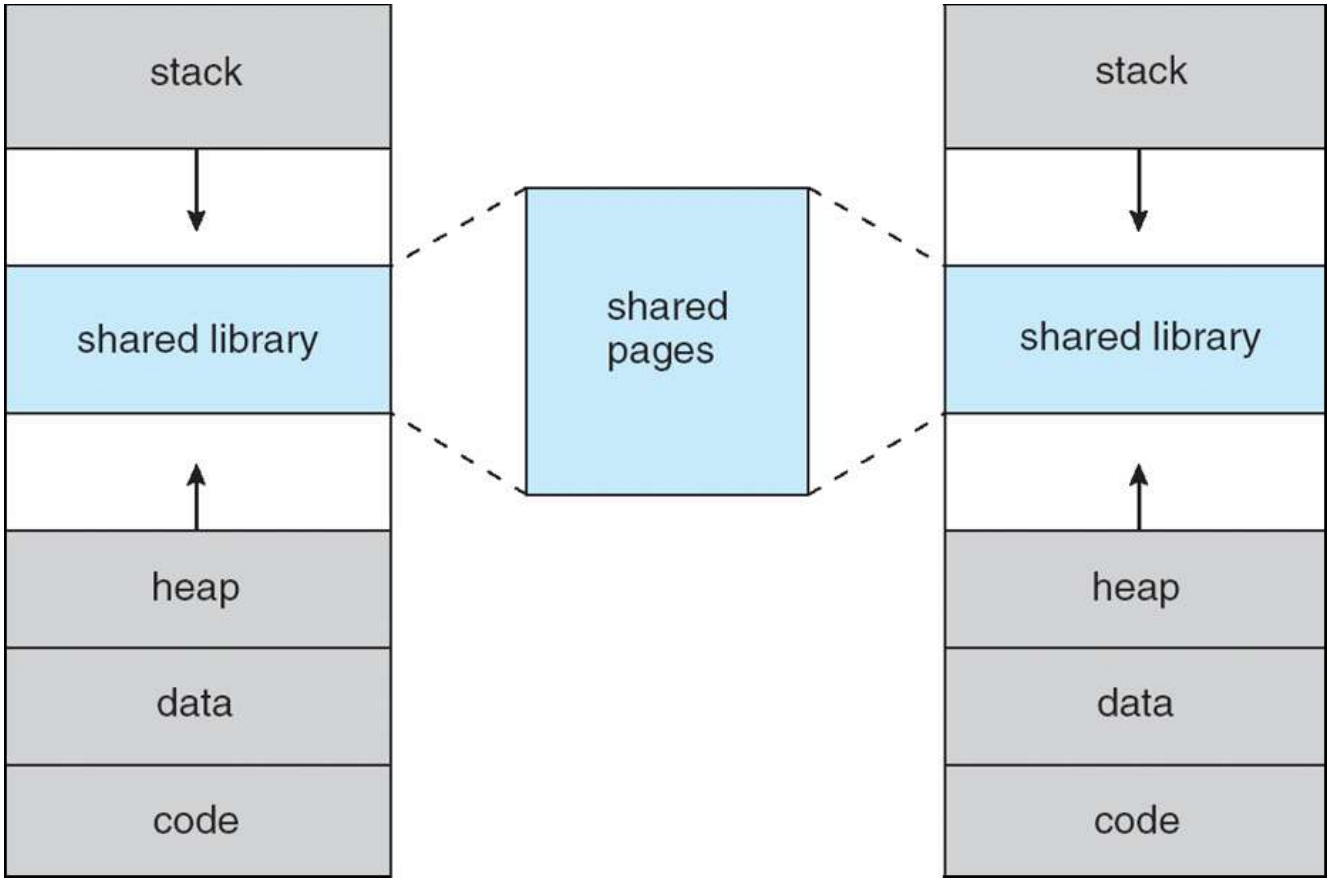
Virtual-address space

Usual design:

- logical address space for stack to start at Max logical address and grows “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
- **sparse** address space 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 (copy-on-write)

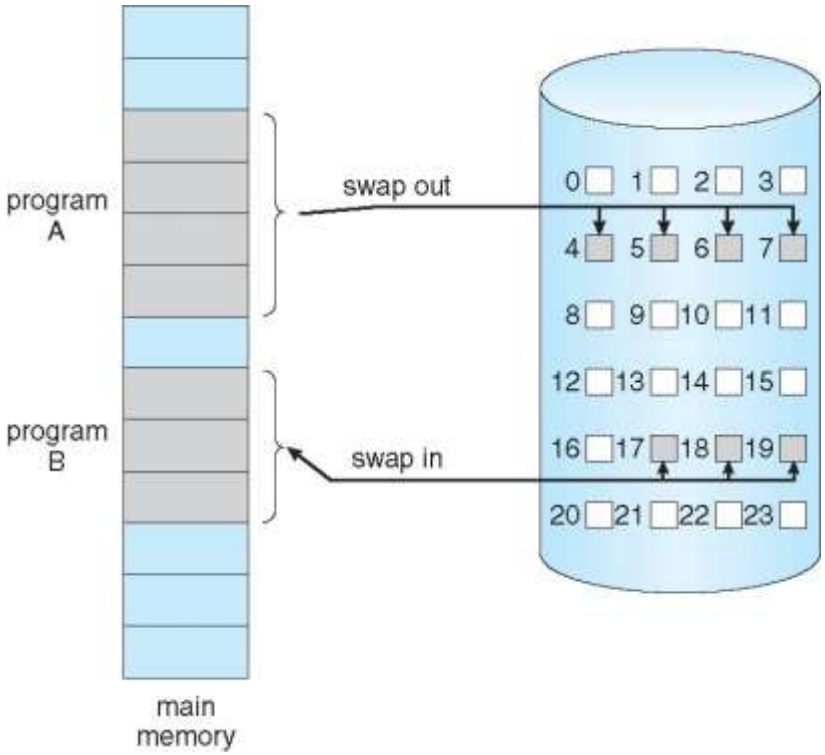


Shared Library Using Virtual Memory



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
- Similar to paging system with swapping (diagram on right)
- 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**

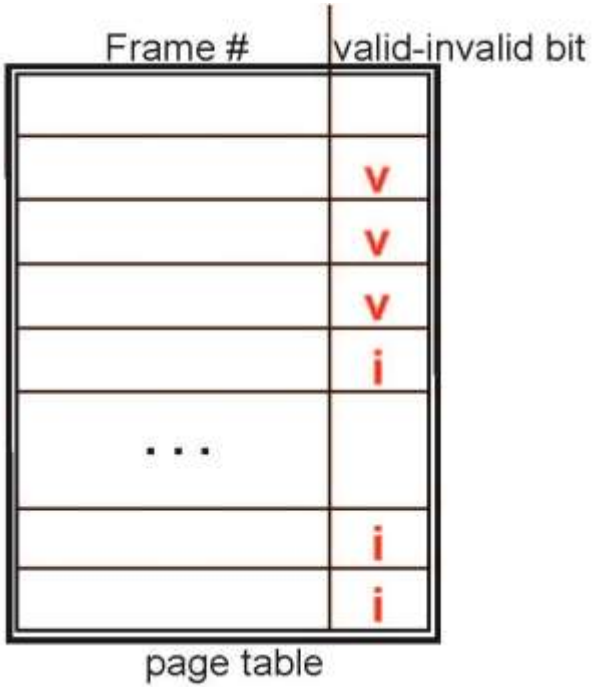


Basic Concepts

- When a process is to be swapped in the pager guesses which pages will be used and brings 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
    - Without changing program behavior
    - Without programmer needing to change code

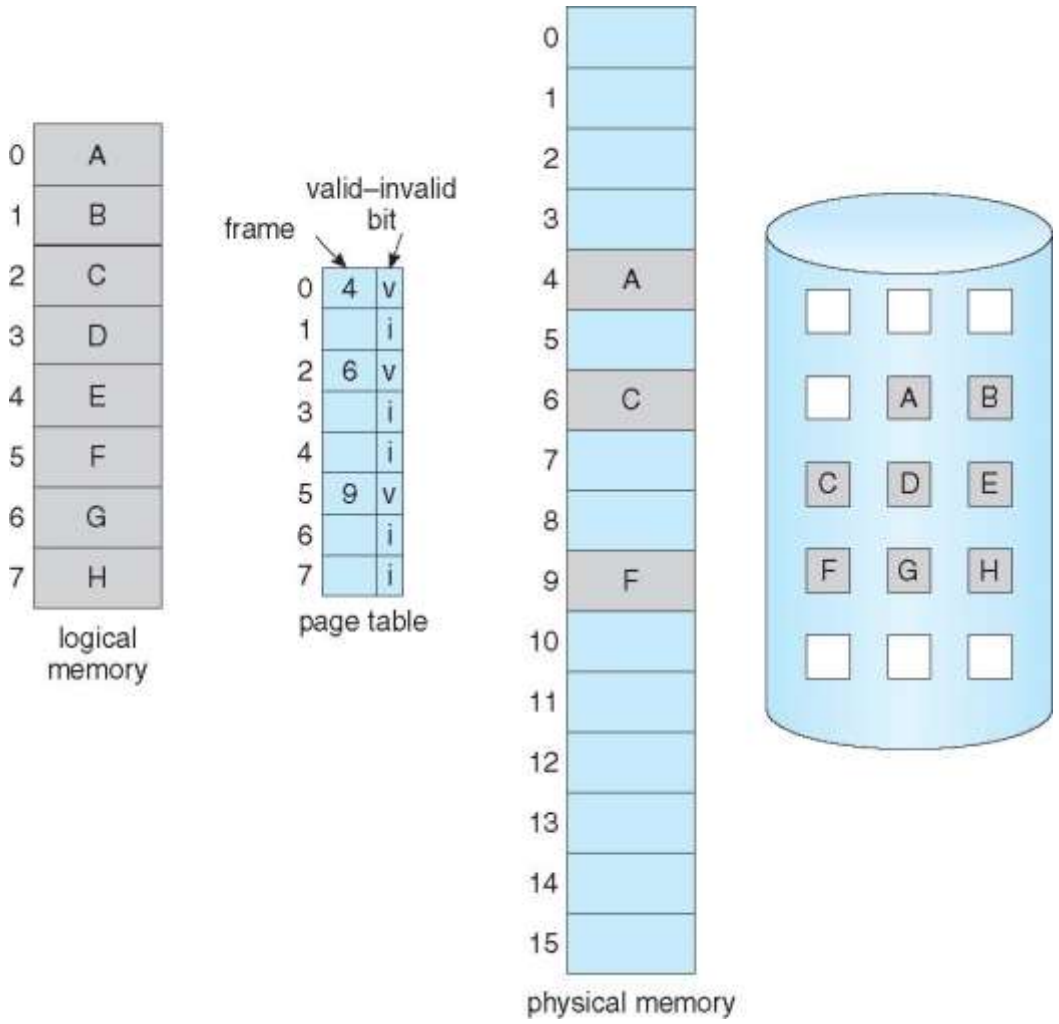
Valid-Invalid Bit

- 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:



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

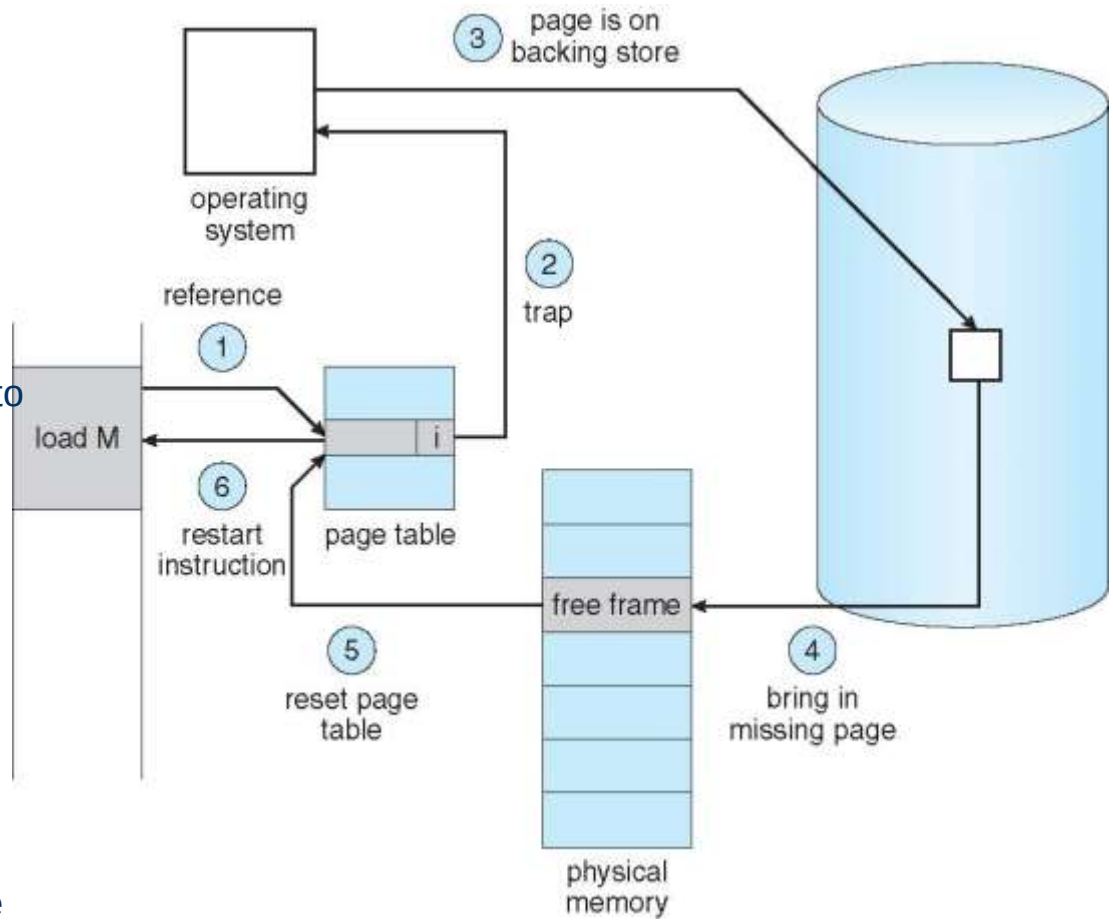
Page Table When Some Pages Are Not in Main Memory



Page Fault

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

  page fault



Handling a **page fault**

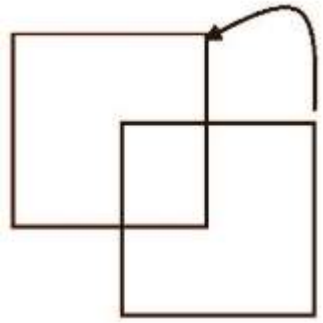
- Operating system looks at page table to decide:
  - Invalid reference ⇒ abort process
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory. Set validation bit = **v**
- **Restart** the instruction that caused the 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
- Note: Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system

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?

Worst-case performance of Demand Paging

- Stages in Demand Paging (worst 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

Average 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 $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

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

Demand Paging Example

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

  = (1 – p) x 200 + p x (8,000,000 +200)

  = 200 + p x 8,000,000 ns
- If one access out of 1,000 causes a page fault, then

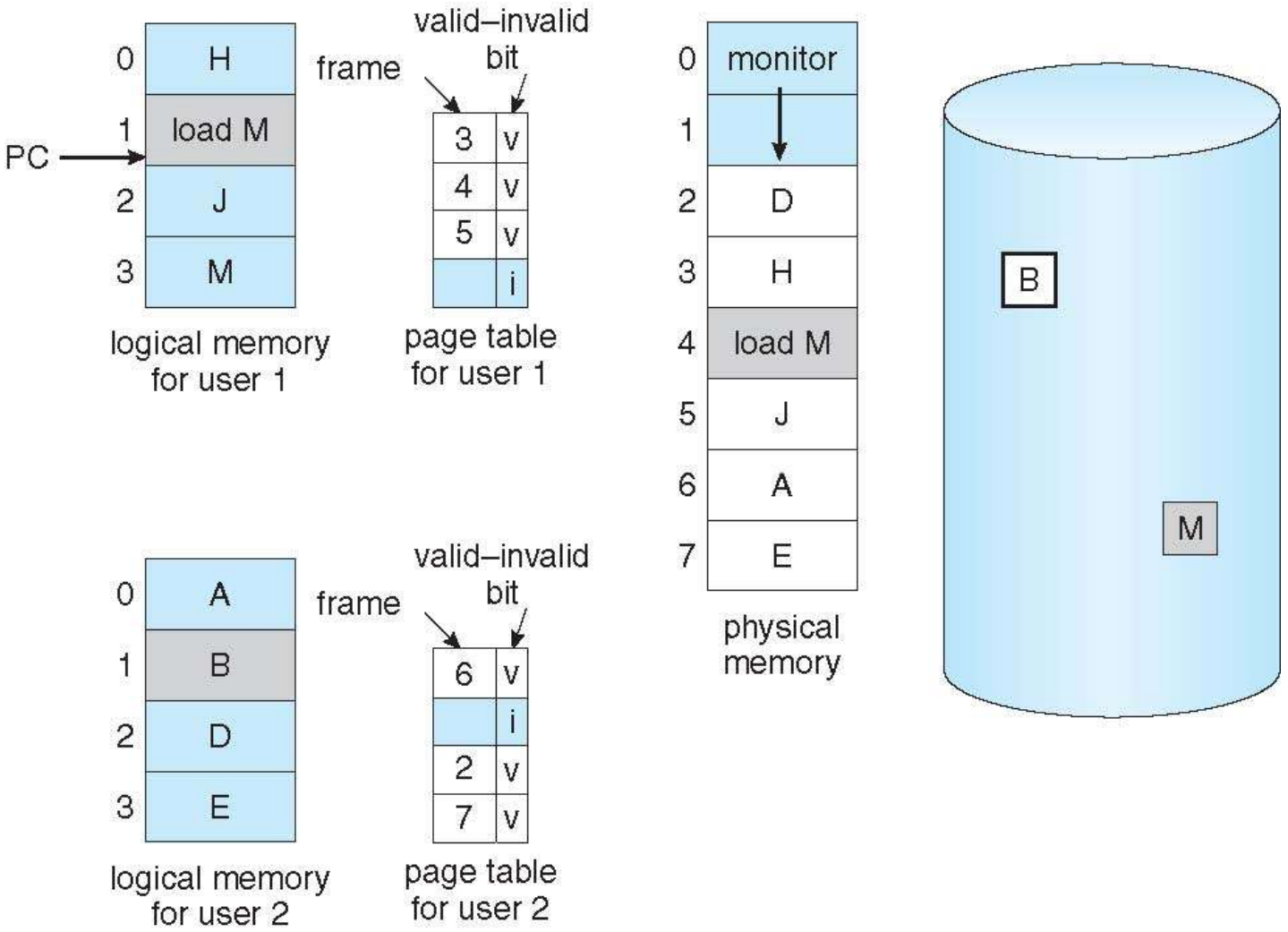
  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- For performance degradation < 10 percent
  - 220 > 200 + 8,000,000 x p
    20 > 8,000,000 x p
  - $p < 2.5*10^{-6}$
  - < one page fault in every 400,000 memory accesses
- What is the average page-fault rate (in pages/sec)?

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

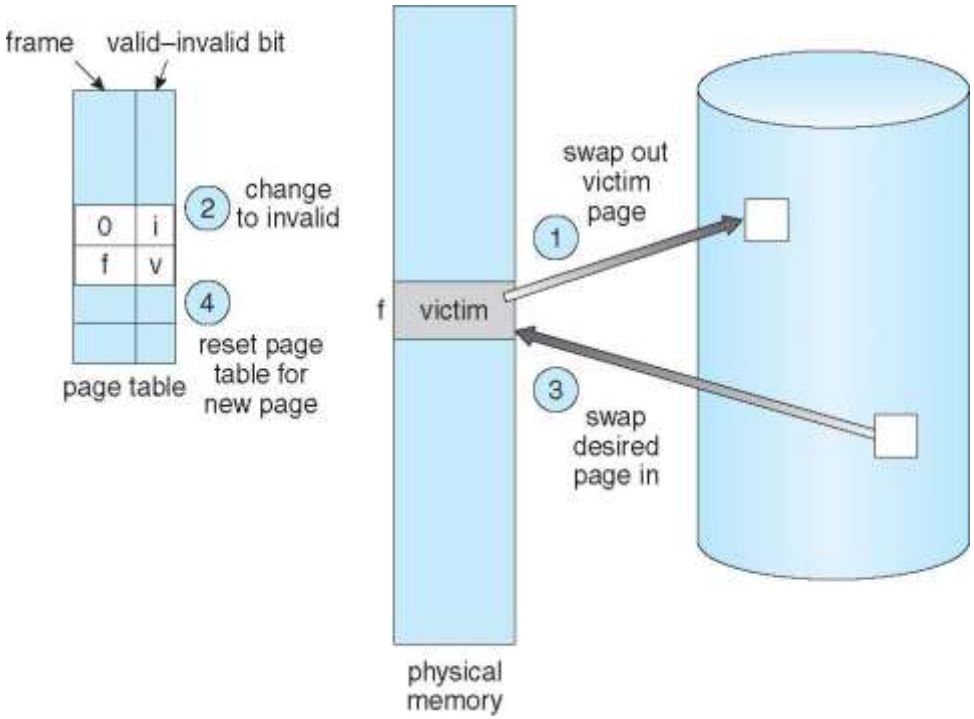
Need For Page Replacement



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 acquired) 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

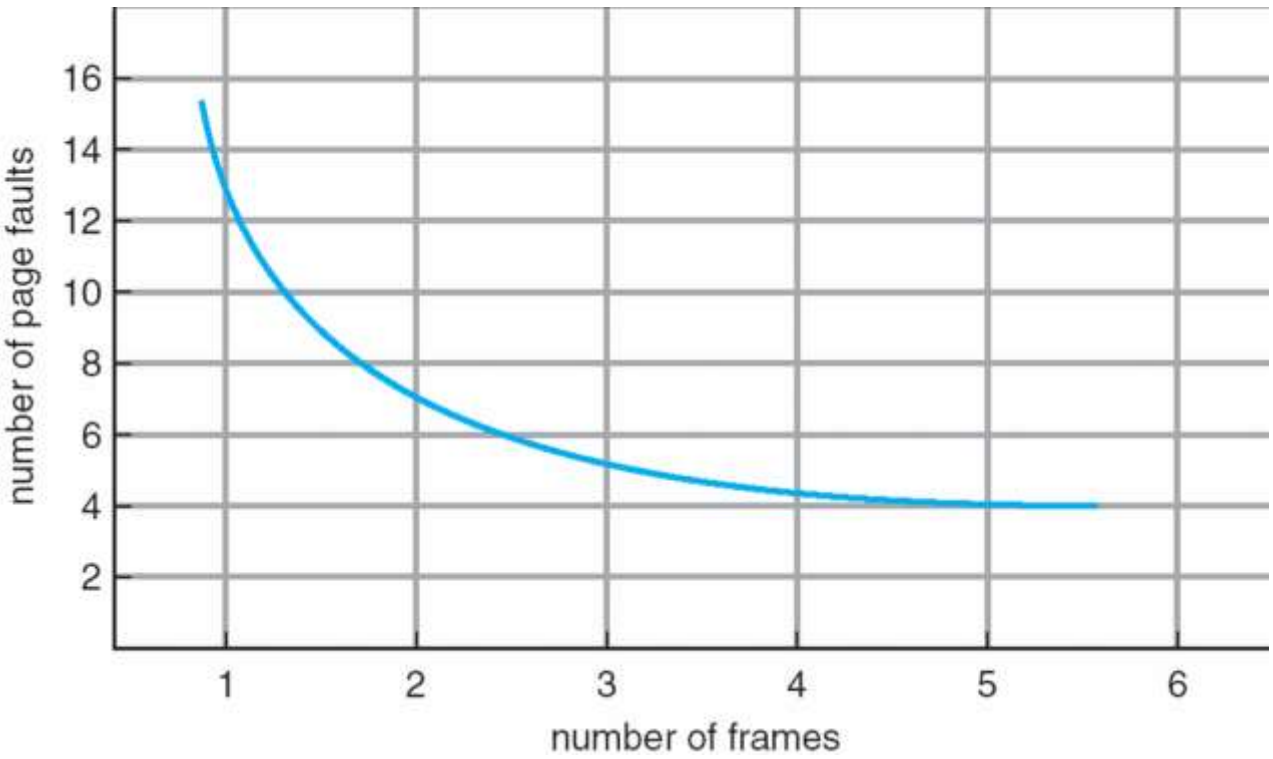


Page and Frame Replacement Algorithms

- **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

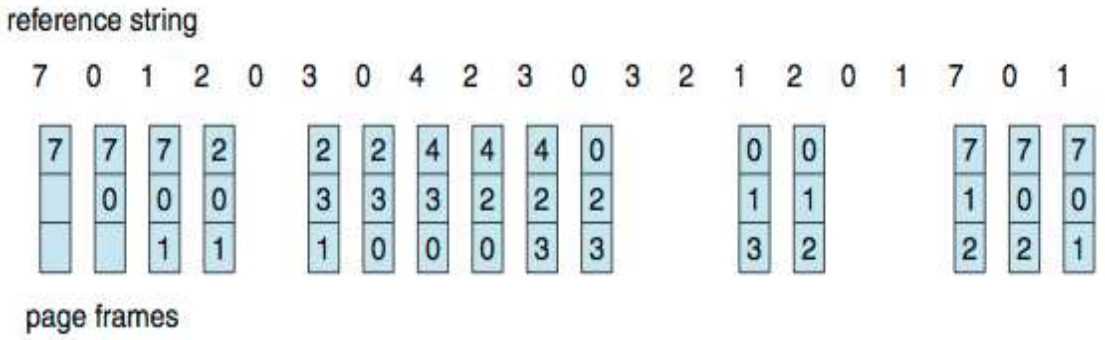
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of expected dependence of Page Faults Versus The Number of Frames



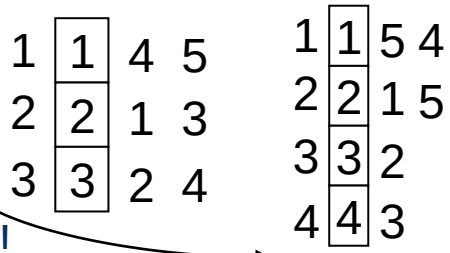
First-In-First-Out (FIFO) Algorithm

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



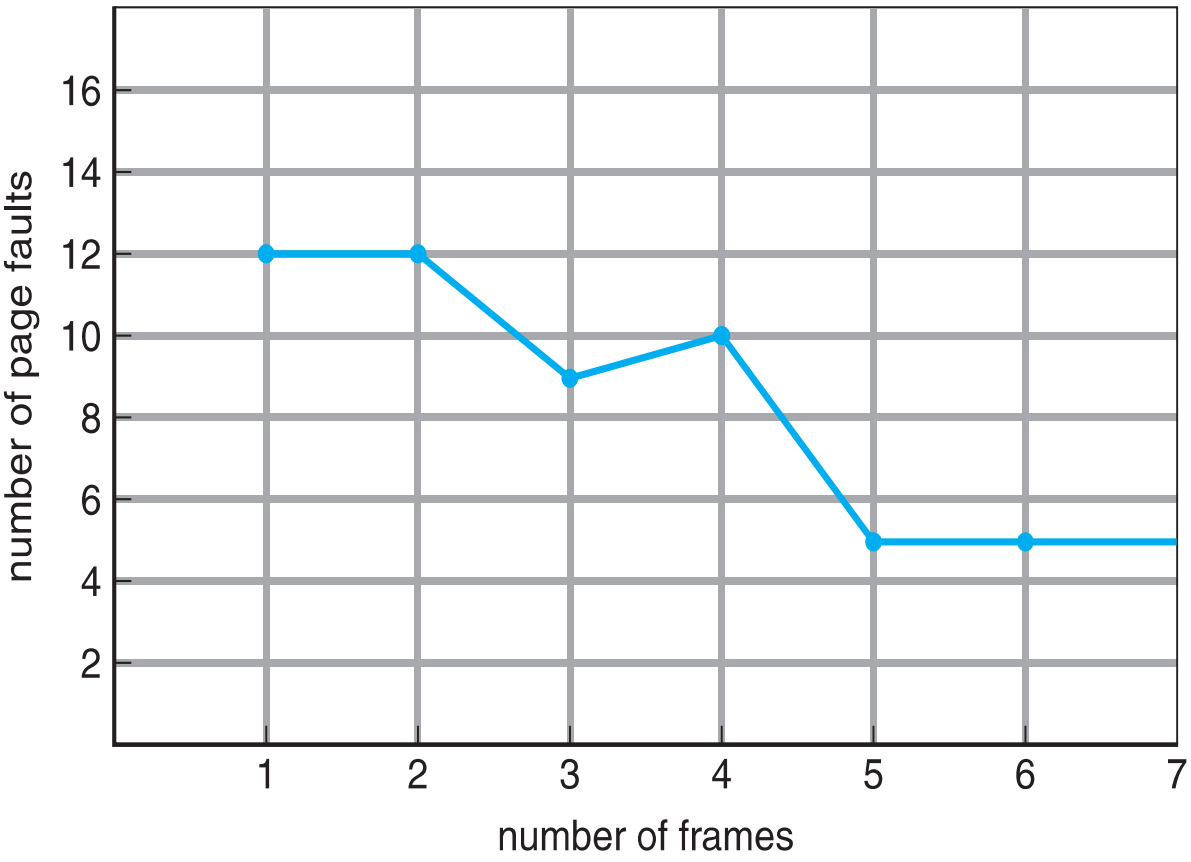
15 page faults

- 4 frames – 10 page faults
- Can vary by reference string: consider **1,2,3,4,1,2,5,1,2,3,4,5**
  - 3 frames -> 9 page faults
  - 4 frames -> 10 page faults



- Adding more frames <u>can</u> cause more page faults!
  - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

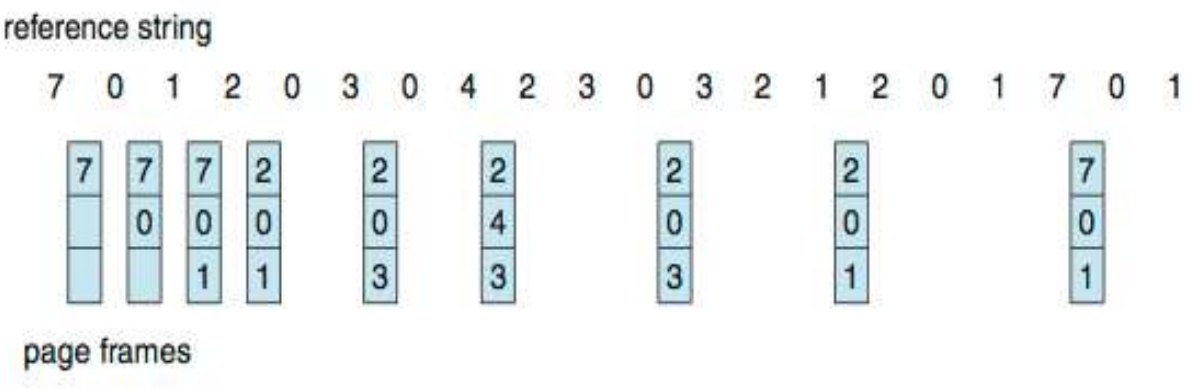
FIFO Illustrating Belady's Anomaly



Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.**

Optimal Algorithm (OPT)

- Replace page that will not be used for longest period of time
  - For 3 frames and ref. string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

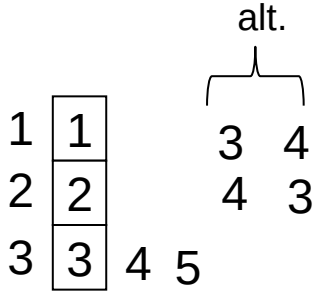


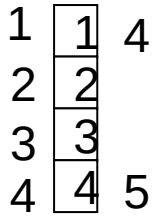
3 frames - 9 page faults

(4 frames – 8PF)

  - For the reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.**

3 frames - 7 page faults





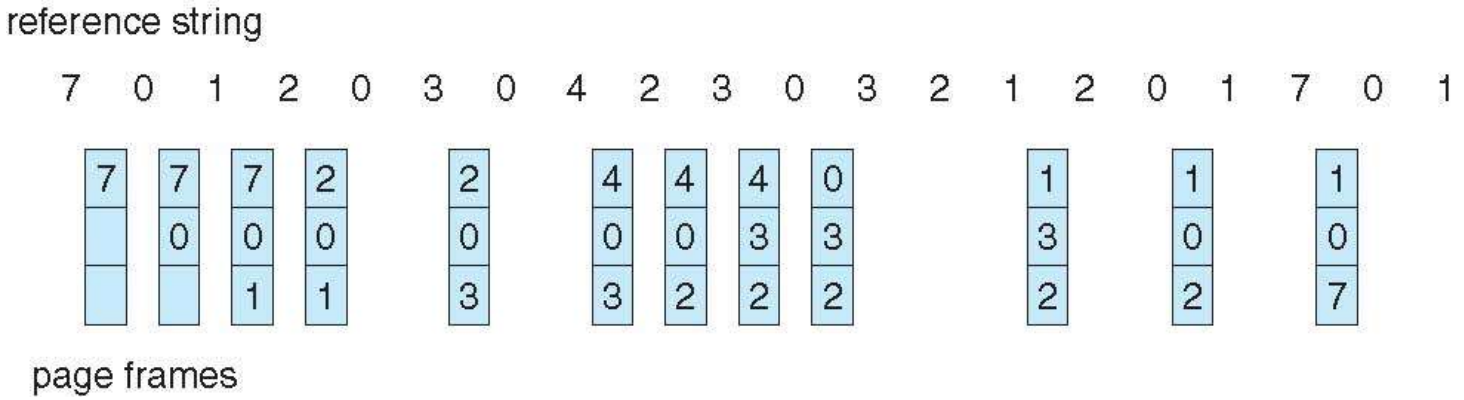
4 frames

6 page faults

- How do you know this?
  - Can't read the future
- OPT is 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

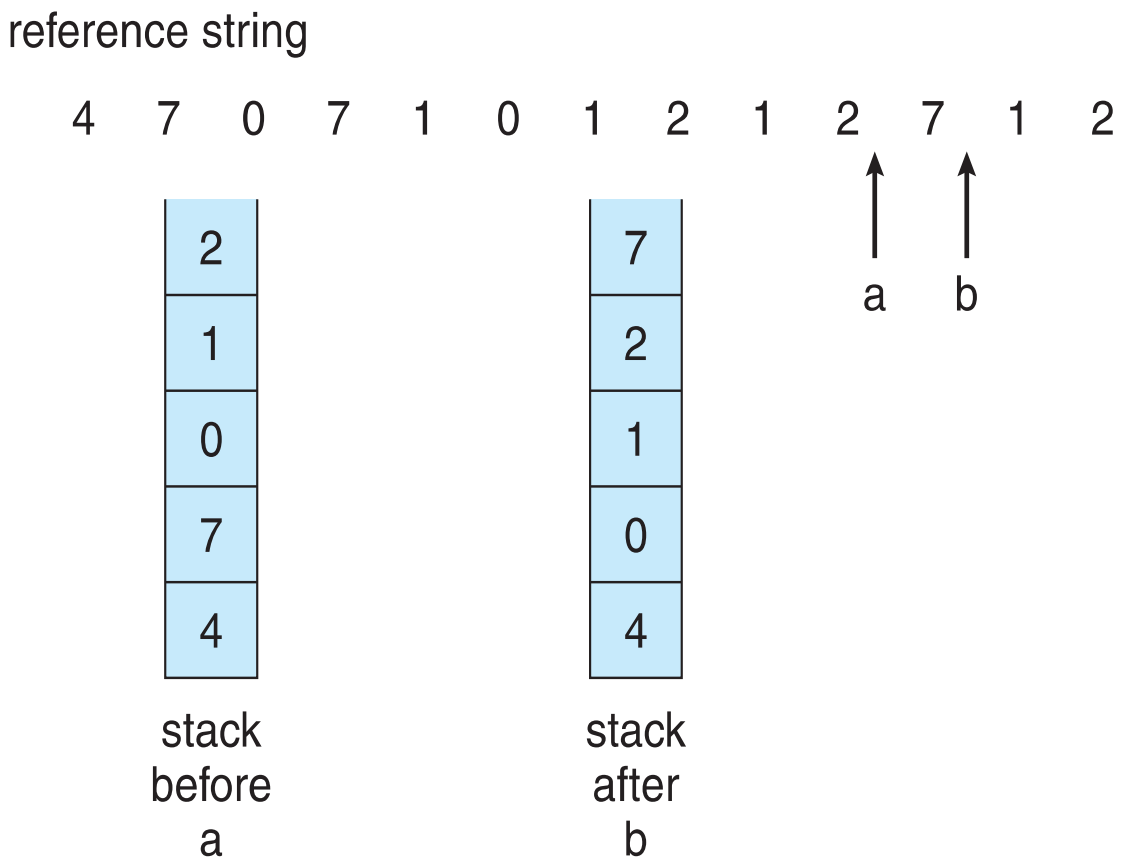


- 3 frames and 12 faults – better than FIFO (15) but worse than OPT (9)
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

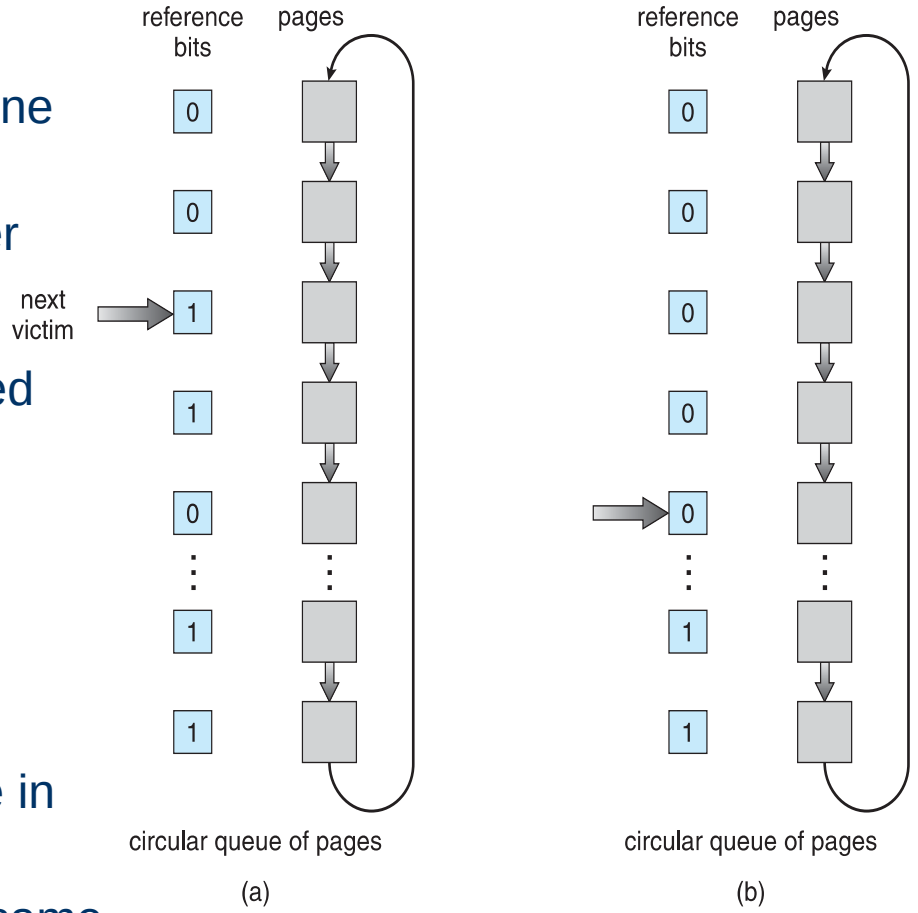
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of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

Use of A Stack to Record Most Recent Page References



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
- **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



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- 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

- 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

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least 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 (*soft page fault*)
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of 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
  - **Raw disk** mode
- Bypasses buffering, locking, etc.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- 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*** of course is total 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

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
    - s_i = size of process p_i
    - $S = \sum s_i$
    - m = total number of frames
    - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

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

Global vs. Local Allocation

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

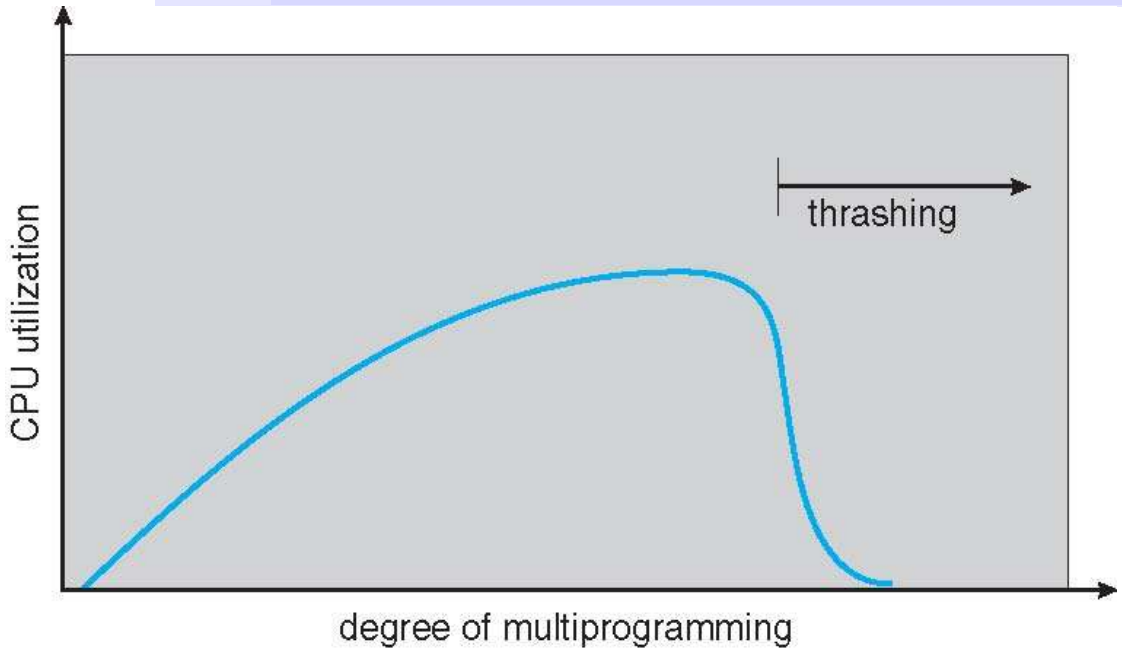
- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - System interface of NUMA systems enables hinting at good processor/core allocation to a task (CPU affinity of a process, thread)

Thrashing

- 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

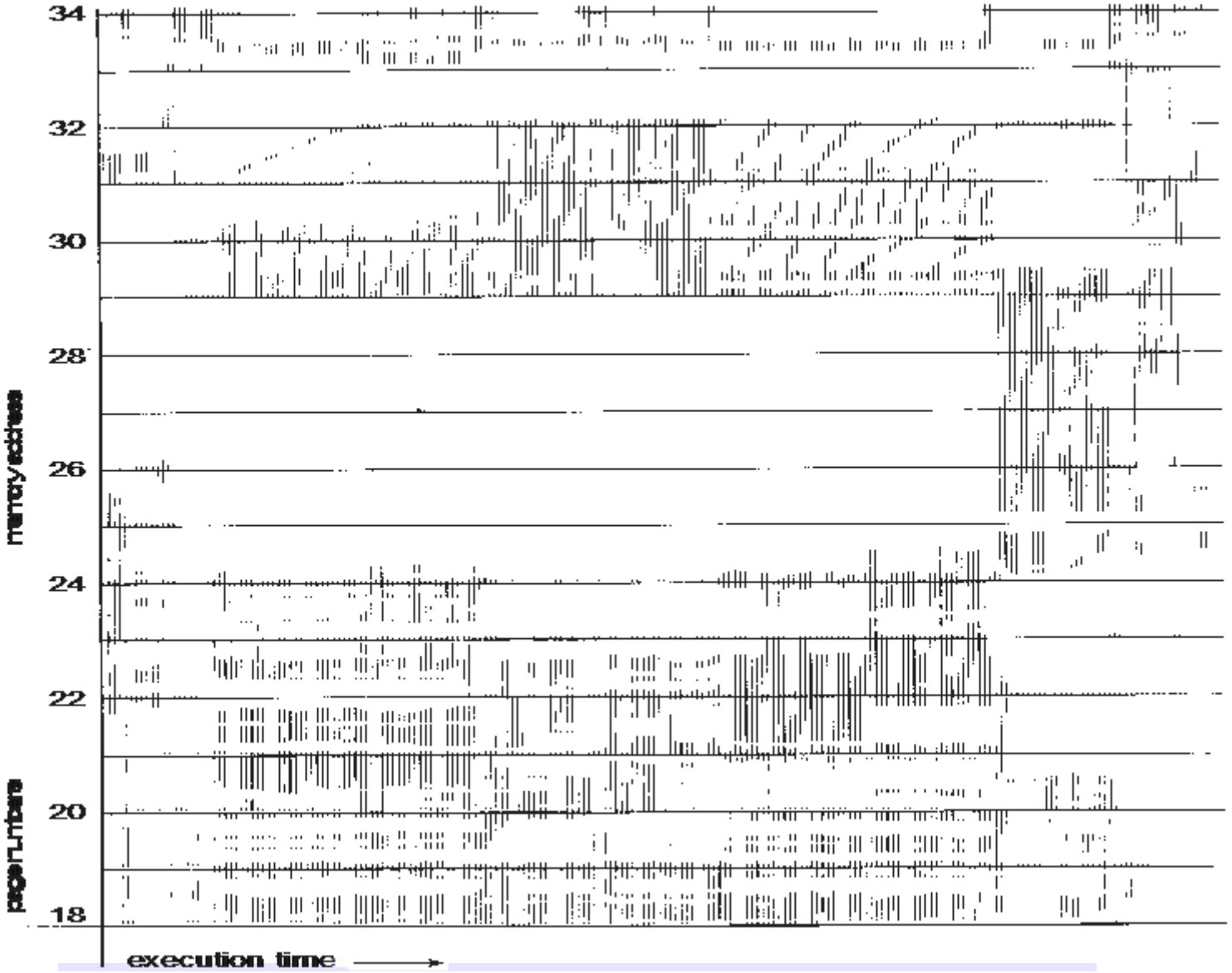
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



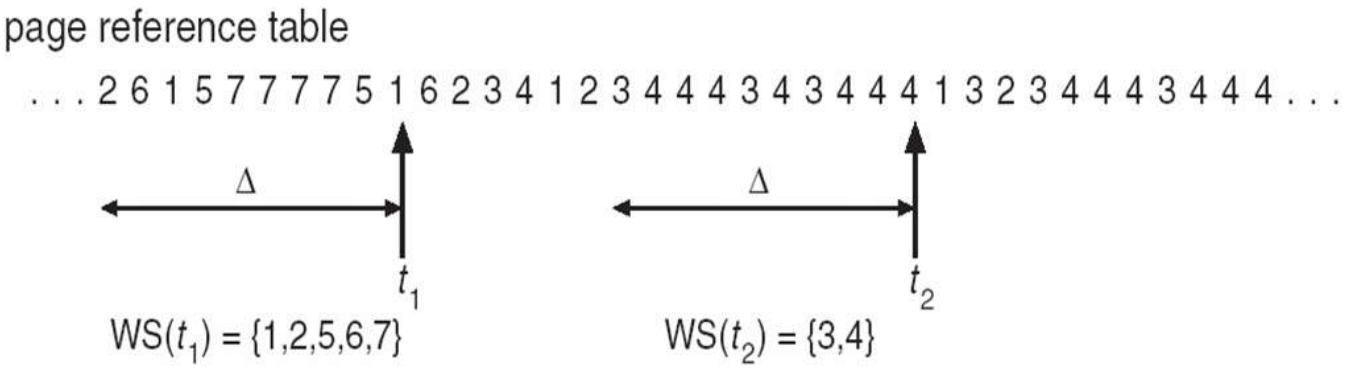
- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WS_i (working set of Process P_i) =
  pages referenced in the most recent Δ (WSS_i – size of WS_i)
  - 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 of frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

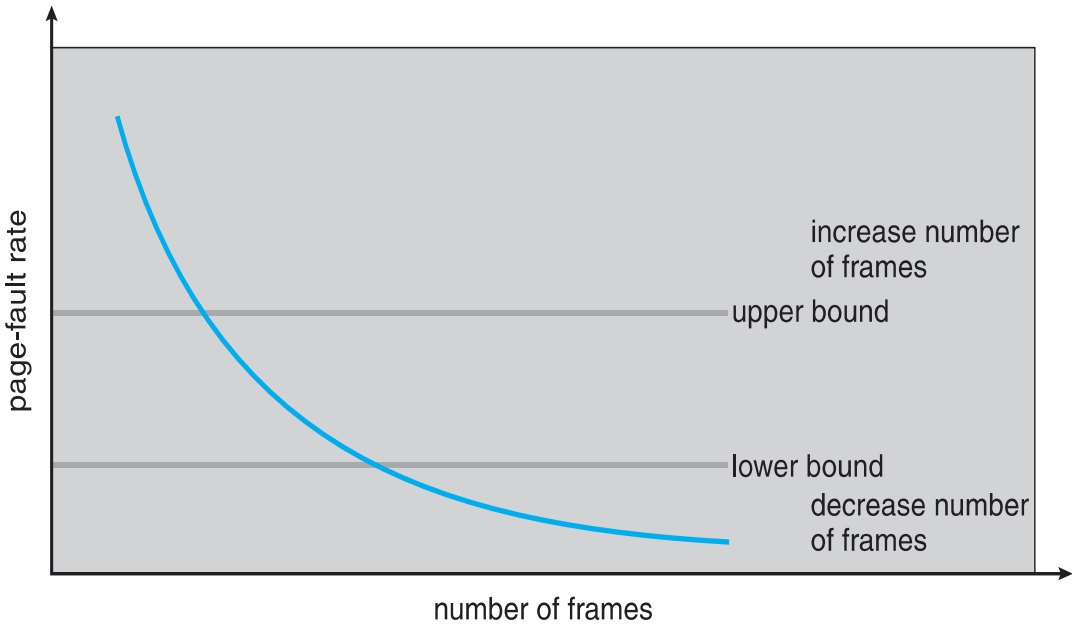


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 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

Page-Fault Frequency

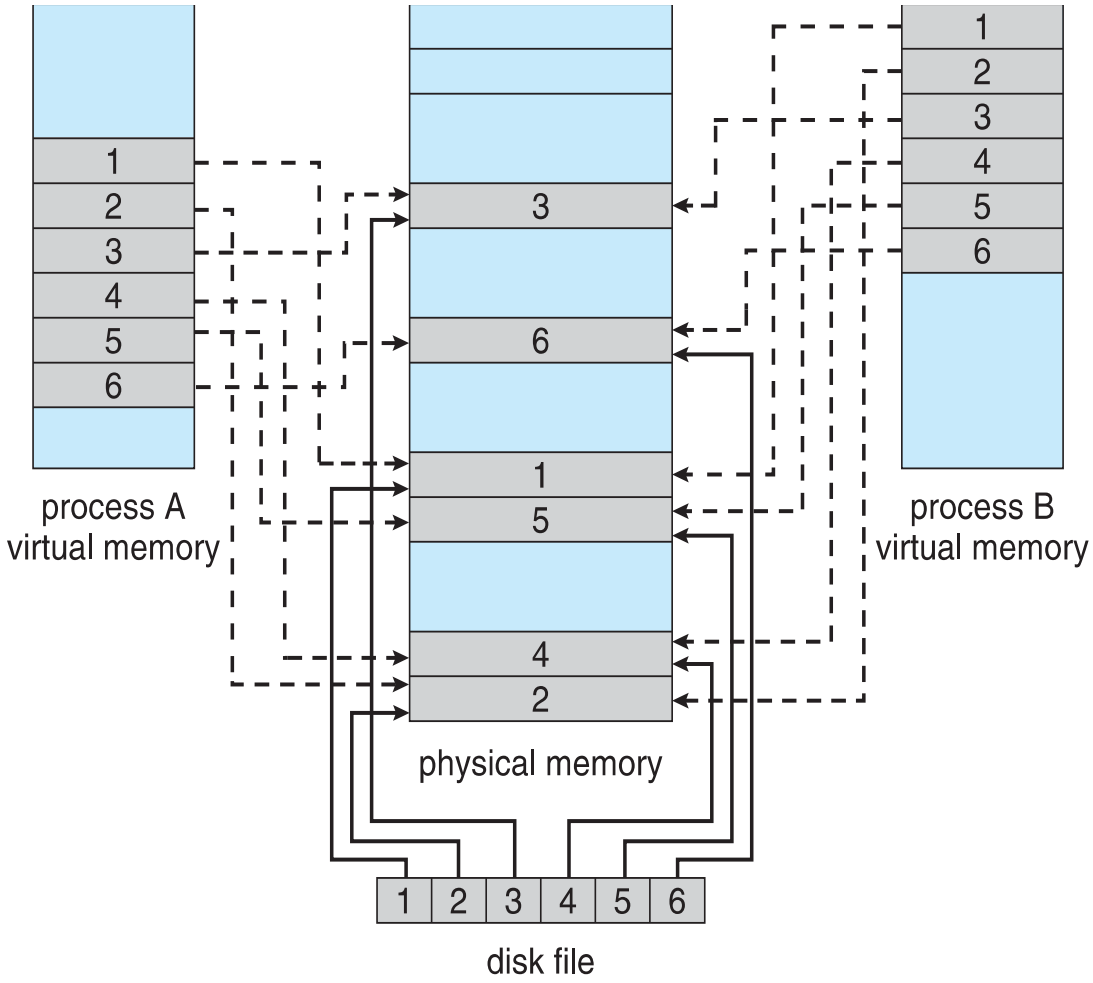
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



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 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
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages
- Some OSes uses memory mapped files for standard I/O

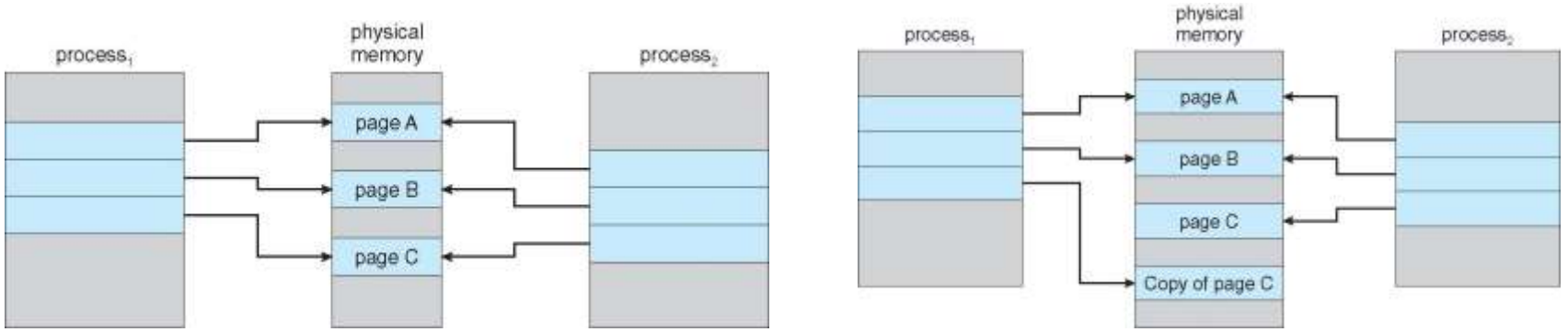
Memory Mapped Files



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
- 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
  - Why zero-out a page before allocating it?

Before (left) and after Process 1 modifies page C

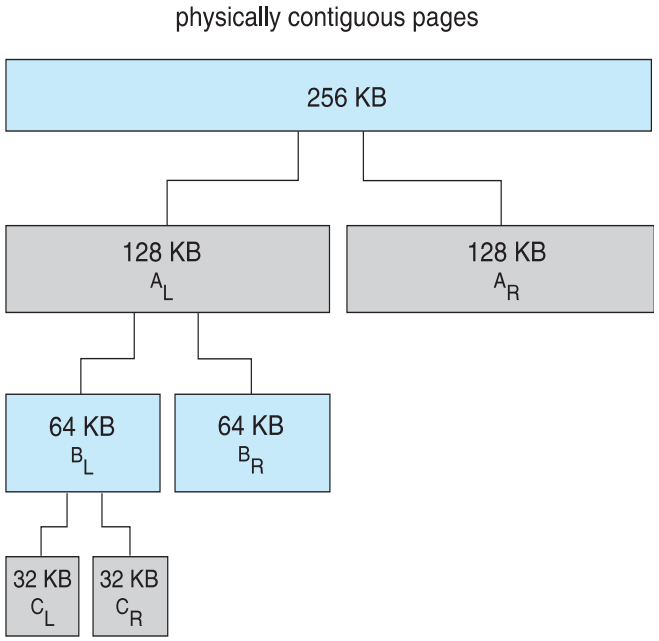


Allocating Kernel Memory

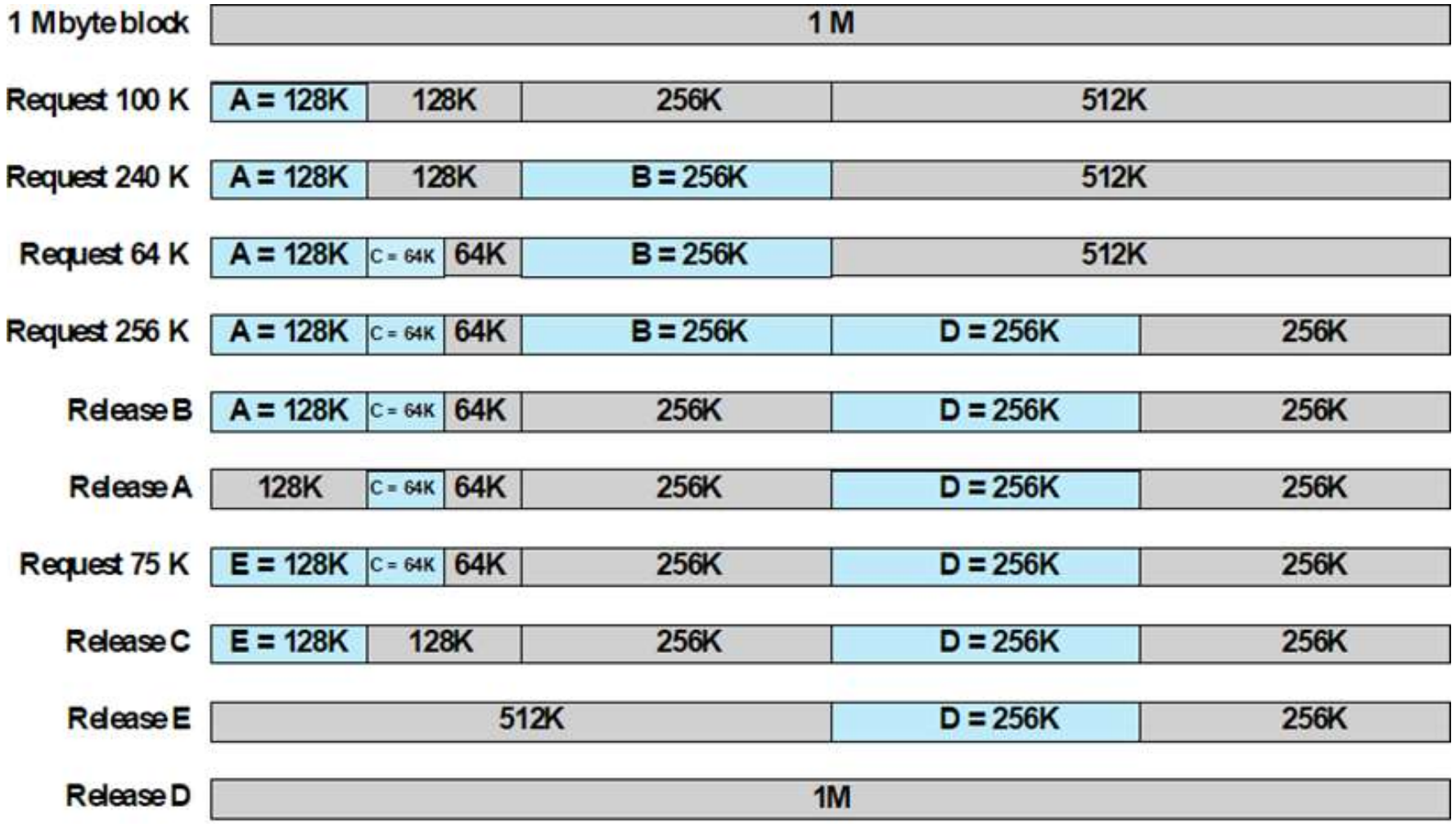
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O
- Popular organizations of kernel memory allocation
  - Buddy system
  - Slab allocation

Buddy System Allocator

- 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
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation



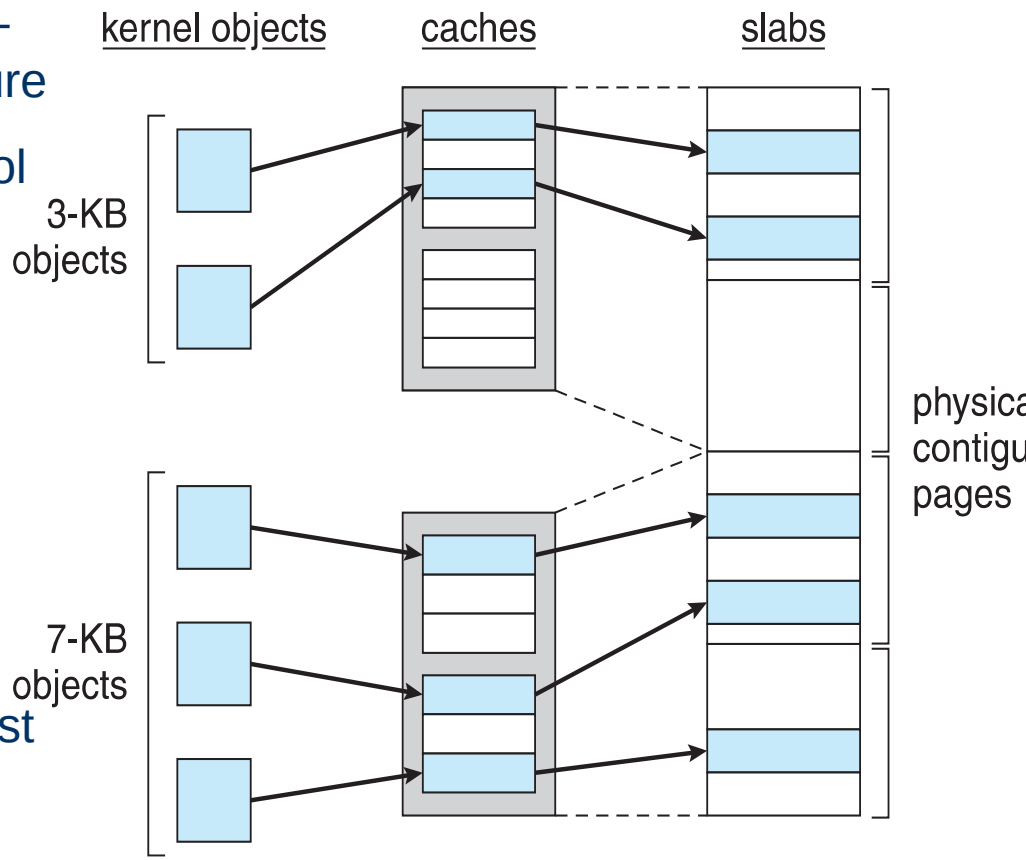
Example allocation with buddy allocator



Allocation status can be kept in a binary tree.

Slab Allocator

- **Cache** consists of one or more slabs
- **Slab** is one or more physically contiguous pages
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with a pool of empty 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 Allocator in Linux

- 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 (Simple List of Blocks) for systems with limited memory. Maintains 3 list objects for small, medium, large objects. Uses first-fit algorithm. Suffers from memory fragmentation.
  - SLUB (the unqueued slab allocator) is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure
- For example process descriptor is of type `struct task_struct;` approx 1.7KB of size
- 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

Other Considerations

- **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

- **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 – 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

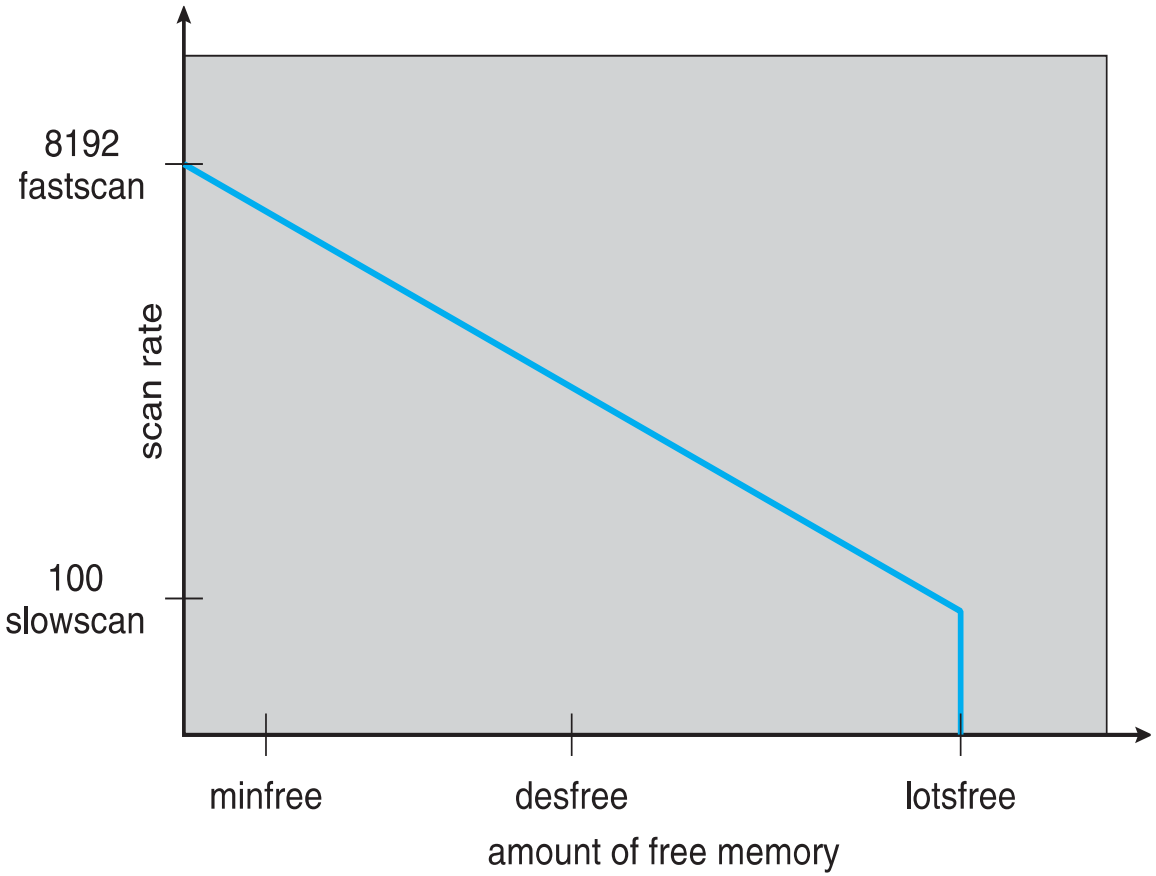
Operating System Examples - Windows

- 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. Attempts to allocate more result in local page replacement.
- 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
- The Windows VM manager uses a two-step process to allocate memory
  - The first step reserves a portion of the process's virtual address space
  - The second step commits the allocation by assigning space in the system's paging file(s)

Operating System Examples – Unix/Solaris

- VM manager maintains a list of free pages to assign faulting processes fast
- **`Lotsfree`** – threshold parameter (amount of free memory) to begin paging (by *pageout* system process). Checking is performed 4 times/sec. Modified clock algorithm is used.
- **`Desfree`** – threshold parameter to increasing paging (*pageout* runs 100 times/sec). If this target cannot be reached within 30 secs the kernel orders process swapping.
- **`Minfree`** – threshold parameter to do swapping when each new page is ordered.
- **`Scanrate`** is the rate at which pages are scanned. This ranges from **`slowscan`** to **`fastscan`**
- **Priority paging** gives priority to process code pages

Solaris 2 Page Scanner

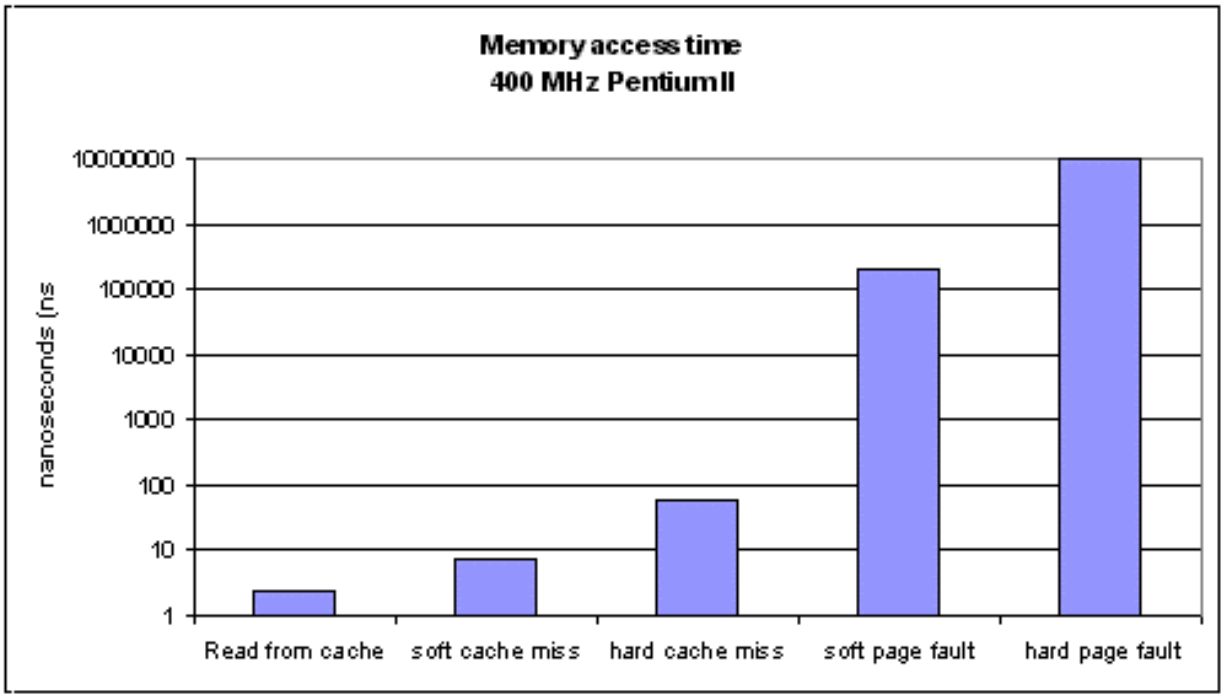


Linux

- Virtual memory implemented as on-demand paging
- Page size dependent on CPU: 4kB (Intel) 8kB (DEC Alpha), 4kB/8MB (Intel 64)
- Page allocator uses buddies algorithm. Kernel keeps lists of continuous sequences of frames of sizes: 1,2,4,8,16,32.
- The kernel swap daemon **kswapd** is started by init (or equivalent) upon system initialization. It wakes up periodically to check if the number of free frames is not too low. If there are not enough free pages, the swap daemon tries three ways to reduce the number of physical pages being used by the system:
  - Reducing the size of the buffer and page caches,
  - Swapping out shared pages,
  - Swapping out or discarding pages.

  By default, the daemon tries to free up 4 pages each time it runs. The above methods are each tried in turn until enough pages have been freed. The swap daemon then sleeps again until its timer expires.
- Page replacement algorithm: LFU (8bit age: page access – incrementation, periodic kernel control – decrementation; ==0 -> page is “old” and can be replaced);

Example memory access time values for PC



- Direct read from L1 cache – 1 cycle (2.5 ns)
- Direct read from L2 cache– 3 cycles (7.5 ns)
- Direct read from RAM– 14 cycles (35 ns)
- Read from a non-resident page – 4000000 cycles(10ms)
- As above, but when the page was buffered by disk driver or kernel – 80000 cycles(200 υs)