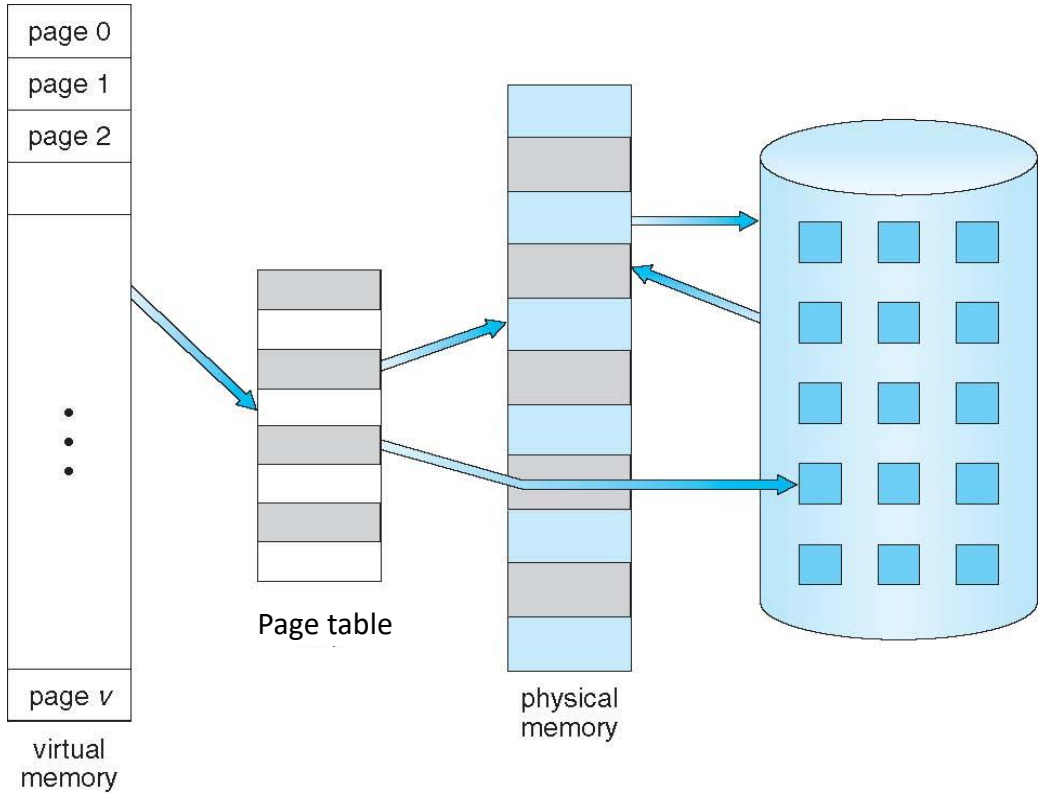
9.1 Virtual memory

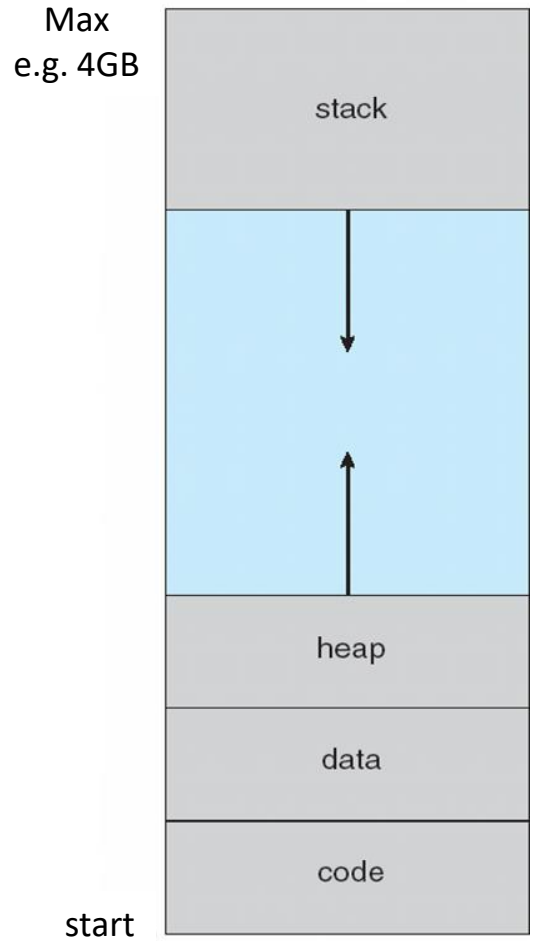
- What we already know from paging:
  - **Virtual address space** – logical view of how process is stored in memory
    - Usually **start at address 0**,
    - **Contiguous** addresses until end of space
  - Meanwhile, **physical memory** is organized in page frames. MMU must map logical (aka virtual) addresses to physical addresses;
    - Physical address space allocated to a process may not necessarily start at 0
    - May not necessarily be contiguous.
- Virtual memory may also allow a process to have a **virtual address space that is much larger than the available physical memory**. This may be implemented via:
  - Demand segmentation
  - **Demand paging** (more common, next topic)



Virtual memory That is Larger Than Physical Memory

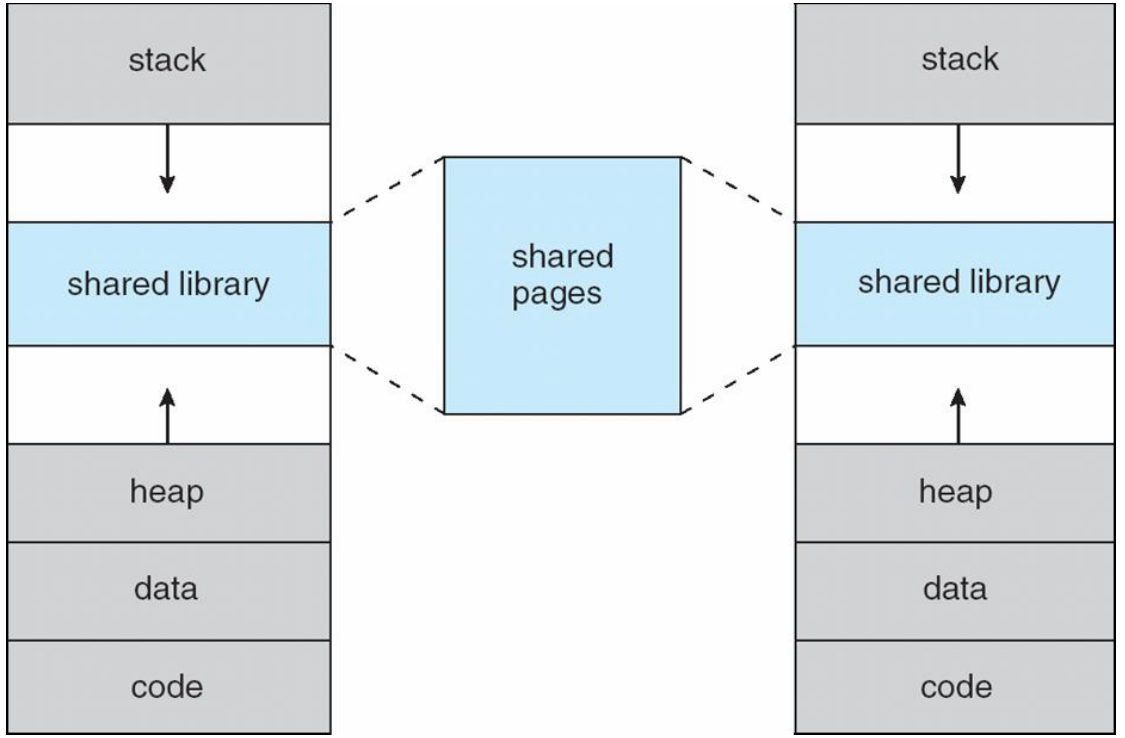
Virtual-address Space

- Usually the virtual address space for the **stack** section is designed to start at program's max virtual address and grow "downwards" while the **heap** grows "upwards" – (virtual addresses)
  - Maximizes address space use
  - Unused address space between the two is a **hole**.
  - Additional physical memory is needed if the heap or stack grow to a given new page
- Note that in 32-bit Linux, the lower 1 GB of every process is reserved for the kernel (code, data, heap and stack).
  - Shared amongst processes
  - Protected from user mode access.



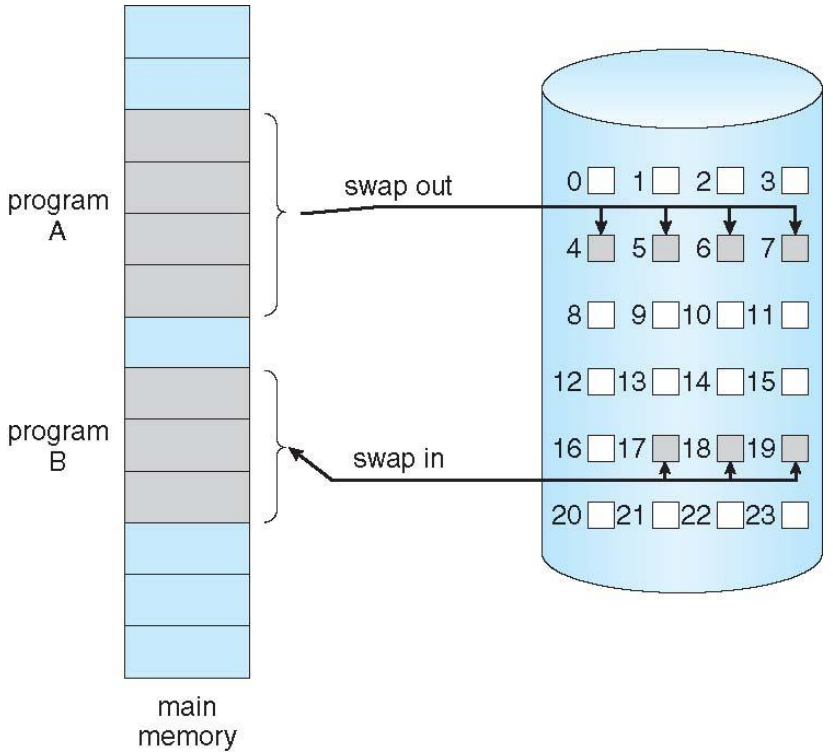
Shared libraries using virtual memory

- This enables **sparse** address spaces with holes left for **growth**, i.e. allows future loading of dynamically linked libraries, shared memory, etc.
  - ***System libraries*** are shared via mapping into virtual address space
  - ***Shared memory*** is implemented by mapping pages (read-write) into the virtual address space



9.2 Demand Paging

- Memory segmentation is a refinement of variable partition strategies:
  - Instead of allocating entire programs using variable partitions → allocate segments instead
- Memory paging is a refinement of fixed memory partition strategies;
  - Instead of allocating entire programs using fixed partitions → allocate pages instead
- The concept of **memory swapping** dictated that we load an entire process into memory as demonstrated by diagram on the right (do you recall the **swap scheduler**?).
- How do we combine swapping strategies with paging to achieve a virtual memory space that is **much larger than** the available physical memory?
  - Swap individual pages instead of whole programs, when needed → **demand paging**



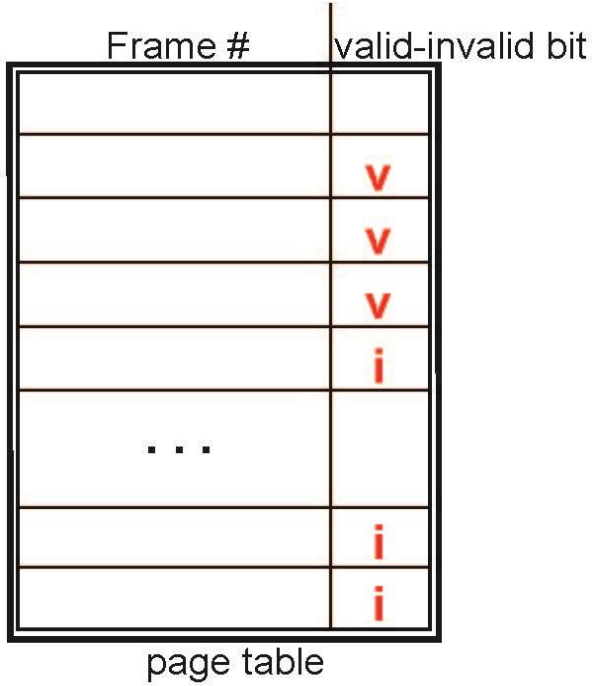
Memory swapping

Demand Paging

- **Demand paging** brings a page into memory only when it is needed, and thus only a portion of a program may be initially loaded by the OS:
  - Less I/O needed, no unnecessary I/O
    - Faster response for initial loading of programs
  - Less memory needed for running a process.
    - Higher degree of multiprogramming
- Page is needed $\Rightarrow$ reference to it
  - not-in-memory $\Rightarrow$ bring to memory (from backing store)
  - invalid reference (i.e. outside the process' addr. Space) $\Rightarrow$ abort
- The **pager** is the kernel component that is in charge of loading/storing a page from/to the backing store.

Valid-Invalid Bit – another usage

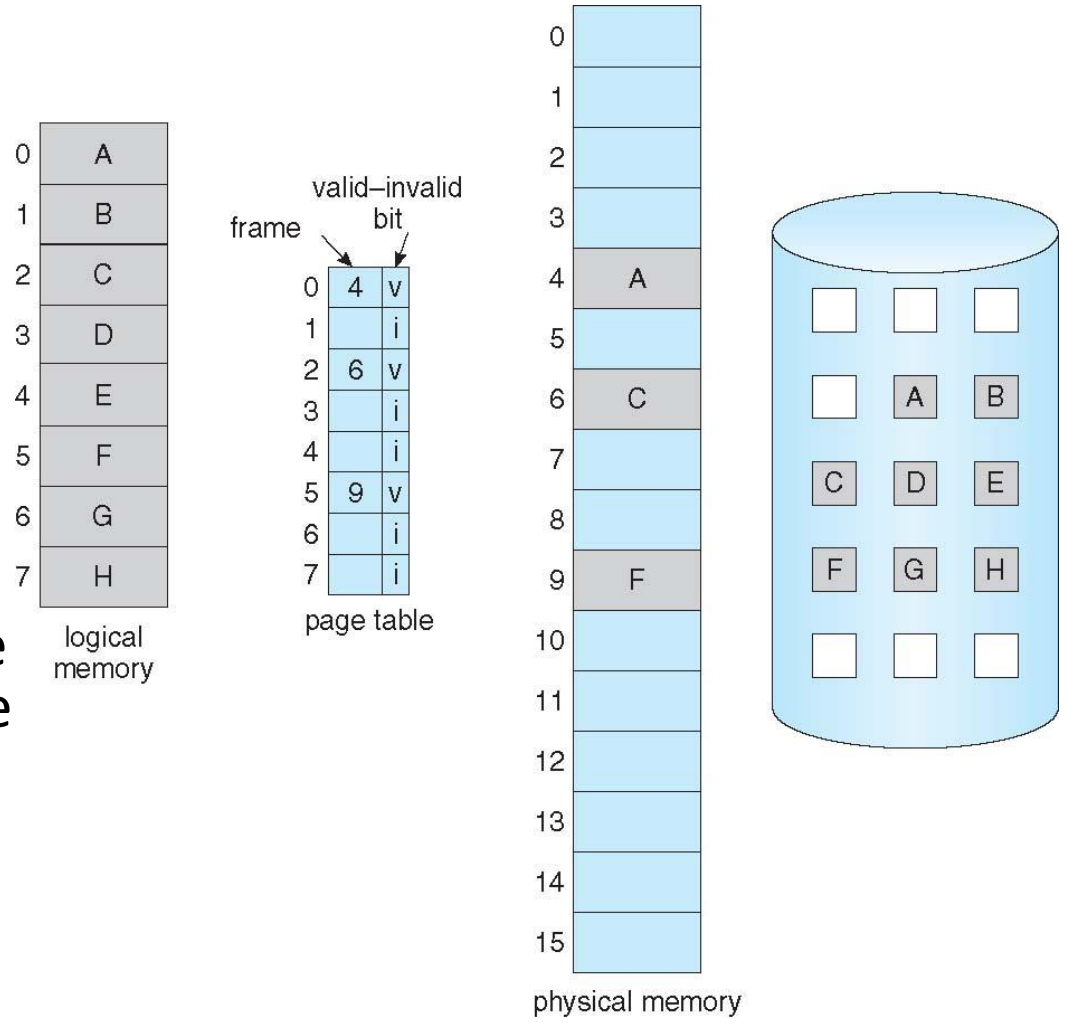
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



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

Example of a page table when some pages are not in main memory

- In demand paging, this is not the same usage of the **valid bit** as in the previous lecture, where it was used to determine whether a page is legal or not. In other words, this is the second valid bit.
  - **First valid bit** decides whether a page belongs to the process' address space or not.
  - **Second valid bit** determines whether a page is memory resident or not.
- The usage here is to determine whether a page is resident in main memory or not.



Page Fault

- If there is a reference to a page, and it happens to be the first reference then it will cause an exception and invoke the operating system:

On a page fault:

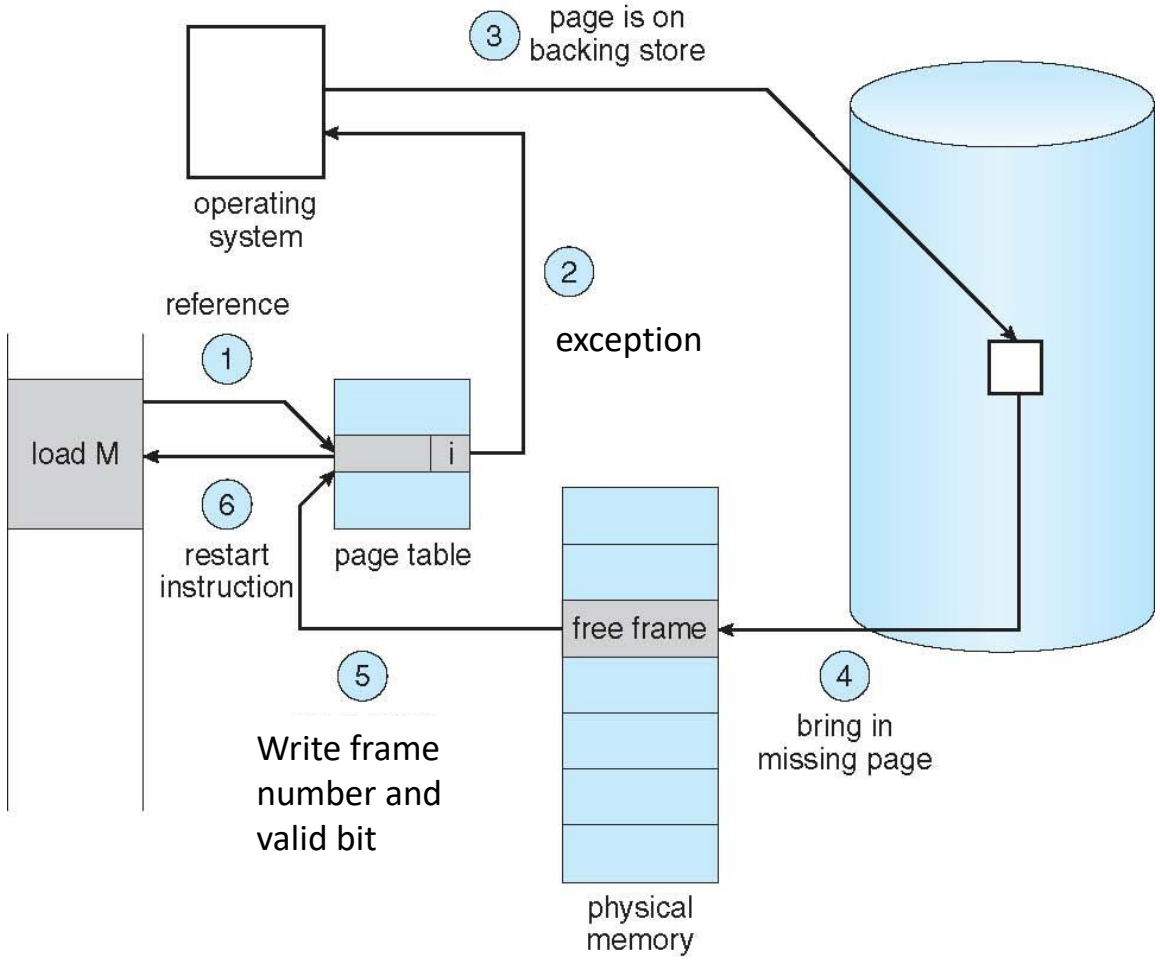
a) Operating system locates the page on the page table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory, then continue to step b

b) Find free frame

c) Load page into frame via scheduled disk operation

d) Modify the page table to indicate page now in memory
   - Set the frame number
   - Set valid bit (second) = **v**

e) Restart the instruction that caused the page fault



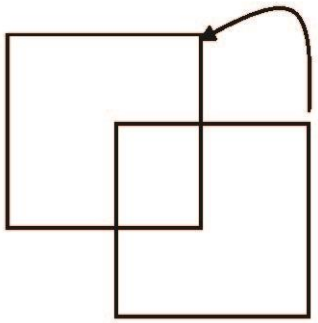
NOTE: Contrary to what "fault" might suggest, **page faults** do not represent errors, unless the page is not in the process' address space, and only in such case that it may be a true fault, **a segmentation fault**, which is an error.

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process (what's that called?), non-memory-resident → page fault
  - Same occurs 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
    - Would this instruction exist in a RISC or in a CISC processor architecture?
- Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit (**H/W traps the OS** if page invalid)
  - Secondary memory (swap device with **swap space**, e.g. hard drive)
  - **Instruction restart**

Instruction Restart issues

- Consider a machine instruction that could access several different locations (e.g. a “move multiple” type instruction)
  - block move (auto increment/decrement location)



  - Restart the whole operation?
    - What if source and destination overlap?

Performance of Demand Paging

Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state (for context switching)
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. Allocate a free frame
6. 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

7. While waiting, allocate the CPU to some other user (i.e. do context switching)
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and process state for the other user
10. Determine that the interrupt was from the disk

11. Correct the page table and other tables to show page is now in memory

12. Wait for the CPU to be allocated to this process again (or invoke the scheduler to run the next process)
13. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- 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) –
  - **Assume** a process is in steady state, i.e. all frames allocated to the process are already filled with process' pages and thus loading a new page requires the removal of one of its already loaded pages from memory in order to free up a frame.

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

Demand Paging Example

- Memory access time = 200 nanoseconds. Average page-fault service time (overhead + swap in + swap out) = 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 41!!

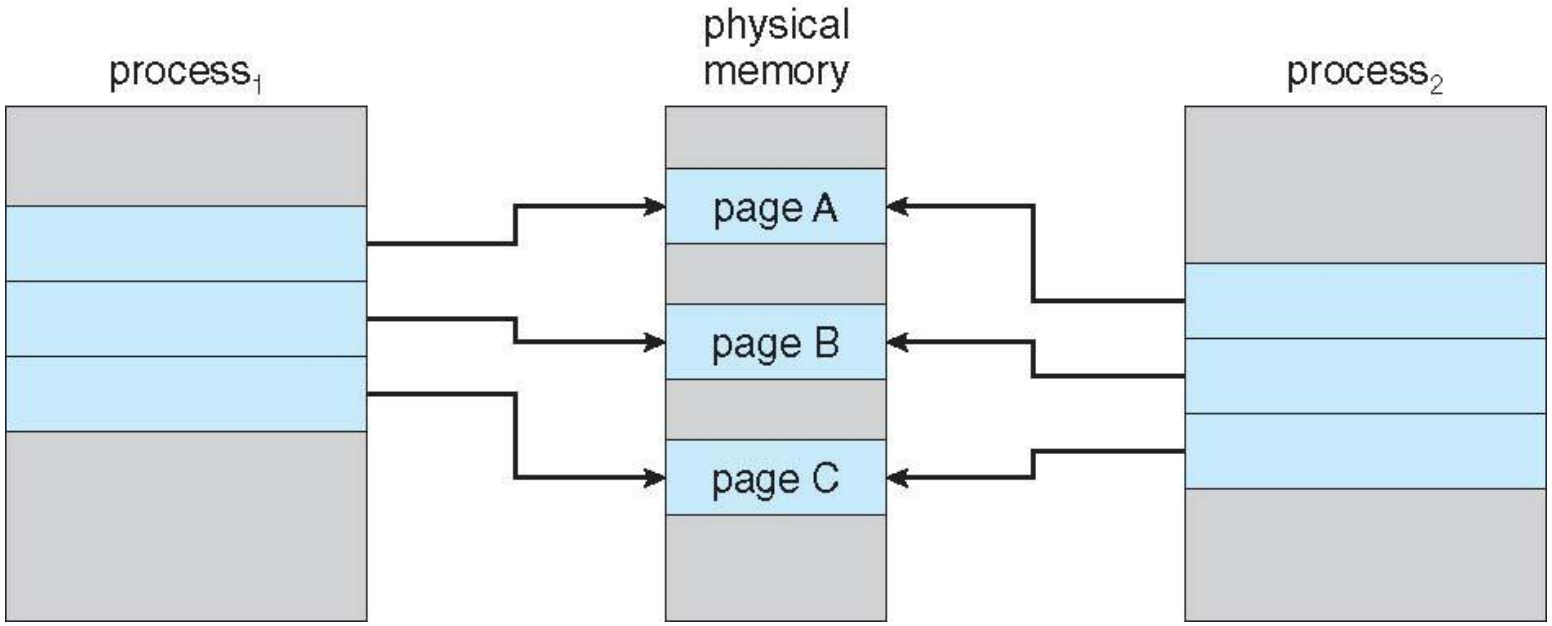
- If we need to limit the performance degradation to < 10 percent
  - $220 > 200 + 7{,}999{,}800 \times p$
    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$ (i.e. 2.5 in a million)
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Use **swap space**: Swap space I/O is faster than file system I/O, even if on the same secondary storage device.
  - Because swap allocated in larger chunks, less management needed than file system → faster access
  - Hence, copy entire process image to swap space at process load time
    - Then page in and out of swap space
    - Used in older BSD Unix
- For **unmodified** or **read-only pages**, page-in from program binary on disk, but discard (when time to do so) rather than paging-out when freeing frame
  - Used in Solaris and current BSD versions
- But the following pages still need to be written to swap space:
  - **Pages associated with a file** that are modified in memory but not yet written back to the file system
  - **Anonymous R/W memory pages** (not associated with a file), e.g. stack and heap pages.

9.3 Copy-on-Write

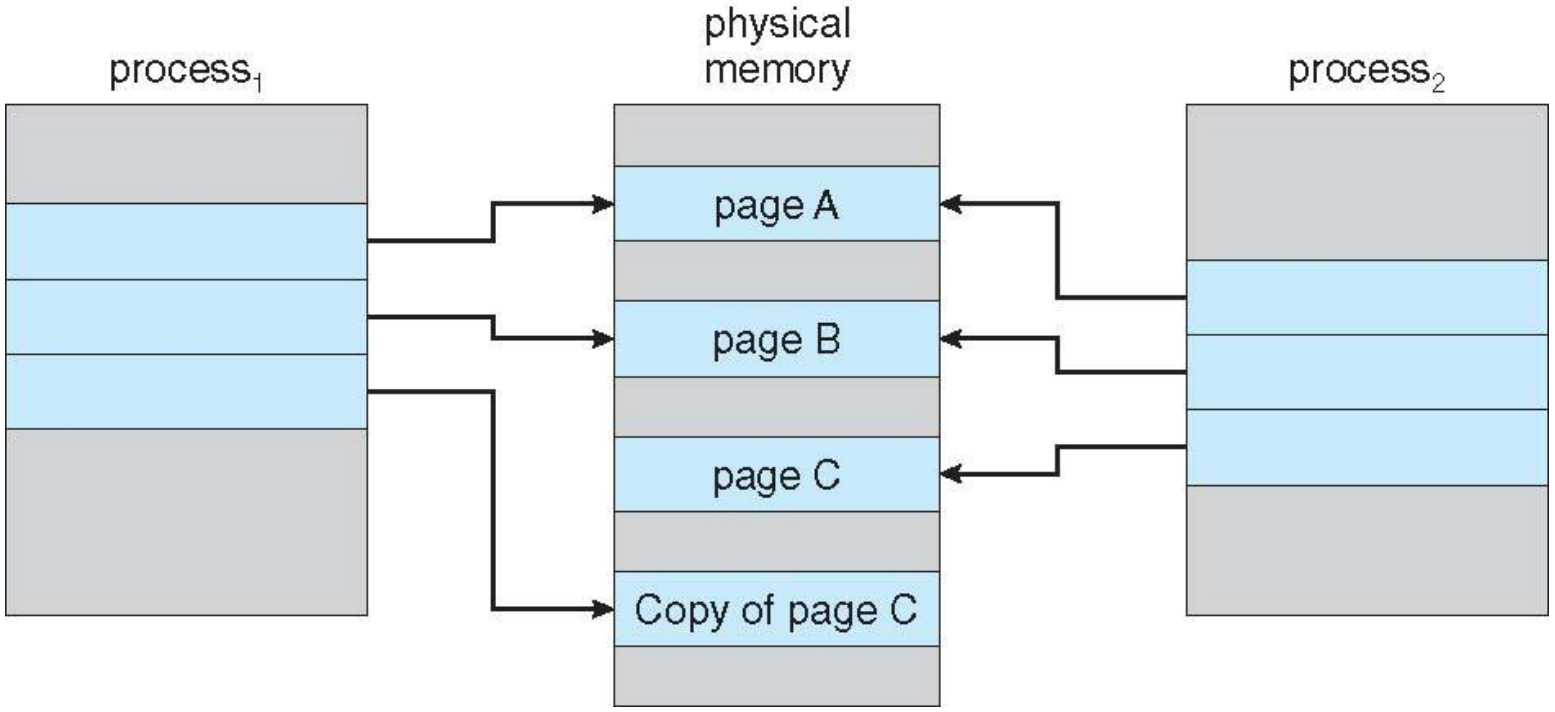
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory (***except for stack pages***).
  - If either process modifies a shared page, only then is the page copied



COW – Before process 1 modifies page C

Copy-on-Write - cont.

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



COW – After process 1 modifies page C

Copy-on-Write - cont.

- In general, free frames 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?
- `vfork()` variation on `fork()` system call has parent suspend till the child calls `execve()` or aborts. Note that the `execve()` takes some parameters, including the path of the file containing the new program to be loaded.
  - Designed to have child immediately call either `execve()` or `_exit()`
    - **If child does anything else, it may corrupt the parent's address space.**
  - Very efficient – It differs from `fork()` in that it shares the same exact address space as the parent **including** the stack:
    - It does not copy the page table (uses the parent's page table, i.e. PTBR points to parent's page table)
    - Does not set copy-on-write bits for all writable pages in *both processes.*
    - Does not flush the TLB
    - Still copies file descriptors from parent process.
- In today's systems, the standard fork() is very efficient, thanks to the COW hardware → no need to use vfork()

Frame allocation and page replacement algorithms

- **Frame-allocation algorithms** determine how many frames are allocated to each process (dynamically or statically)
- **Fetch policies** decide on which page to fetch next. In **Demand paging**, the next page to fetch is the one that caused a page fault. The other alternative is to use **prefetch**, which is not common due to difficulty in predicting the next page.
- **Page placement algorithms** determine where to place a fetched page in the available frames (if more than one frame is available).
- **Page-replacement algorithm** determine which page to evict in order to free up a frame (if no free frames are available for a process) to load the new demanded page.
  - 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**, aka **page trace**) and computing the number of page faults on that string
  - A reference string contains accessed (referenced) page numbers, not offset addresses within a page.
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

Frame allocation and page replacement algorithms – cont.

Page trace (aka **reference string**)

- In a 16-bit system, if we trace a particular process, we might record the following address sequence:

  0x0100, 0x0432, 0x0101, 0x0612, 0x0102, 0x0103, 0x0104,

  0x0101, 0x0611, 0x0102, 0x0103, 0x0104, 0x0101, 0x0610,

  0x0102, 0x0103, 0x0104, 0x0101, 0x0609, 0x0102, 0x0105

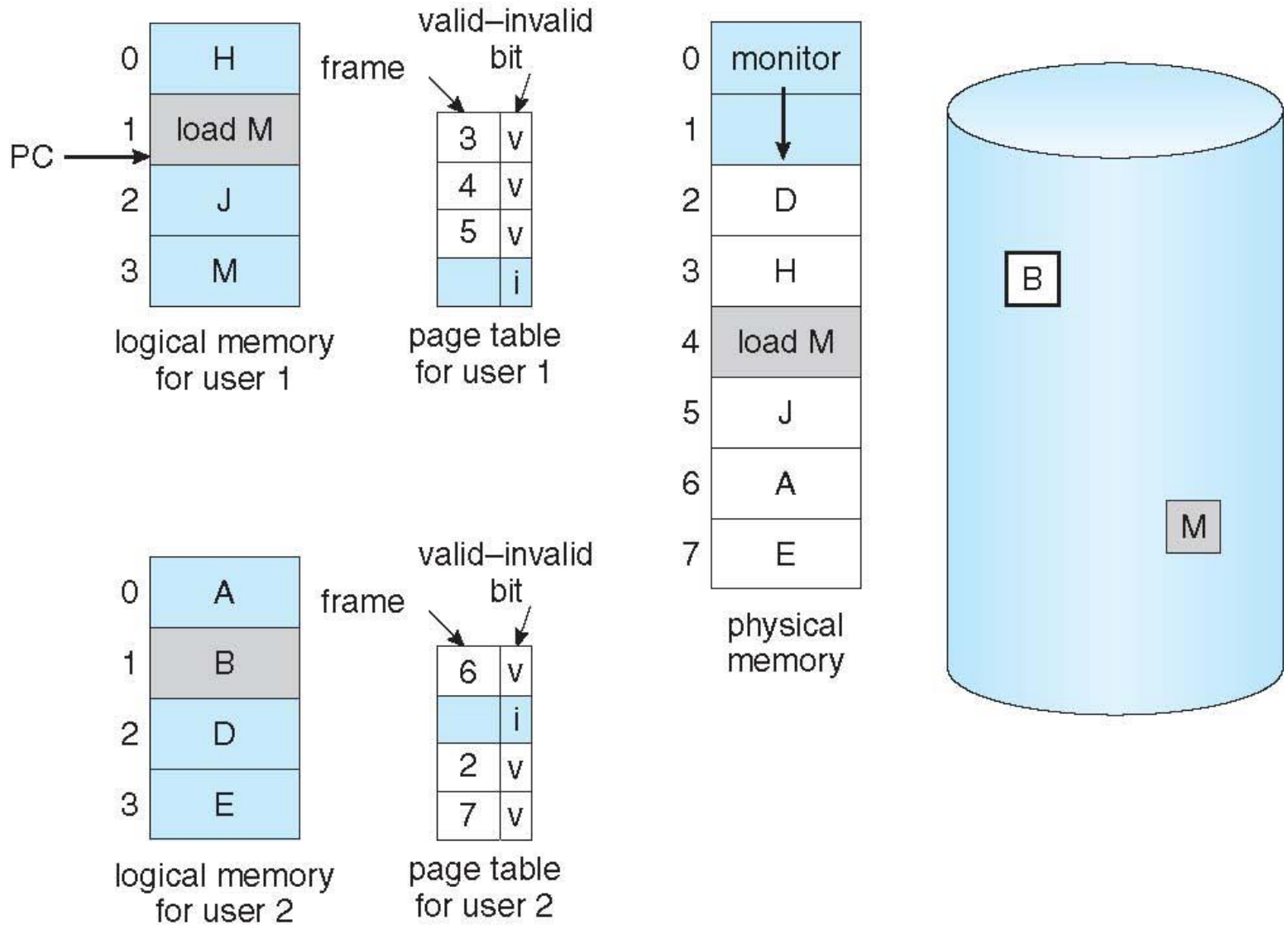
- If the page size is 256, then the page trace is:

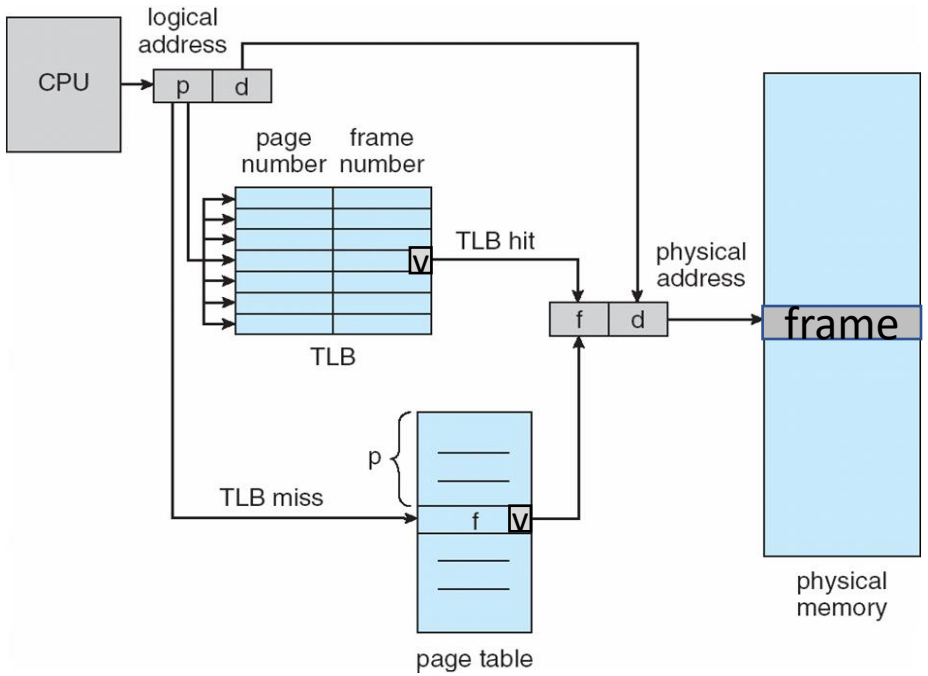
  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

9.4 Page replacement

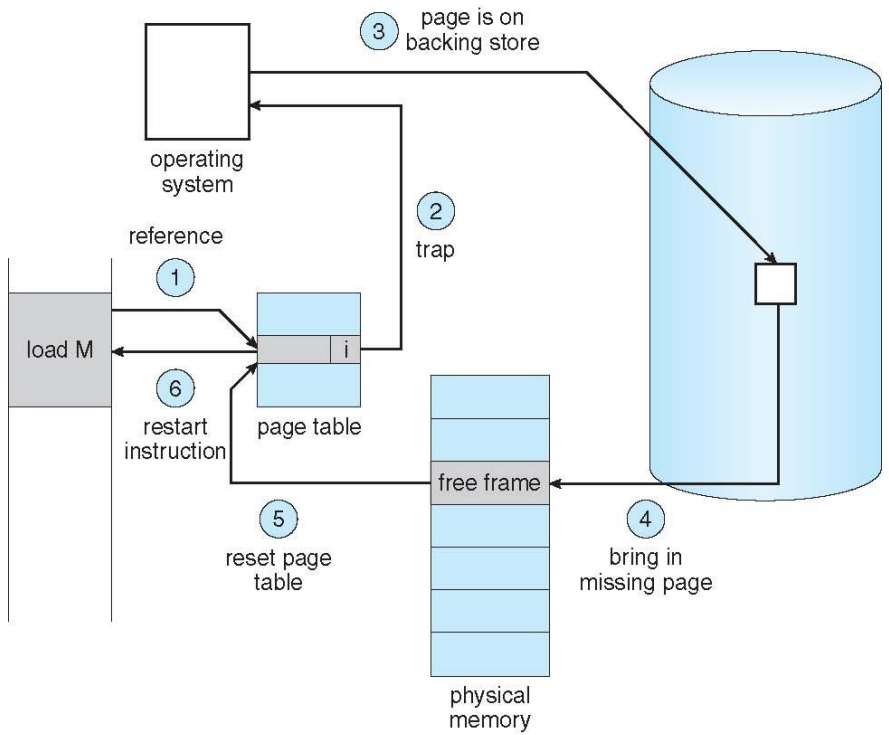
- What happens if there are no free frames? Options are:
  - Terminate process – not a good idea
  - Page replacement – find some page in memory (that is least used for example, determined by some algorithm) and page it out.
    - Performance – want an algorithm which will result in minimum number of page faults
- Even if there are free frames in the system, it may be desired to prevent **over-allocation** of memory to one process and limit its allocated number of frames.
- A **modify** (**dirty**) **bit** may be used (for each page entry in the page table) to reduce overhead of page transfers – only modified pages are written to disk (if chosen to be evicted from main memory)
  - So now the page table has a valid bit (or two), a cow bit and we just added a modify bit!

Need For Page Replacement





No page fault (page is in a memory frame)



page fault (page not in memory frame)
TLB not shown

9.4.1 Basic Page Replacement

Handling a page fault exception:

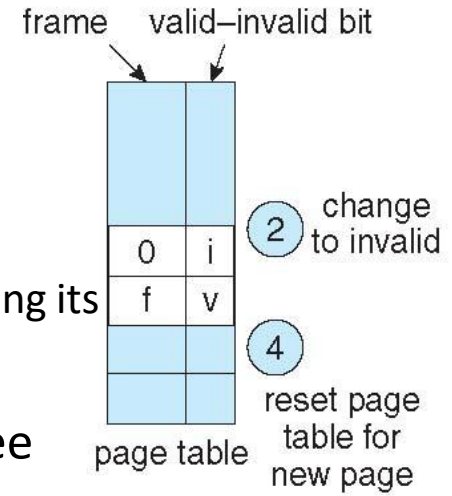
1. Find the location of the desired page on disk

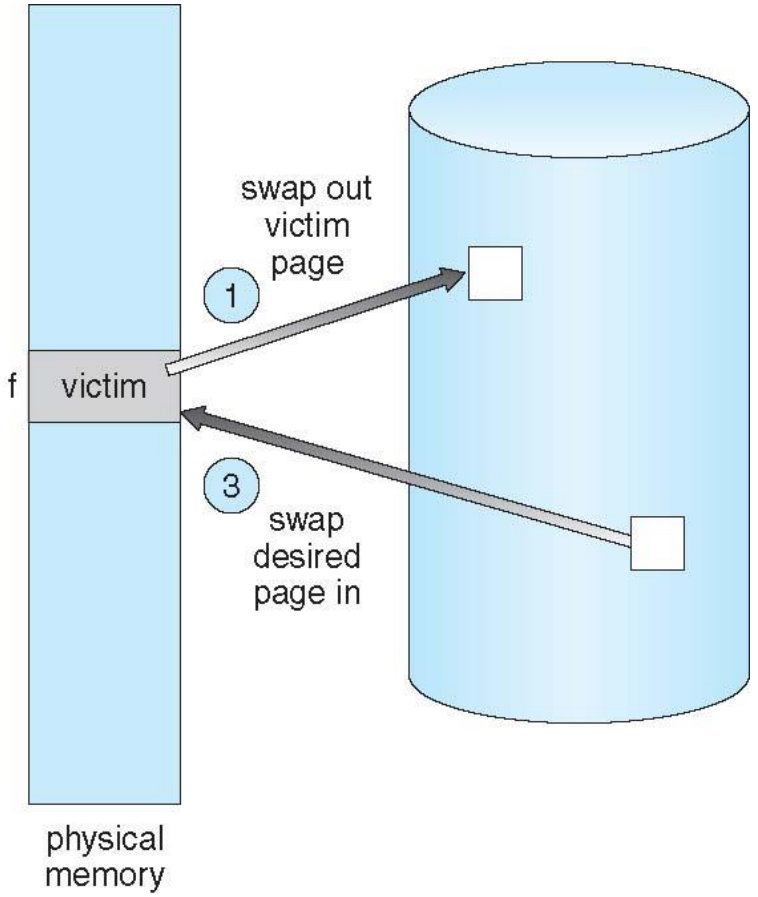
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty (after changing its page table entry to invalid)

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

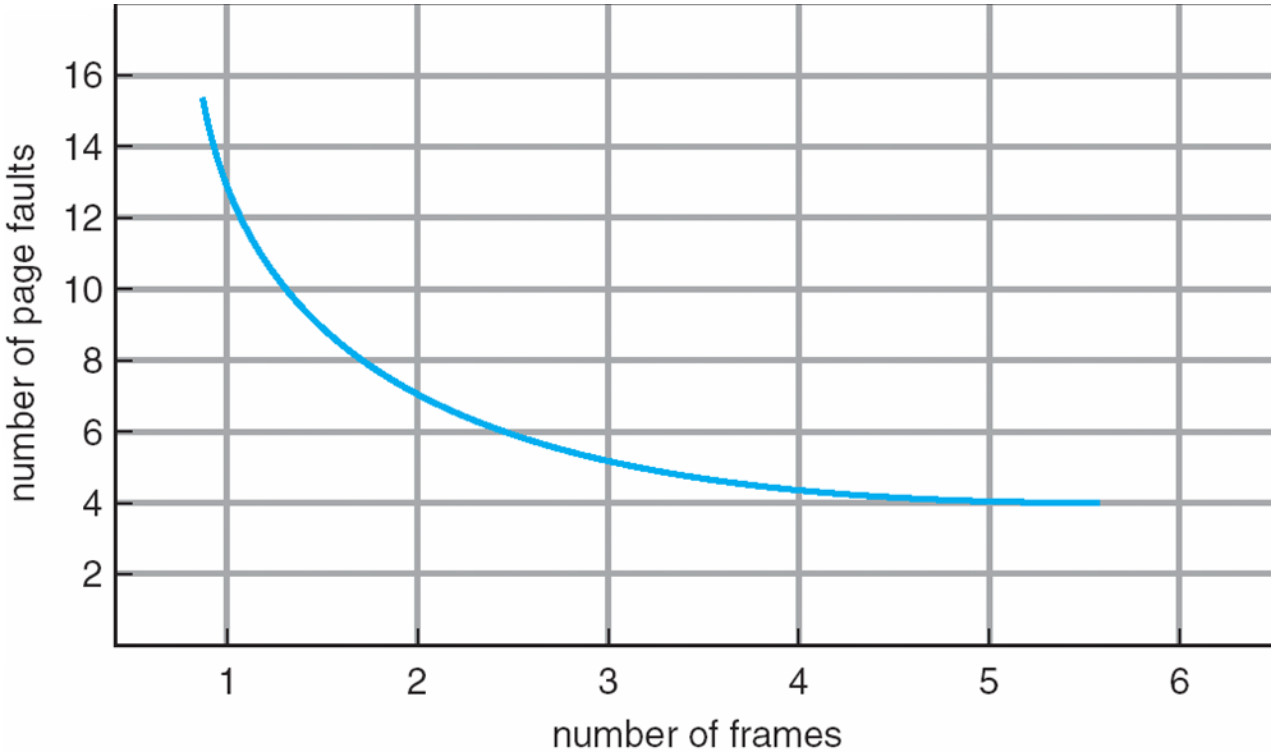
4. Continue the process by restarting the instruction that caused the trap

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





Graph of page faults versus the number of allocated frames

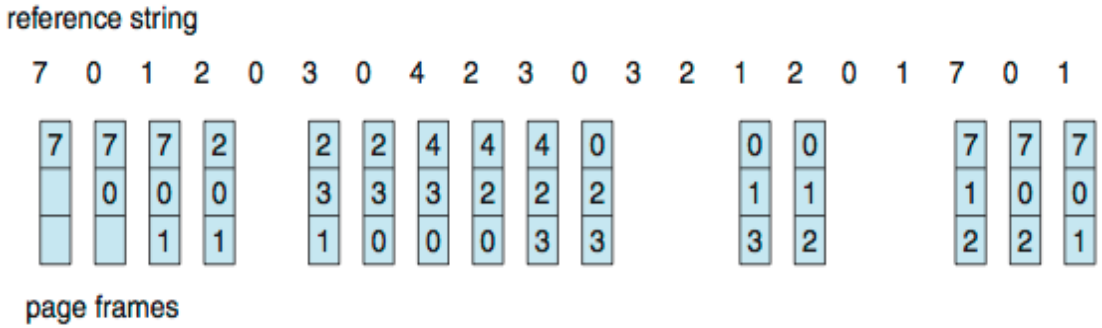


9.4.2 First-In-First-Out (FIFO) Algorithm

- In the proceeding examples, the **reference string** used is

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

- 3 frames allocated (3 pages can be in memory at a time per process)



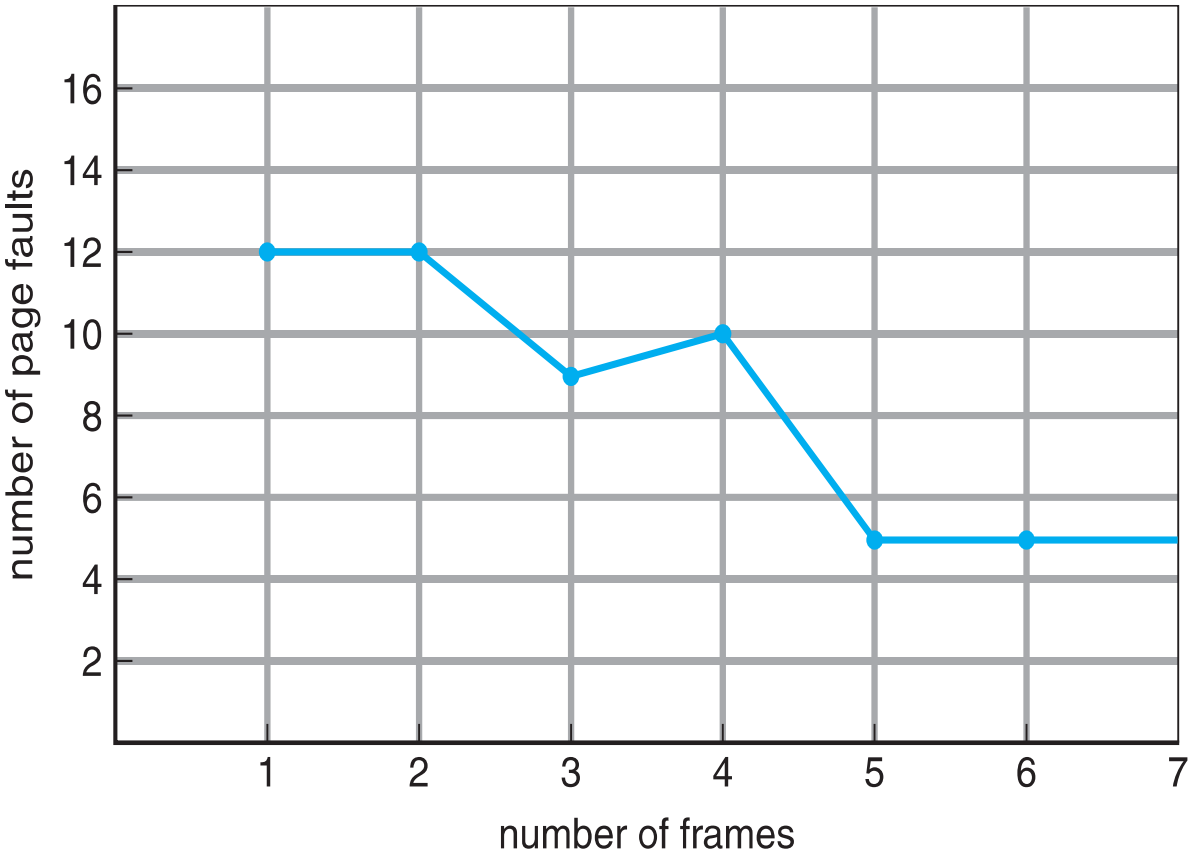
15 page faults

- **Issue:** If a page is in constant use, but was loaded early in program execution, it is then picked for replacement, despite it being in constant usage → page faults (repeatedly)
- How to track ages of pages?
  - Just use a FIFO/queue

The Belady's anomaly

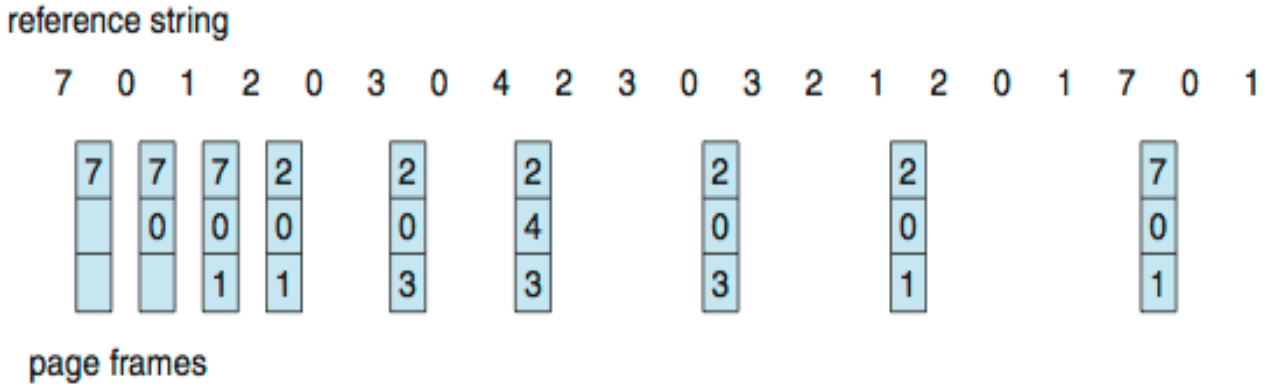
- Results may vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
- Adding more frames can cause more page faults (instead of improving things)!

→ **Belady's Anomaly**



9.4.3 Optimal Algorithm

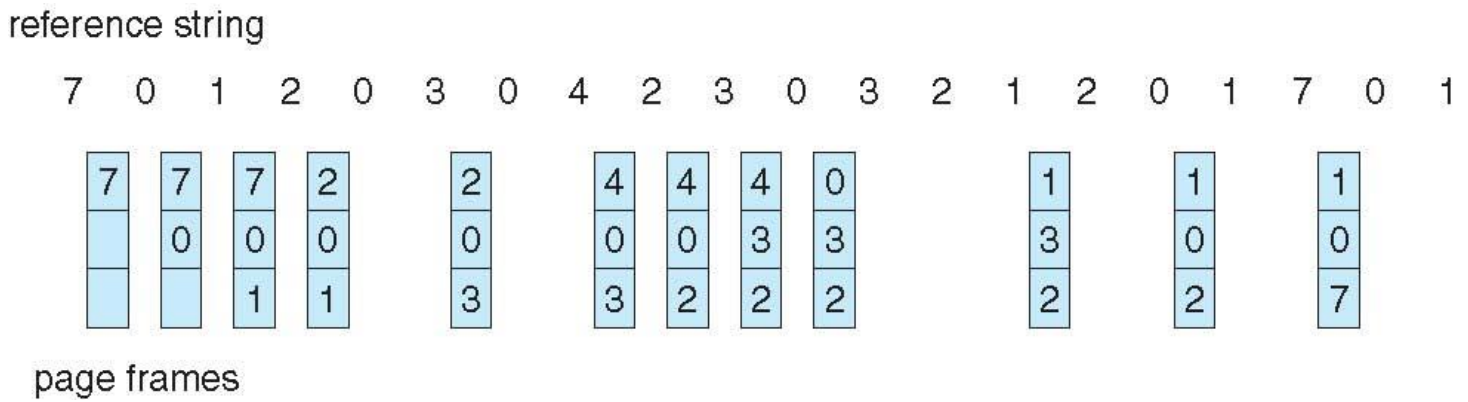
- Replace page that will not be used for longest period of time (i.e. measured by the **maximum forward distance** on page trace)
  - The optimal algorithm results in 9 page faults for the example page trace.
- How do you know this “not used for longest period of time”?
  - We don’t, we can’t read the **future**
    - But we can try to predict (as we shall see later)
  - It is a **hypothetical system** used for measuring how well your algorithm performs



9 page faults

9.3.4 Least Recently Used (LRU) Algorithm

- Use **past knowledge** rather than future
- Replace page that has not been used in the most amount of time (i.e. **maximum backward distance** on page trace)
- Associate time of last use with each page



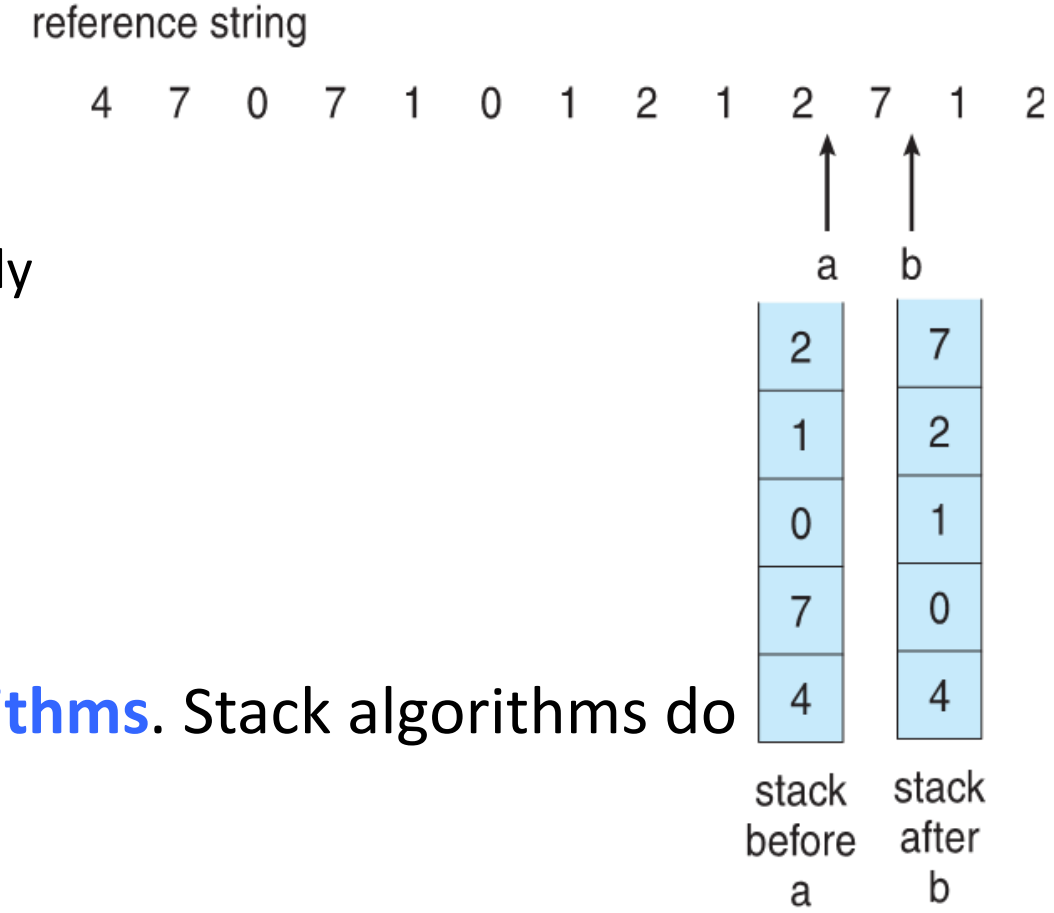
12 page faults

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

LRU Algorithm (Cont.)

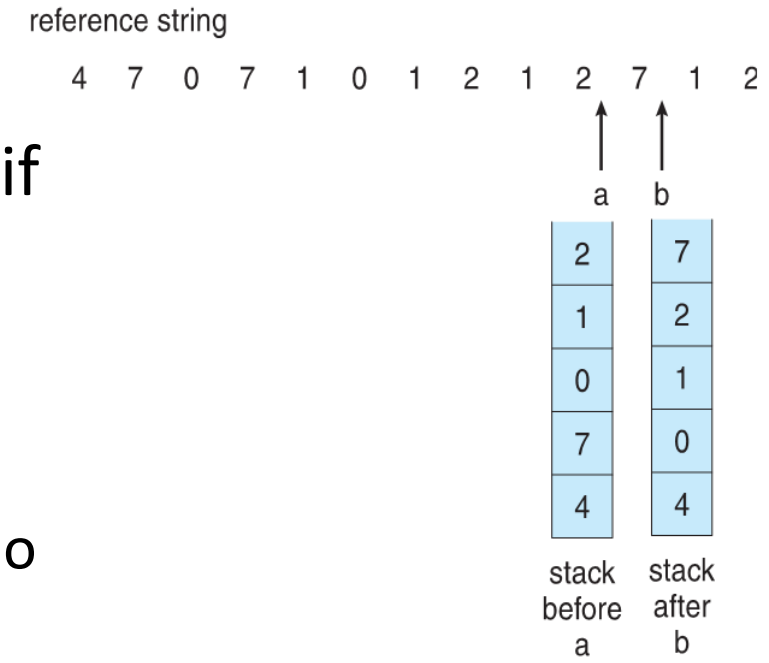
- Counter implementation
  - Every page entry has a counter variable; every time the page is referenced, the H/W copies the clock (Process' clock, not the CPU clock) into that page's counter
  - When a page needs to be changed, the OS kernel looks at the counters to find smallest time value

  (i.e. oldest reference, or LRU)
    - Search through page table needed
- Stack implementation
  - Keep a stack of page numbers in a doubly linked list form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement is needed
  - But each update is more expensive
- LRU and OPT are cases of **stack algorithms**. Stack algorithms do not exhibit the Belady Anomaly



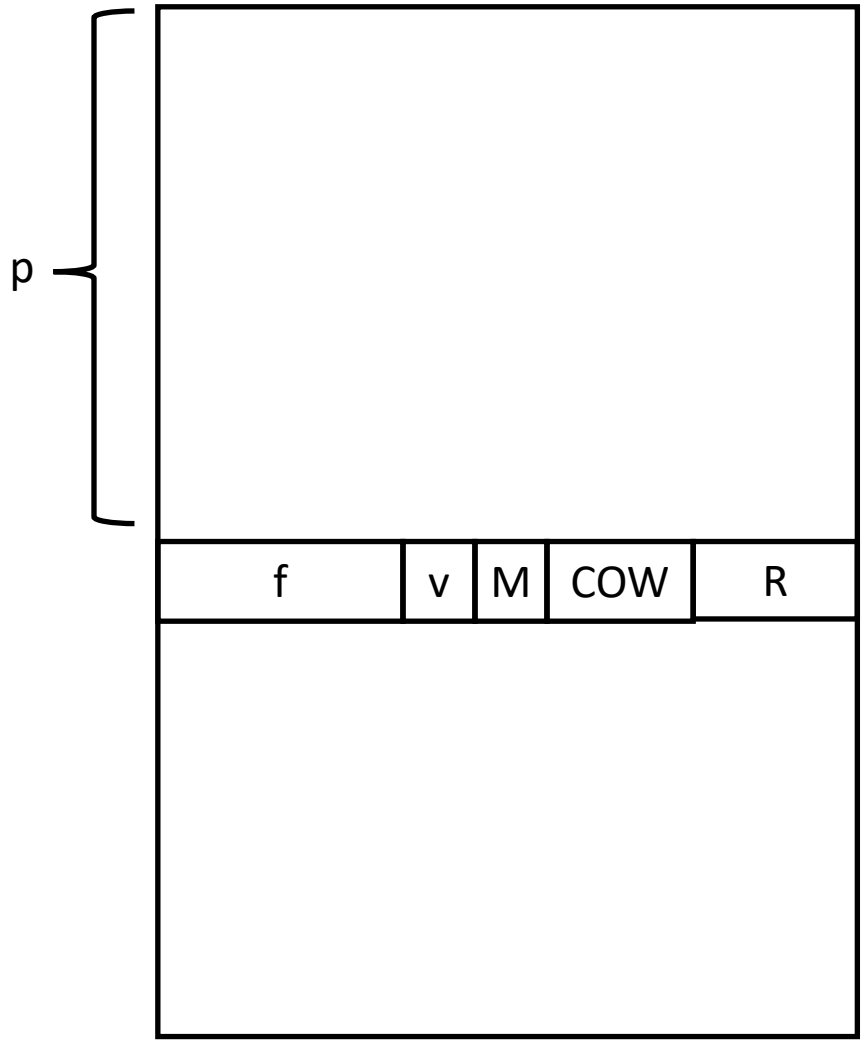
Least Recently Used – cont.

- In stack algorithms, a set of pages in memory for *N* frames is always a subset of the set of pages that would be in memory if *N + 1* frames were used.
  - If the number of frames increases from N to N+1, then the N *stacked* pages will still be the most recently referenced and remain in memory + one more --> predictable and has no randomness.
- This statement is not true for FIFO.



9.4.5 LRU Approximation Algorithms

- LRU needs special hardware and still slow
  - The OS cannot be invoked in every page access to update the reference bit → hardware must update that bit.
- **Reference bit**
  - With each page associate a reference bit:
    - All initialized to 0 (I.e. for all pages)
  - When page is referenced, set to 1 (by H/W)
  - Replace a page whose reference bit = 0 (if one exists).
    - However, we may have more than one page with a reference bit of 0 and we do not know which one is older



Page Table

9.4.5 LRU Approximation Algorithms – cont.

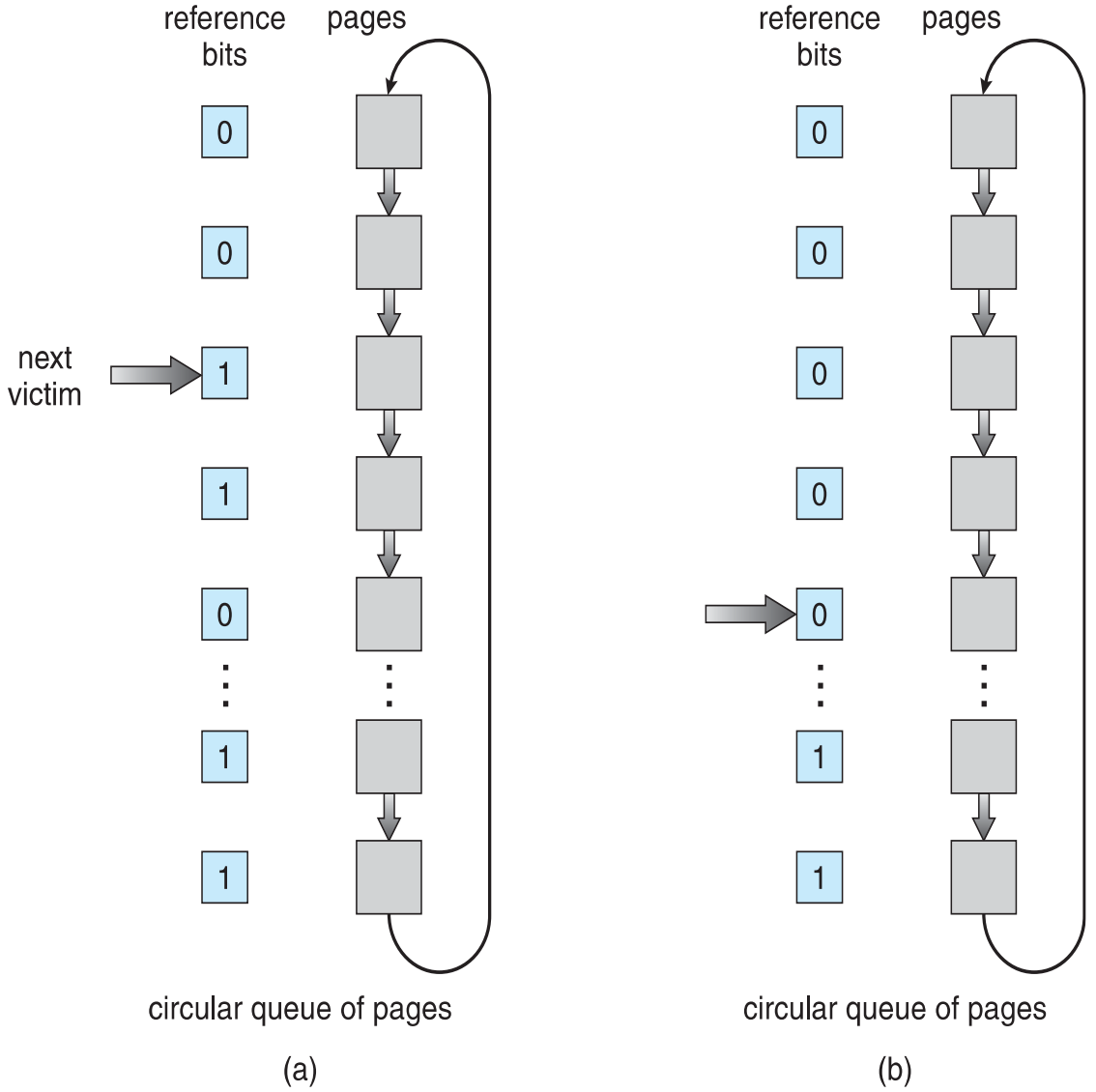
Additional-Reference-bits algorithm

- We can keep an 8-bit shift register for each page in a table in memory.
  - At regular intervals (e.g. 100 milliseconds), a **timer interrupt** transfers control to the operating system.
  - The OS right-shifts the register by one and then inserts the new reference bit (written by H/W on each page access, cleared by OS at end of the timer interrupt) on bit 7 → The 8-bit shift registers contain the history of page use for the last eight time periods.
  - A shift register containing **00000000** indicates that the page has not been used for eight time periods.
  - A page that is used at least once in each period has a shift register value of **11111111**.
  - A page with a history register value of 11000100 (value = 0xC4) has been used more recently than one with a value of 01110111 (value = 0x77).
- On page faults, the OS find the page with the lowest value is the LRU page → it can be replaced
- When multiple pages have the same value → use the FIFO method to choose among them.

9.4.5 LRU Approximation Algorithms – cont.

Second-chance algorithm

- No shift registers involved, but still requires the hardware reference bit.
- Essentially a FIFO algorithm with modifications – On a page fault:
  - A circular buffer implements the FIFO (looks like a **clock** whose hands point to the next page to examine)
  - On page fault, if currently examined page has:
    - Reference bit = 0 -> replace it.
    - Reference bit = 1, then:
      - set reference bit 0, leave page in memory → give it a ***second chance.***
      - Repeat for next page, subject to same rules



Second-Chance (clock) Page-Replacement Algorithm

LRU Approximation Algorithms – cont.

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 nor modified – best page to replace

2.(0, 1) not recently used but modified – not quite as good, must swap the page 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 is called for, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

9.4.6 Counting Algorithms

- Keep a counter of the number of references that have been made to each page (since start of the reference string)
  - Since it is required to update the counter in each reference, this is done by the hardware → too complex → Makes this class of algorithms uncommon

- **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
  - thus eject page with MFU count assuming it was brought in too long ago and less likely to be needed again.

9.4.7 Page-Buffering Algorithms

- Always **keep a pool of free frames**, thus a frame is readily available when needed instead of being searched for when the page fault takes place:
  - Read page into free frame (swap in)
  - Select victim to evict (using some replacement algorithm)
  - When convenient, evict victim, i.e. **swap out on the background** and add its frame to free pool.
- Possible optimization:
  - keep list of modified pages
  - When backing store otherwise idle, write modified pages there and set to non-dirty

  i.e. try to keep as many pages as possible clean (**clean the pages on the background**)
  → clean pages do not need to be swapped out.
- Another possible optimization:
  - keep free frame content intact and note which pages reside on them (i.e. **don't swap out**)
  - If one of those pages is referenced again before the frame got a new resident, then there is no need to load contents again from disk.
  - Generally useful in reducing penalty if the wrong victim frame was selected.

9.4.8 Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
  - However, some applications have better knowledge – e.g. databases
  - Additionally, in database systems, the same memory may be buffered twice:
    - OS keeps copy of page in memory as disk I/O buffer
    - Database 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 allows a database application for example to have full access to a raw disk partition.
  - Bypasses buffering, filename search, locking, etc

9.5 Allocation of Frames

- Each process needs ***minimum*** number of frames
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - Dynamic allocation
- Many variations on these two schemes.

9.5.2 Fixed (static) vs variable (dynamic) Allocations

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - **Static** (fixed) allocation.
  - Keep some as free frame buffer pool

- **Proportional allocation** – Allocate according to the size of process
  - **Dynamic**, reacts to changes in the degree of multiprogramming or changes in process sizes.

— s_i = size of process p_i

— $S = \sum s_i$

— m = total number of frames

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

$m = 62$

$s_1 = 10$

$s_2 = 127$

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

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

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- Alternatively, allocate based on priority + size

9.5.3 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 (depending on other processes running in the system)
  - Greater throughput so more common

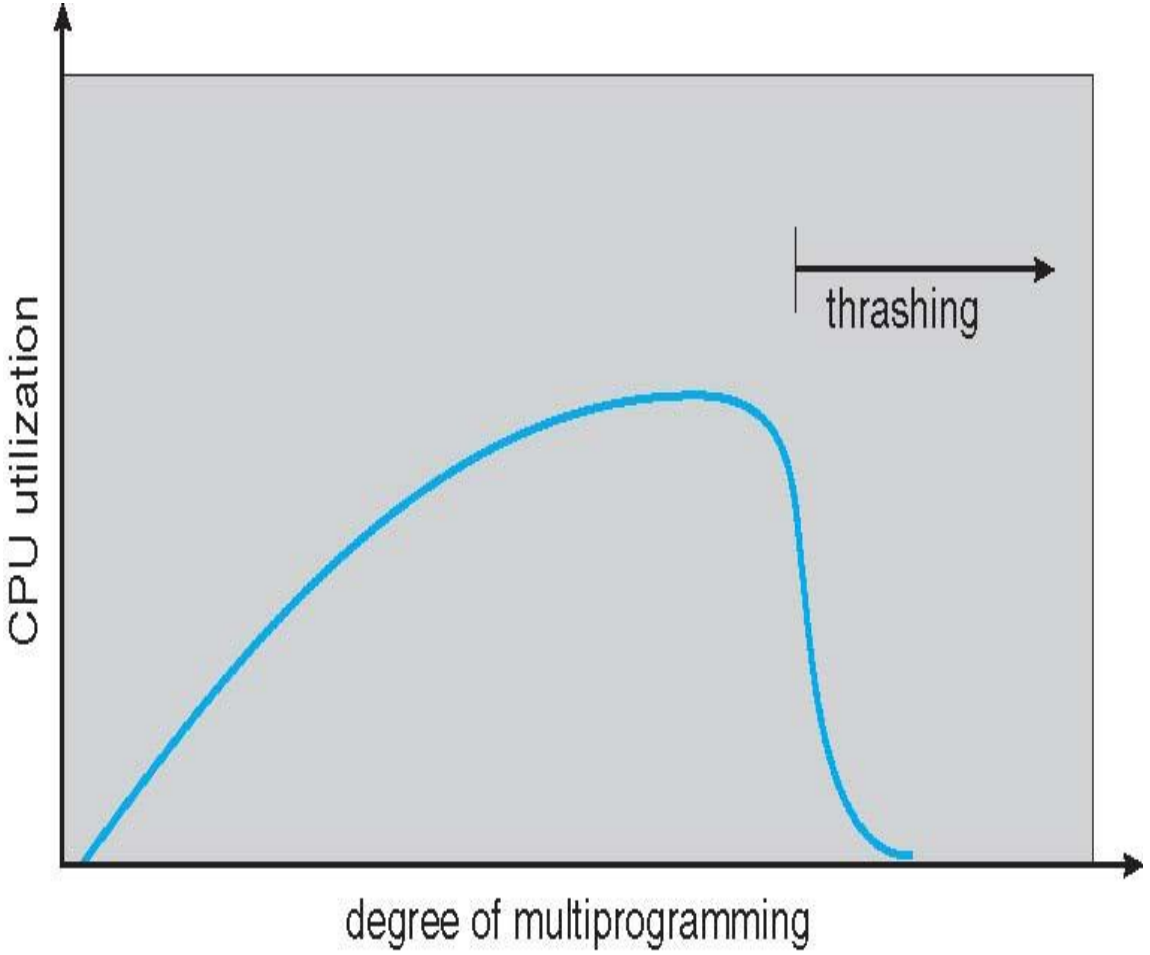
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

9.5.4 Non-Uniform Memory Access

- So far, all memory accessed equally, and we allocated numbers of frames to processes, without specifying their locations.
- 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 all threads of a process on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used by scheduler and pager
    - When possible schedule all threads of a process and allocate all memory for that process **within the same lgroup**

9.6 Thrashing

- If a process does not have “enough” frames allocated, 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 may think it needs to increase the degree of multiprogramming
    - Another process added to the system

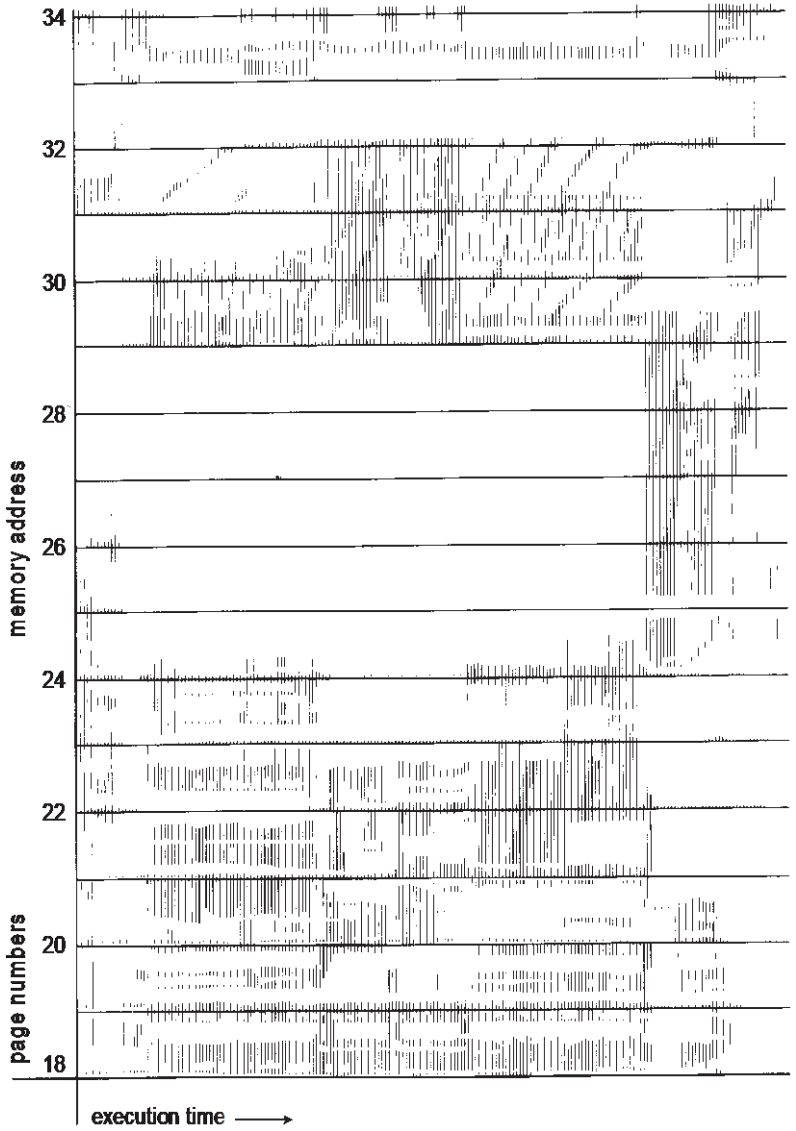


- **Thrashing** ≡ a process is busy swapping pages in and out

Demand Paging and Thrashing

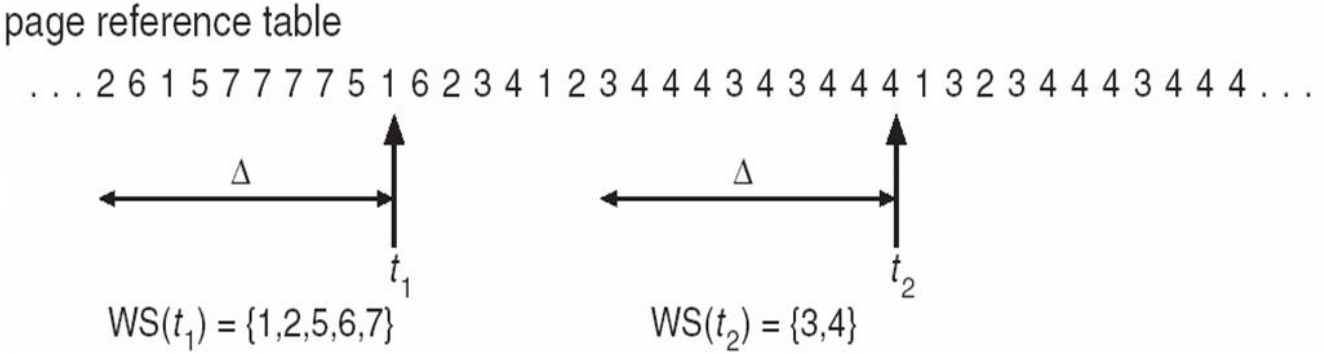
- Why does demand paging work?
  Locality model
  - As a process executes, it moves from locality to another. A locality is a set of pages that are actively used together.
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size allocated
  - Thus allocate the process the frames it needs to avoid thrashing.
  - Limit effects by using local or priority frame allocation (priority frame allocation is global)



Working-Set Model

- **$\Delta \equiv$ working-set window** $\equiv$ a time window specified by a fixed number of page (or memory) references. Note that this **is a sliding window**.
  Example: 10,000 instructions (or 10,000 cycles $\equiv$ 10,000 page references)
- ***WSS_i* (working set size of Process *P_i*)** =
  total number of pages referenced in the most recent Δ (varies over time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program



- $D = \Sigma\ WSS_i \equiv$ total demand frames for all processes
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes
  - **Which process scheduler** does that (short term, medium term or long term scheduler?)

Keeping Track of the Working Set

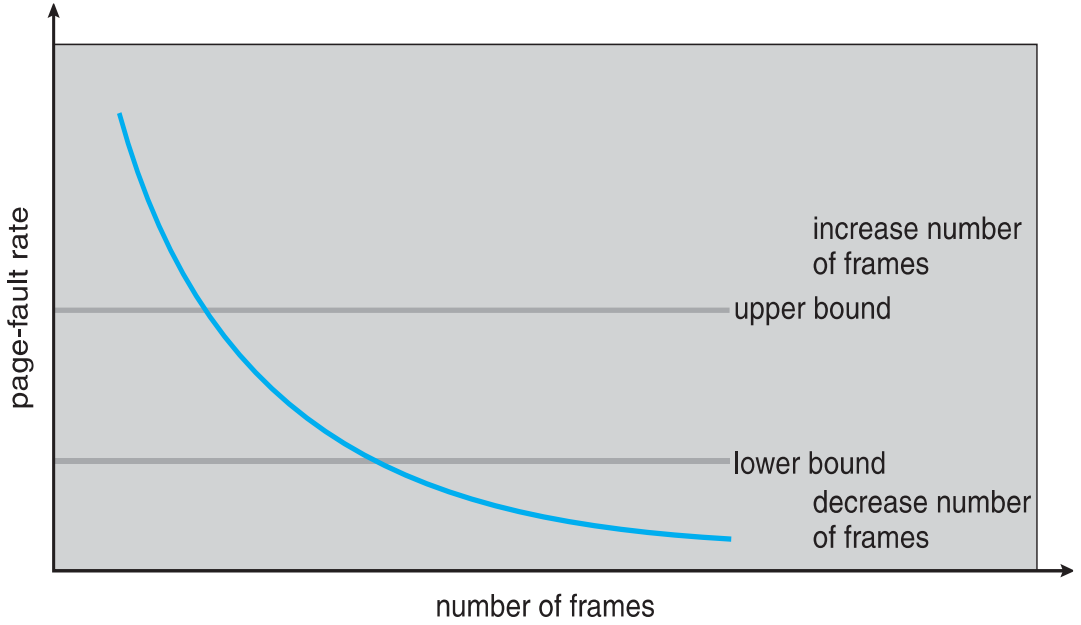
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bit that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 8 bits and interrupt every 1250 time units

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bits that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Working set slides by $\Delta/2$
- Improvement = 8 bits and interrupt every 1250 time units
  - Working set slides by $\Delta/8$

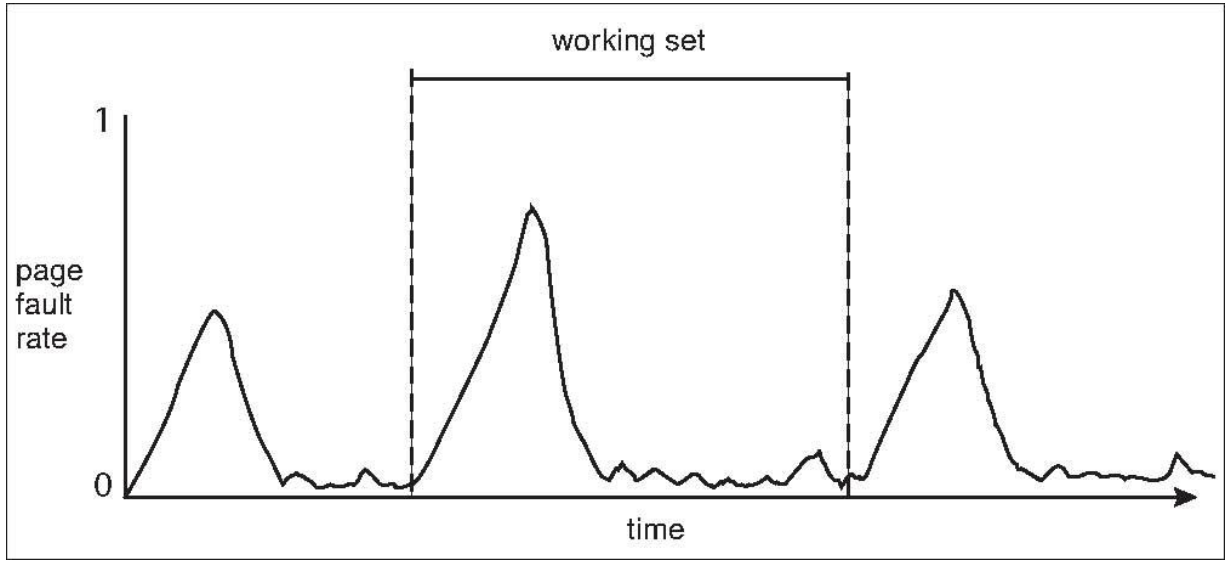
Page-Fault Frequency

- 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



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
  - Page faults peaks are when a process is changing locality (or working set)
- Peaks and valleys over time

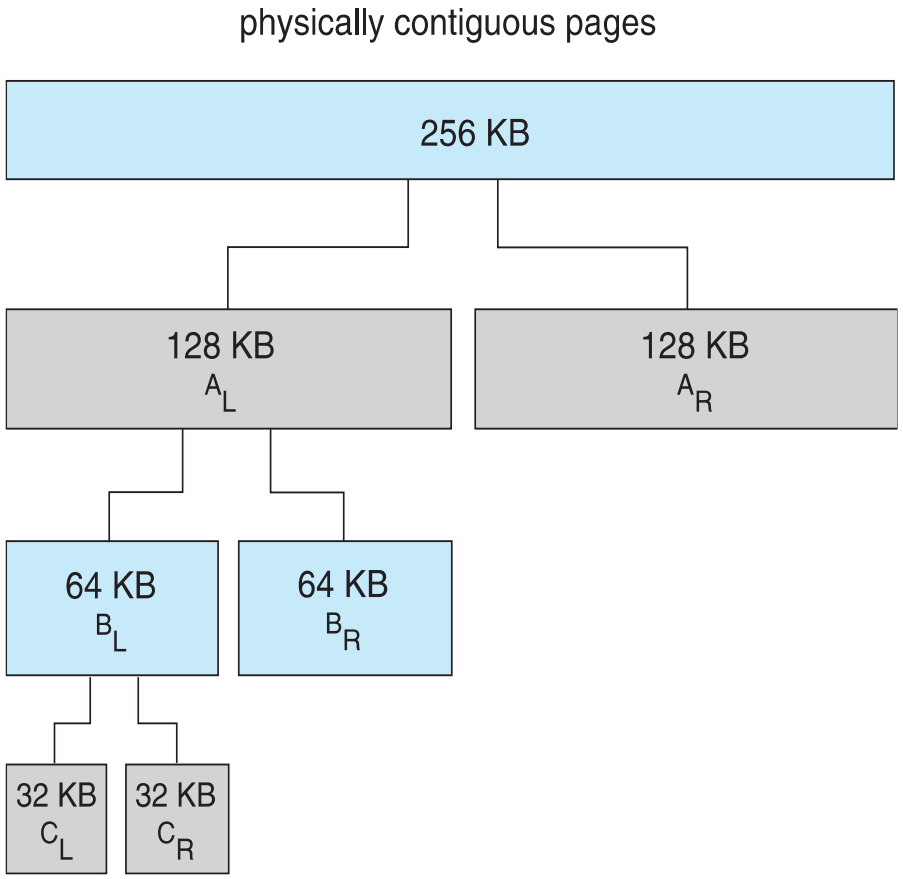


9.8 Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some, actually much of the kernel memory allocations needs to be **contiguous**
    - e.g. for device I/O where DMA is used. Some DMA hardware is simple and only operates on contiguous memories (source or destination), whereas other DMA hardware may be complex enough to handle linked lists of memory blocks for I/O transfers.

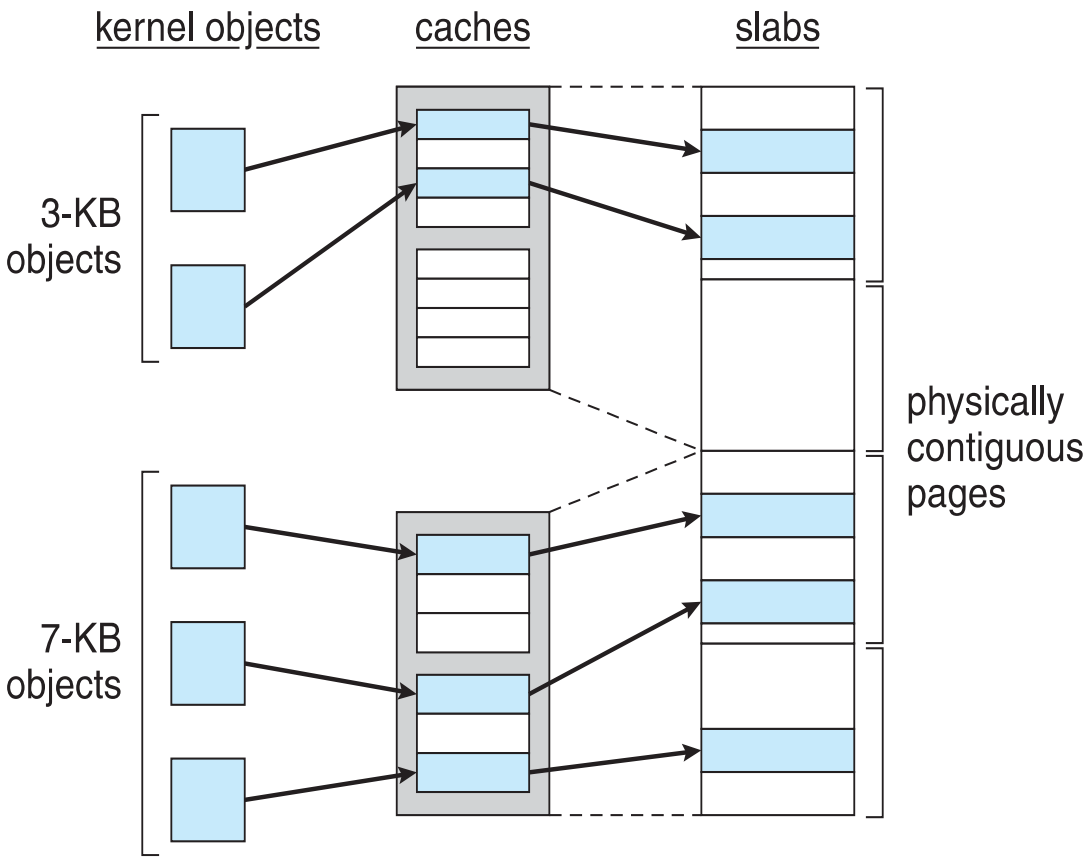
9.8.1 Buddy System

- 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
- Advantages:
  - Blocks may be arranged as a tree for quickly finding the requested empty block.
  - Can **quickly coalesce** unused chunks into larger chunk
- Disadvantage - **internal fragmentation**



9.8.2 Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
  - A Single cache for each unique kernel data structure, e.g. a semaphore data structure or a process descriptor structure, etc.
  - Each cache is filled with **objects** – instantiations of the data structure
  - For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects.
- 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 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 (e.g. embedded systems)
    - SLOB = Simple List of Blocks – maintains 3 list objects for small, medium, large objects
    - Objects are allocated from the appropriate list using a first fit approach.
    - May suffer from some internal fragmentation (i.e. first fit is not perfect).
  - SLUB allocated started in Linux 2.6.24 - Improved performance over SLAB by reducing some overheads:
    - Moves some metadata stored in slabs into the page structure
    - Removes per-CPU queues maintained for objects of each cache – big savings for systems with a large number of CPUs.