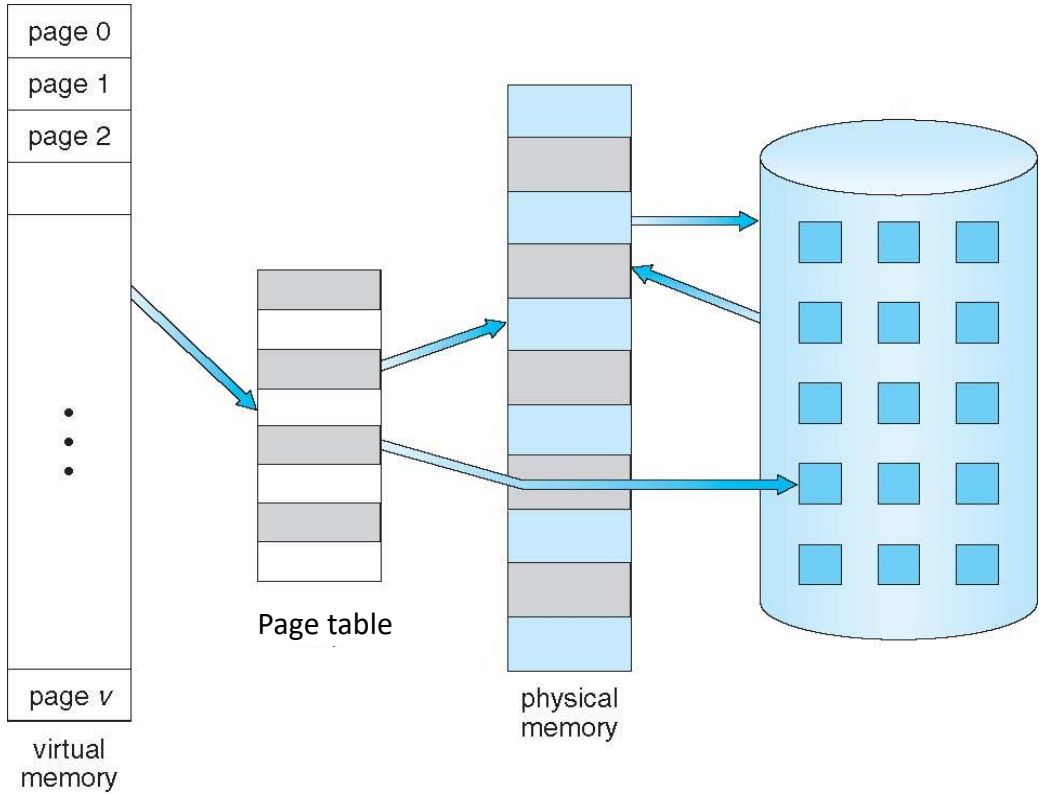
9.1 Virtual memory

- Motivation:
  - Code needs to be in memory to execute, but **entire program is rarely needed** at the same time:
    - Error code, unusual routines, large data structures, etc.
    - Programs usually have a **locality of reference**, in which only a portion of the program is actually being invoked within each time window, e.g. loops.
- Consider ability to **partially load and execute** programs
  - Program no longer constrained by limits of physical memory and may thus have a **virtual address space** that is much larger than the actual physical memory.
  - Each program takes less memory while running → a higher degree of multiprogramming may be achieved → increased CPU utilization.
  - Compared to swapping, less I/O time is then needed to swap processes in/out of memory → each user process runs faster
- Supported by segmentation or paging.

Virtual memory – cont.

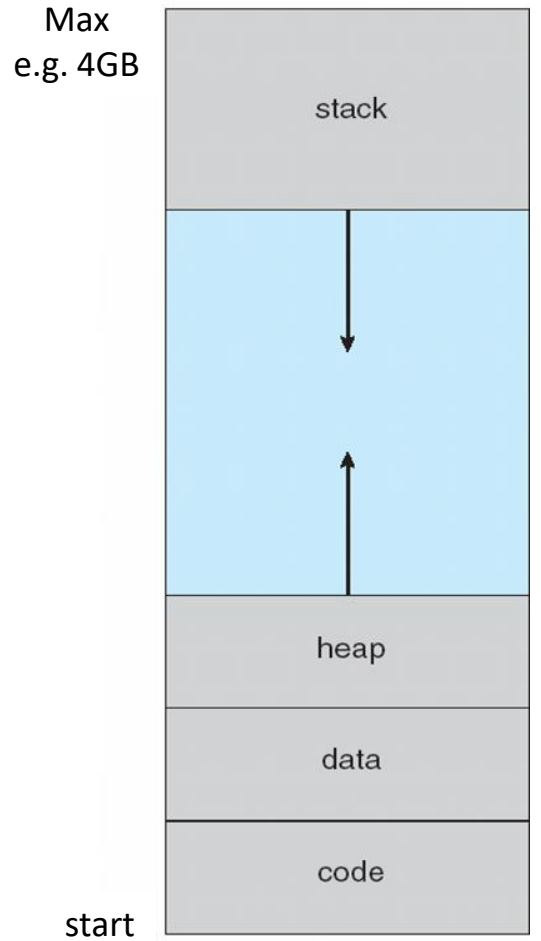
- What we already know from paging:
  - **Virtual address space** – logical view of how process is stored in memory
    - Usually **start at address 0**,
    - **Contiguous** addresses until end of space
  - Meanwhile, **physical memory** is organized in page frames. MMU must map logical (aka virtual) 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

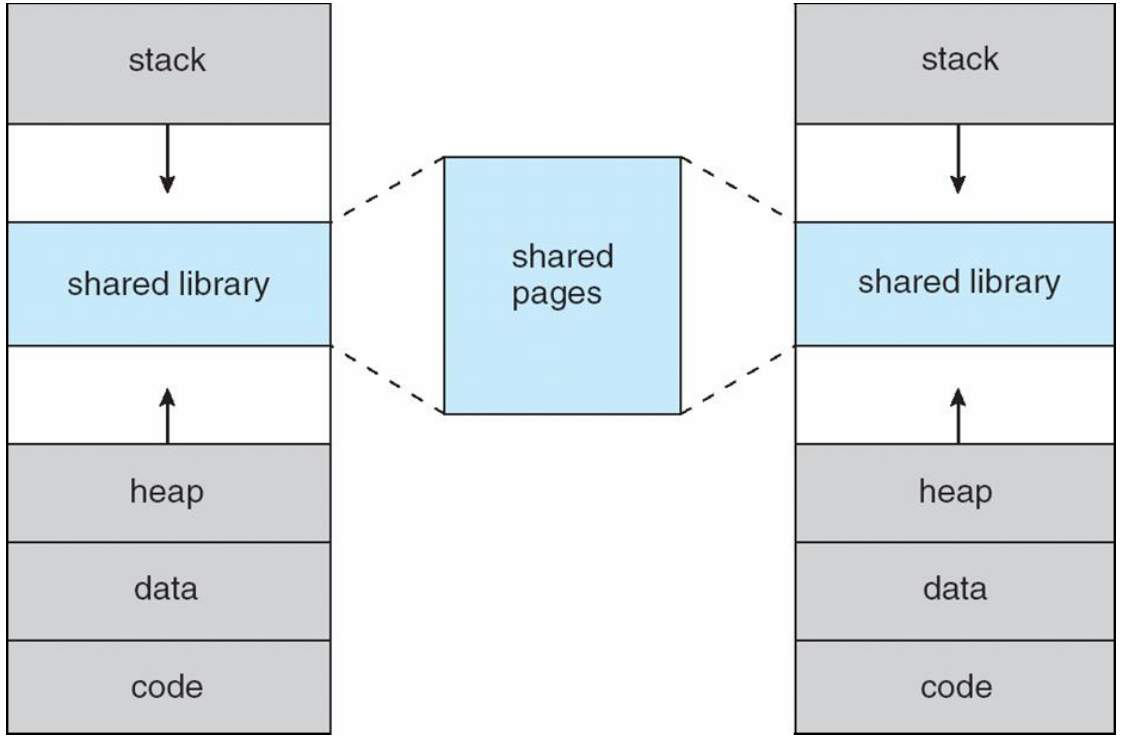
Virtual-address Space

- 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 and require additional pages.
- Note that in 32-bit Linux, the upper 1 GB of every process' virtual space is reserved for the kernel (code, data, heap and stack).
  - Shared amongst processes
  - Protected from user mode access.



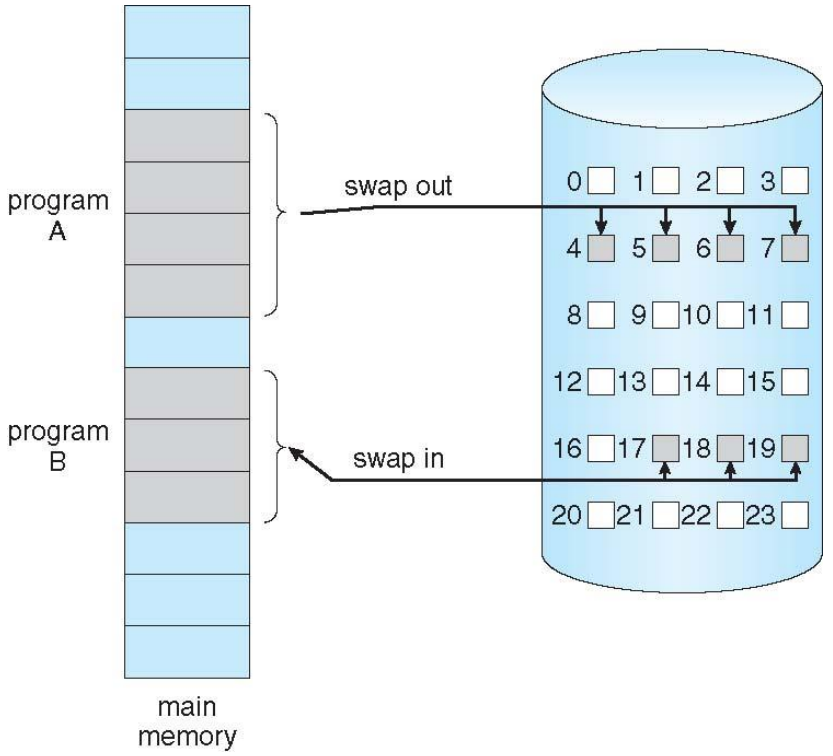
Shared libraries using virtual memory

- 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**



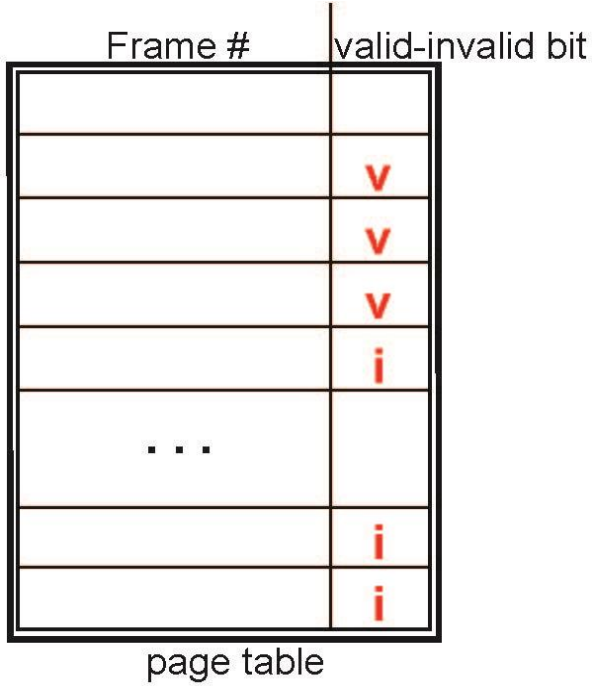
Memory swapping

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

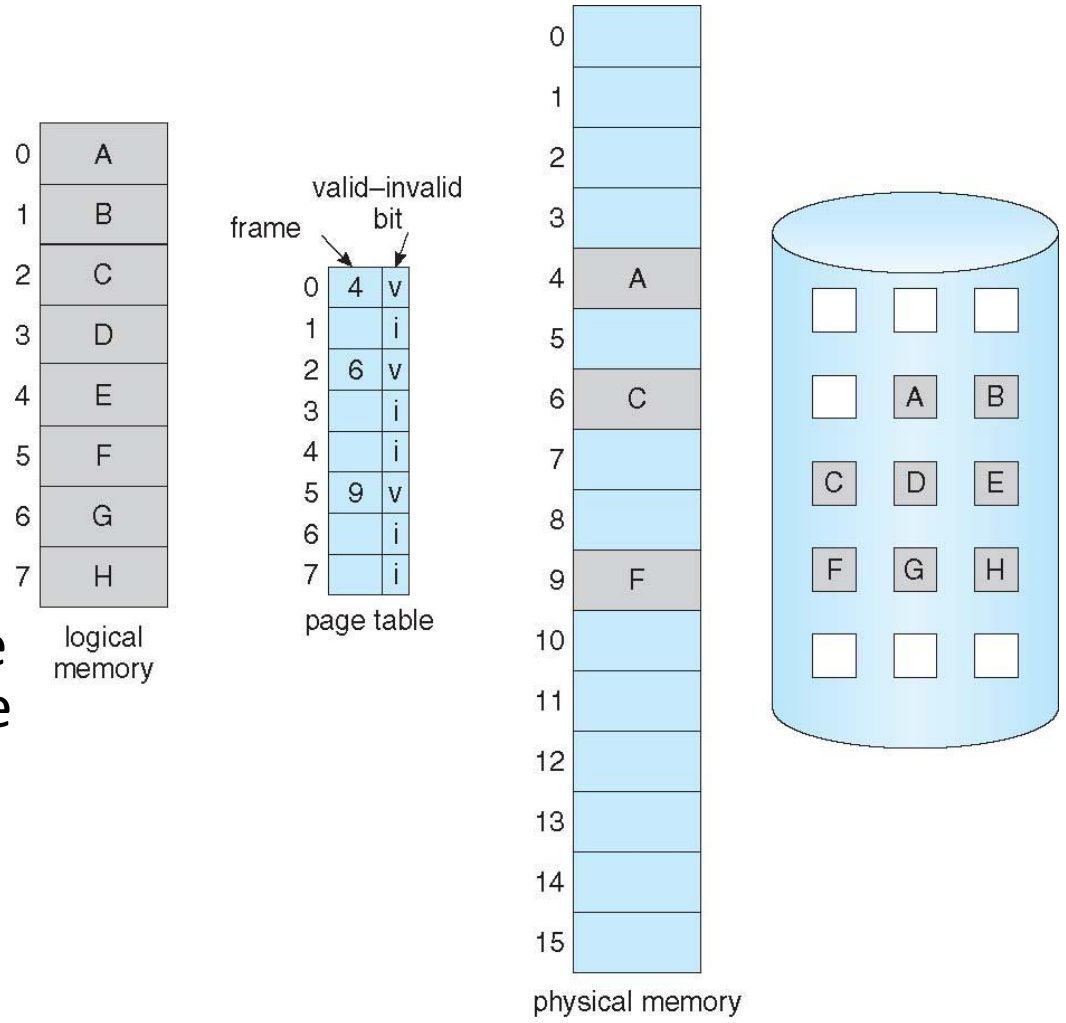
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- 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:

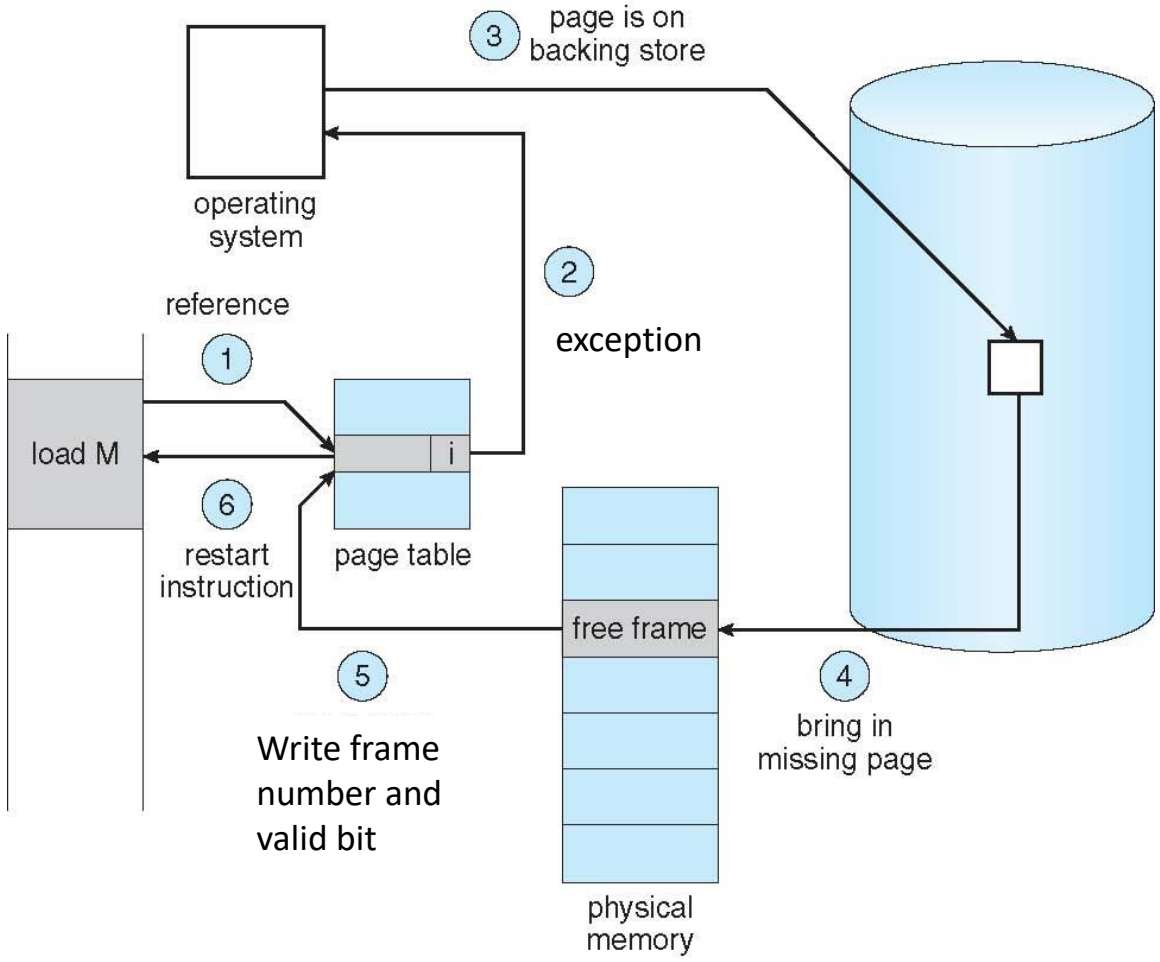
a) Operating system locates the page on the page table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory, then continue to step b

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

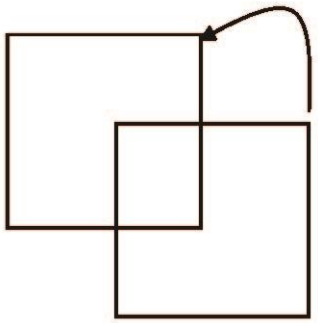


NOTE: Contrary to what "fault" might suggest, **page faults** do not represent errors, unless the page is not in the process' address space, and only in such case that it may be a true fault, **a segmentation fault**, which is an error.

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)
  - Would this occur in a CISC or a RISC machine?
  - block move  ent/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 CPU registers (for interrupt 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 (when I/O has completed)
9. Save the CPU registers (for interrupt context switching, interrupting the other process)
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. A) Either resume the other process (restoring its CPU registers). Scheduler invoked during normal timer ticks. OR

    B) Invoke the scheduler immediately. Scheduler may thus perform process context switching if it decides to run the page-faulting 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!!

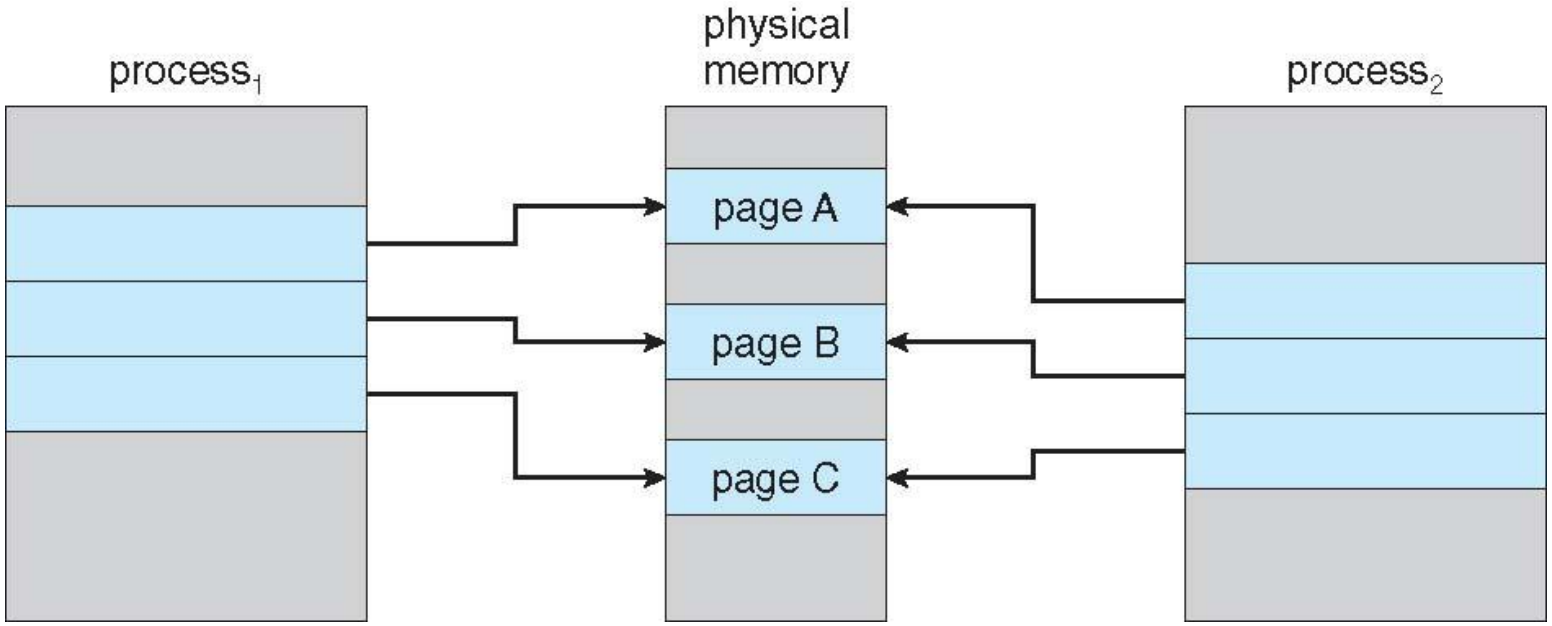
- If we need to limit the performance degradation to < 10 percent
  - $220 > 200 + 7{,}999{,}800 \times p$
    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$ (i.e. 2.5 in a million)
  - < one page fault in every 400,000 memory accesses

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

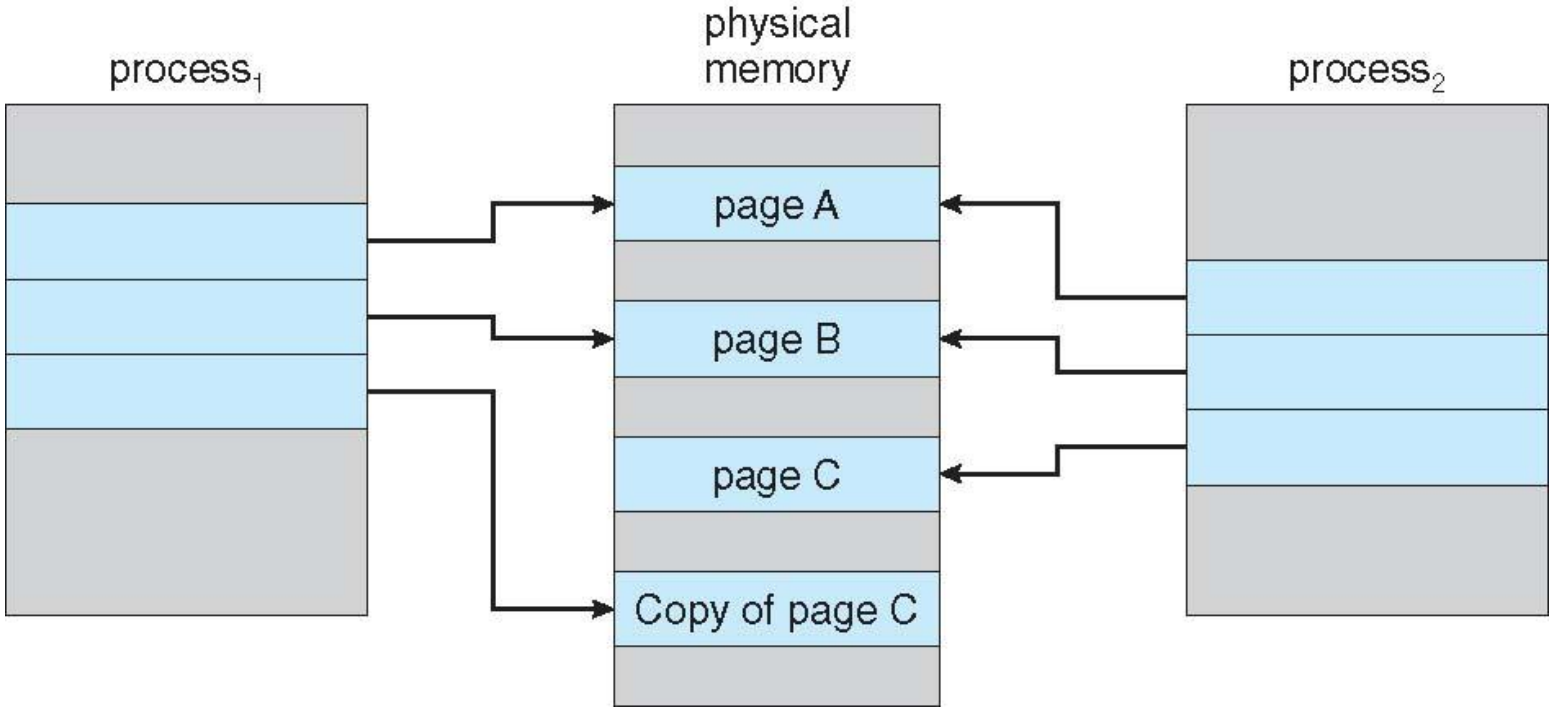
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory (***except for stack pages***).
  - If either process modifies a shared page, only then is the page copied



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 (swap out) 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

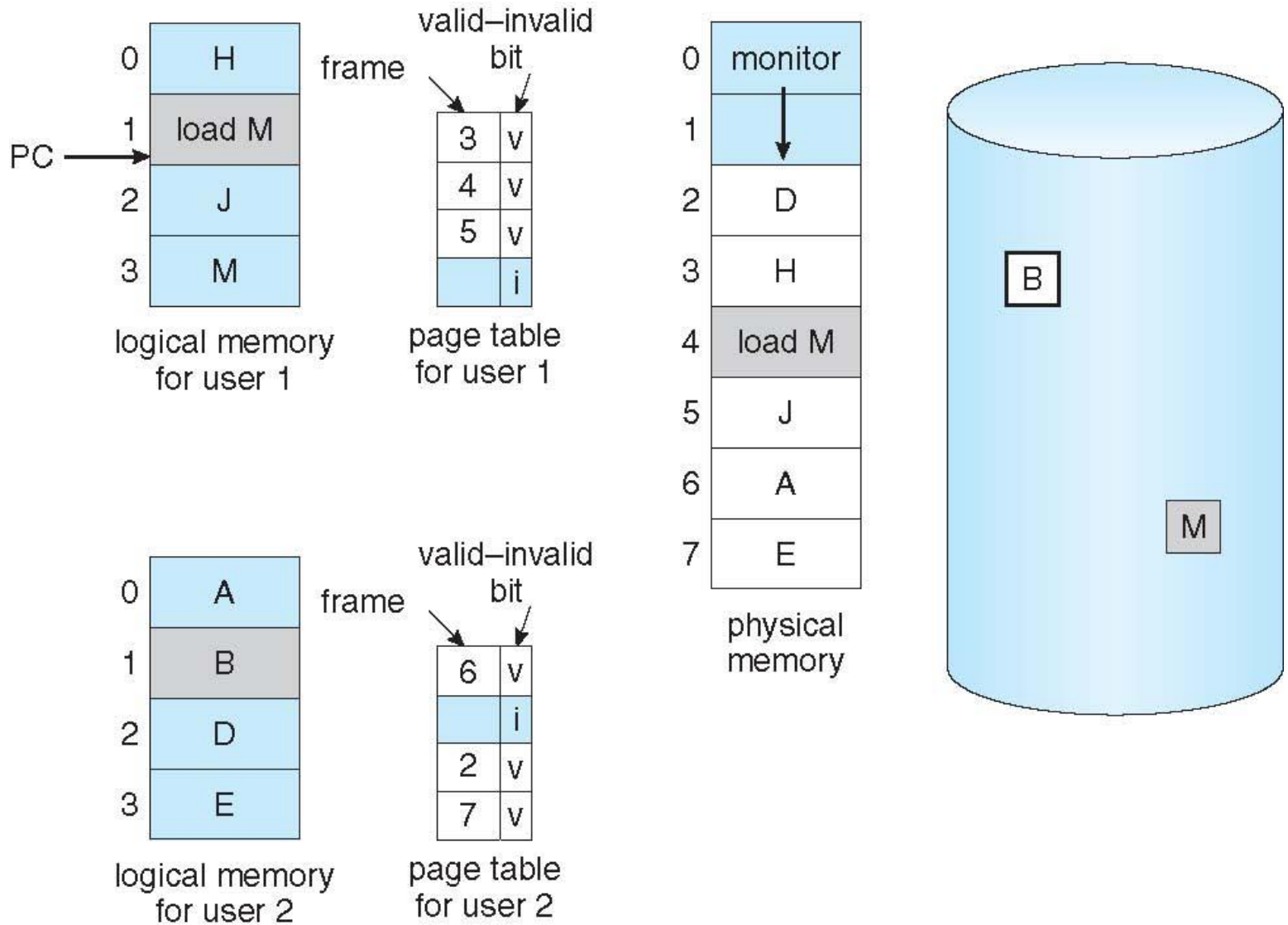
- If the page size is 256, then the page trace is:

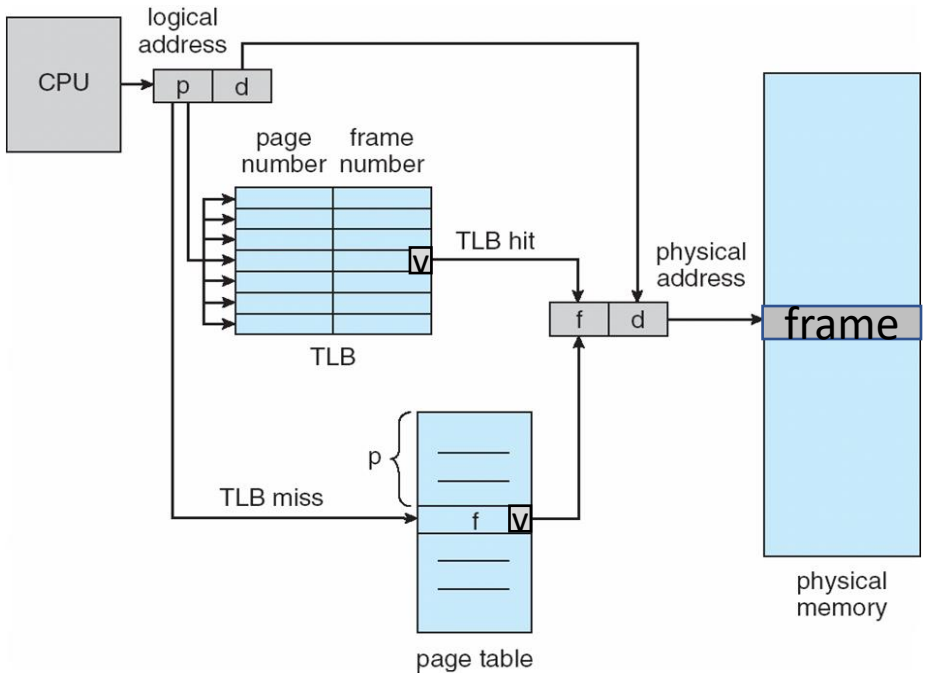
  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

9.4 Page replacement

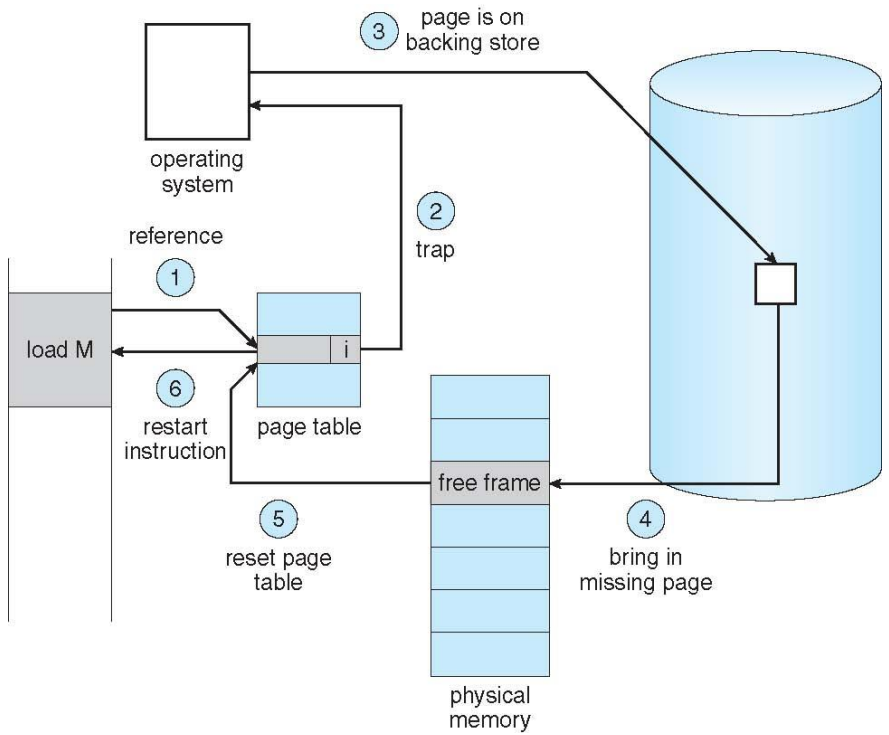
- What happens if there are no free frames? Options are:
  - Terminate process – not a good idea
  - Page replacement – find some page in memory (that is least used for example, determined by some algorithm) and page it out.
    - Performance – want an algorithm which will result in minimum number of 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:

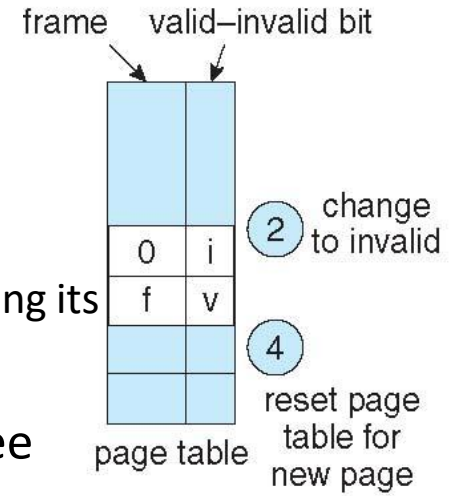
1. Find the location of the desired page on disk

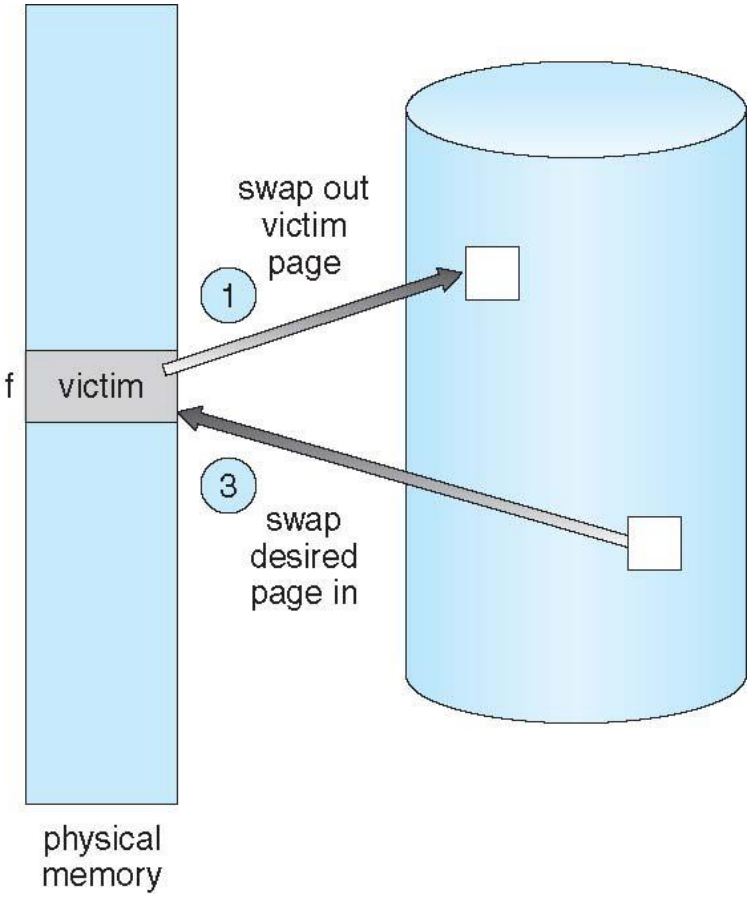
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty (after changing its page table entry to invalid)

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

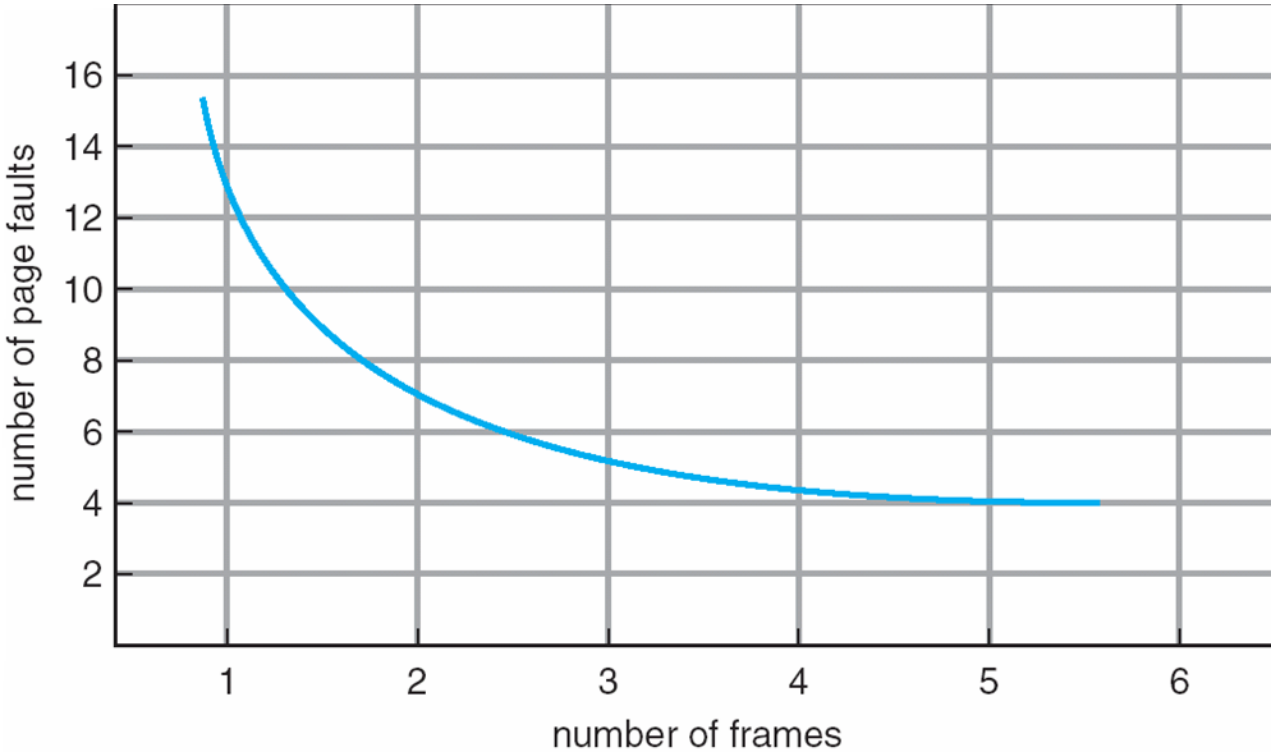
4. Continue the process by restarting the instruction that caused the trap

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





Graph of page faults versus the number of allocated frames

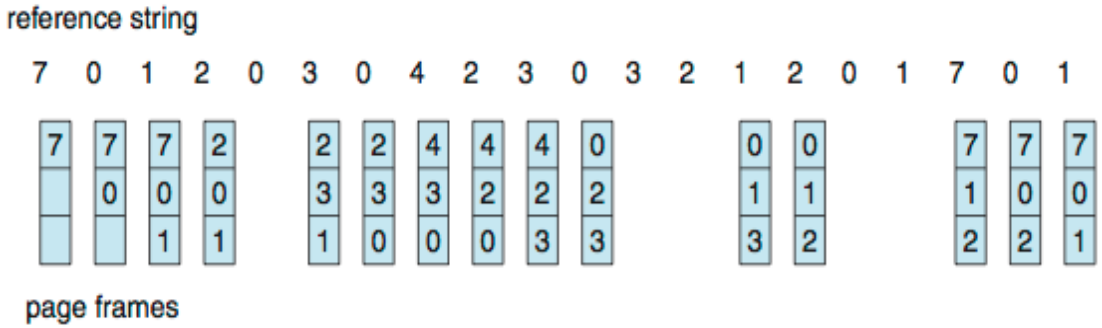


9.4.2 First-In-First-Out (FIFO) Algorithm

- 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**