Operating Systems

Virtual Memory Management

Virtual Memory Management

- VIRTUAL MEMORY MANAGEMENT: Need for Virtual Memory management – Demand Paging – Copy on write - Page Fault handling – Demand Segmentation – Combined demand segmentation and paging - Thrashing- working set model.

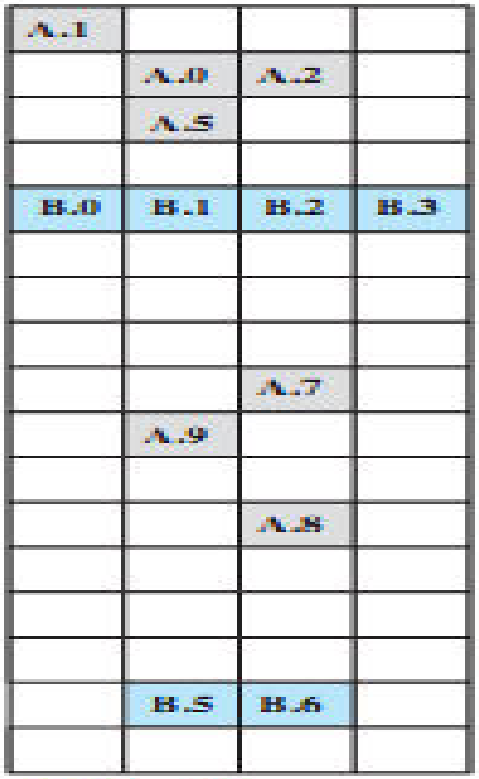
Background (1)

- Code needs to be in memory to execute, but entire program rarely used:
  - Error code, unusual routines, large data structures.
- Entire program code not needed at same time.
- Consider ability to execute partially-loaded program:
  - Program no longer constrained by limits of physical memory.
  - Each program takes less memory while running -> more programs run at the same time:
    - Increased CPU utilization and throughput with no increase in response time or turnaround time.
  - Less I/O needed to load or swap programs into memory -> each user program runs faster.

Background (2)

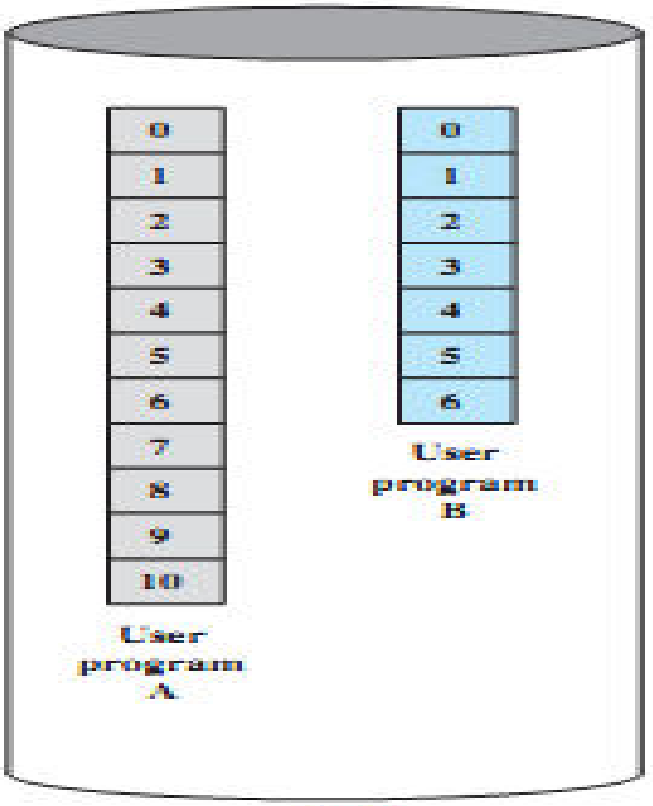
- **Virtual memory** – separation of user logical memory from physical memory:
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
  - More programs running concurrently.
  - Less I/O needed to load or swap processes.

Virtual Memory Components



Main Memory

Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.



Disk

Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the operating system are on disk, as are files.

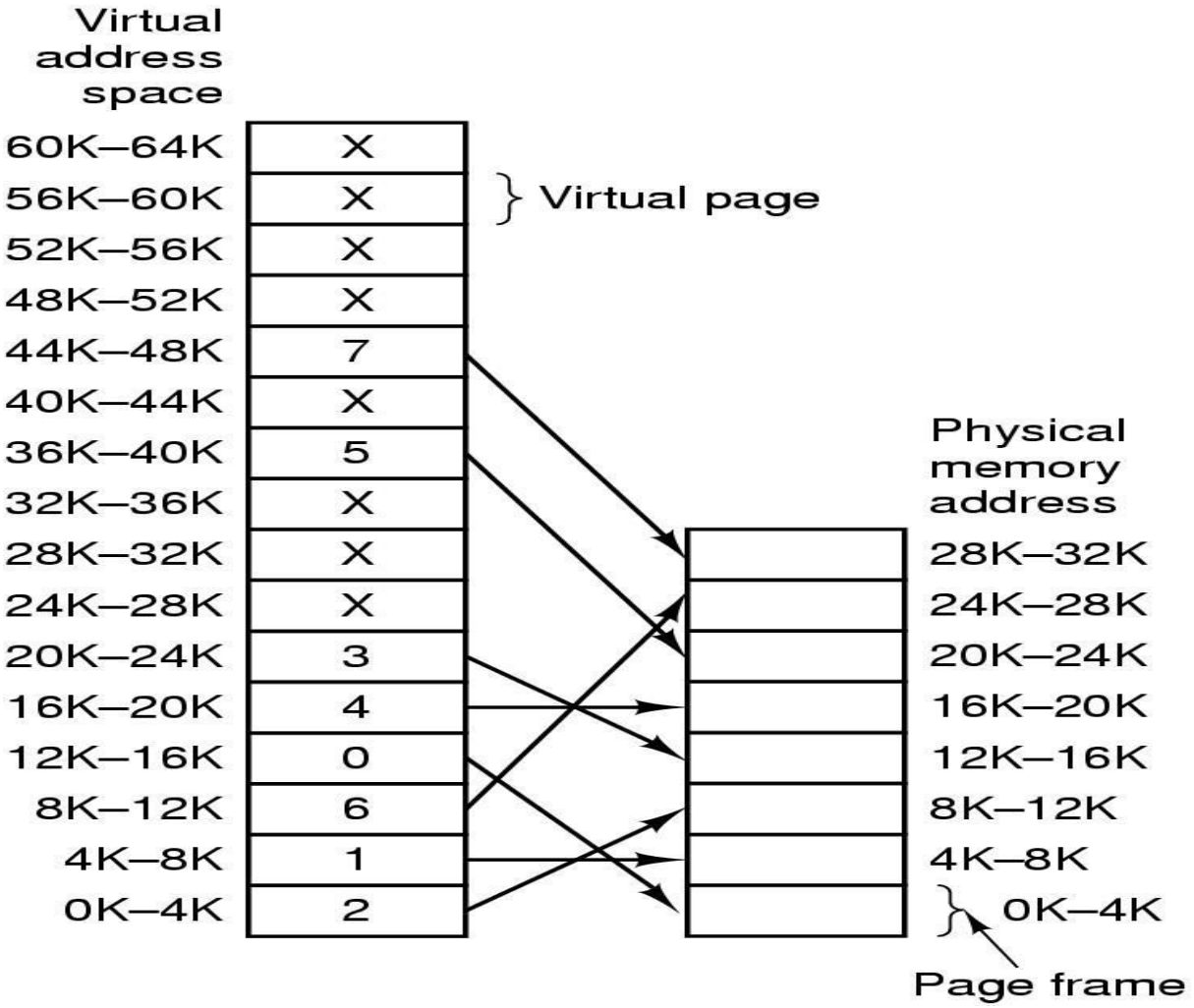
Background (3)

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

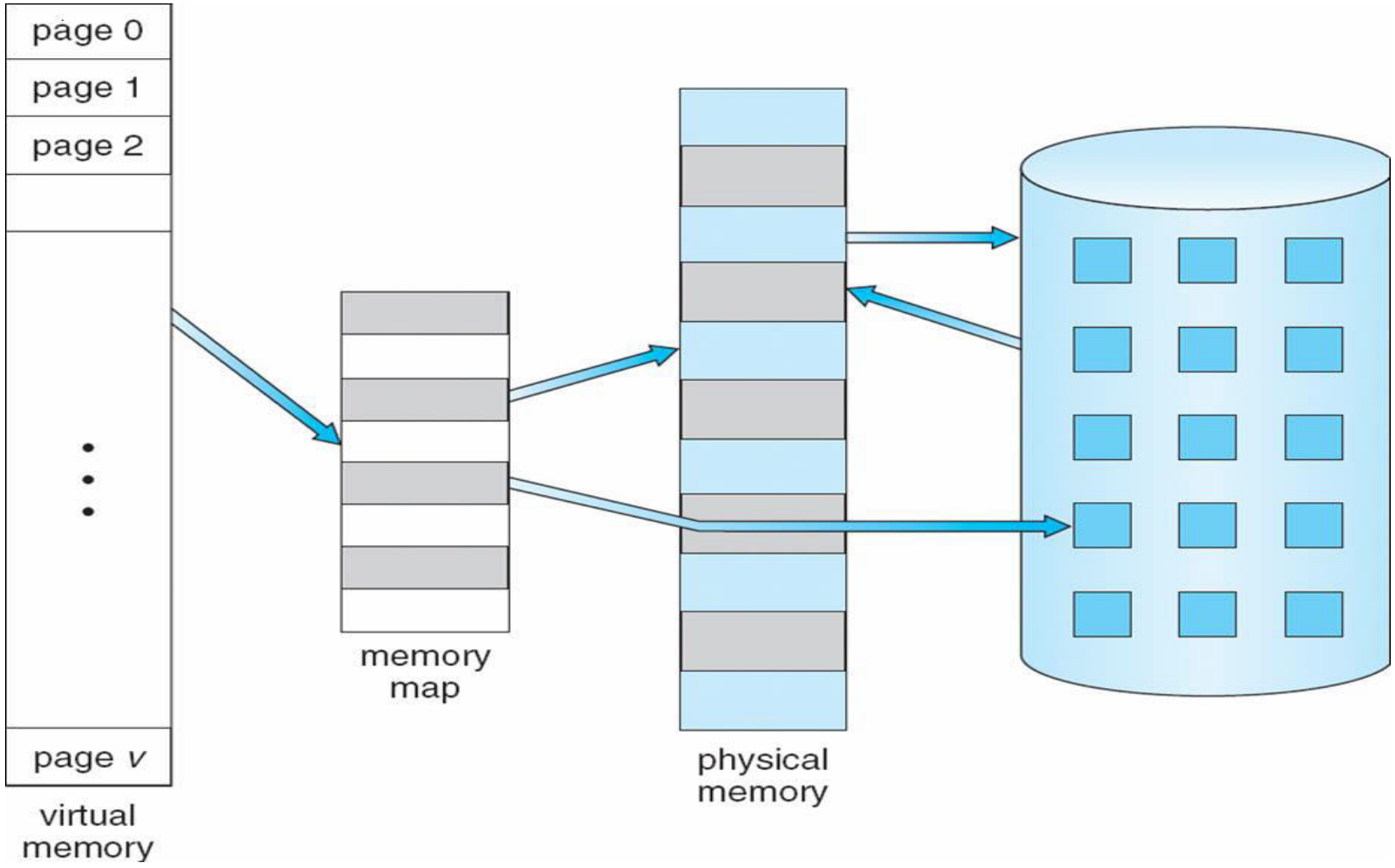
Background (4)

- Based on Paging/Segmentation, a process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory.
- Based on the Locality Principle, all pieces of a process do not need to be loaded in main memory during execution; all addresses are virtual.
- The memory referenced by a virtual address is called virtual memory:
  - It is mainly maintained on secondary memory (disk).
  - pieces are brought into main memory only when needed.

Virtual Memory Example



Virtual Memory that is larger than Physical Memory



Advantages of Partial Loading

- More processes can be maintained in main memory:
  - only load in some of the pieces of each process.
  - with more processes in main memory, it is more likely that a process will be in the Ready state at any given time.
- A process can now execute even if it is larger than the main memory size:
  - it is even possible to use more bits for logical addresses than the bits needed for addressing the physical memory.

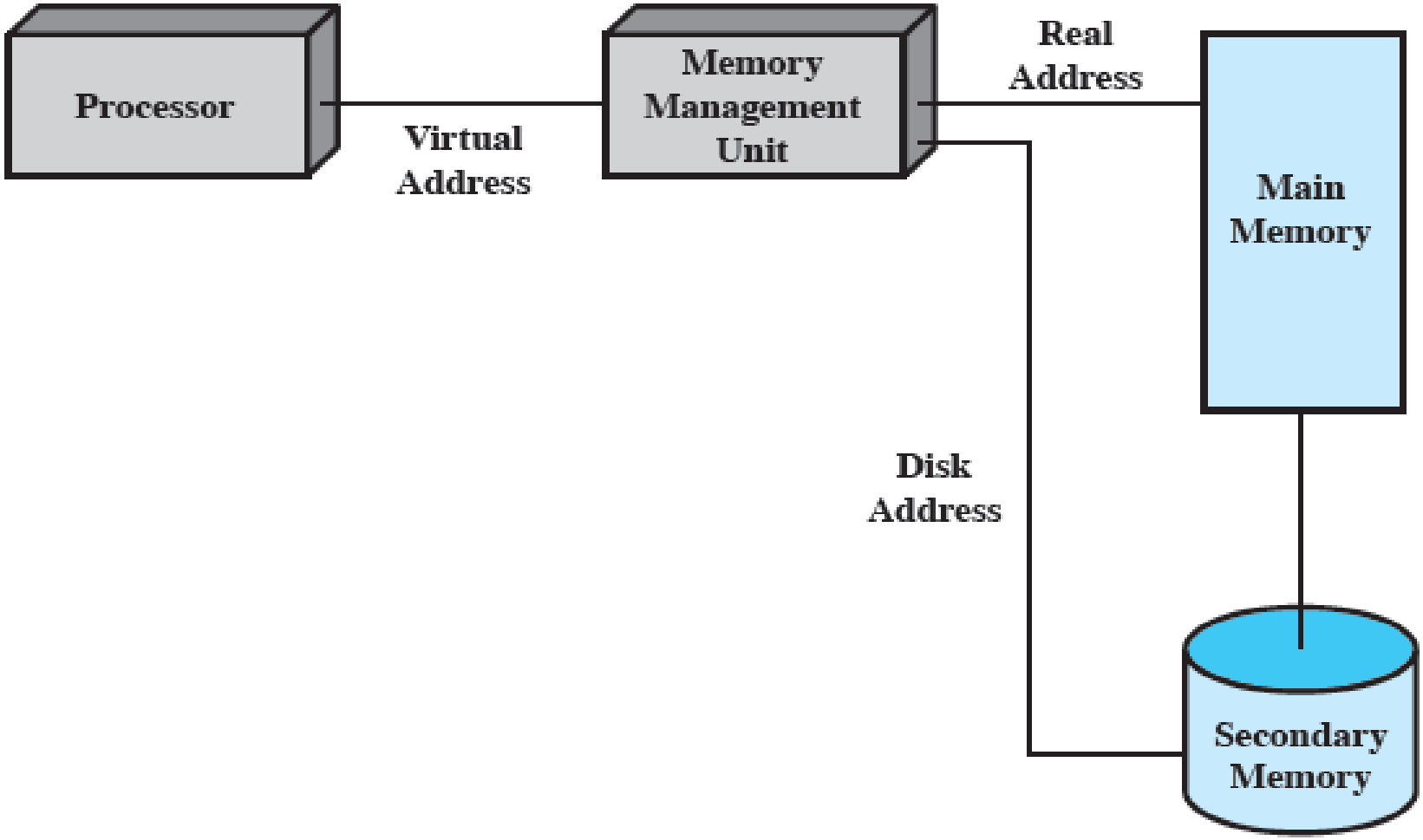
Virtual Memory: Large as you wish!

- Example:
  - Just 16 bits are needed to address a physical memory of 64KB.
  - Lets use a page size of 1KB so that 10 bits are needed for offsets within a page.
  - For the page number part of a logical address we may use a number of bits larger than 6, say 22 (a modest value!!), assuming a 32-bit address.

Support needed for Virtual Memory

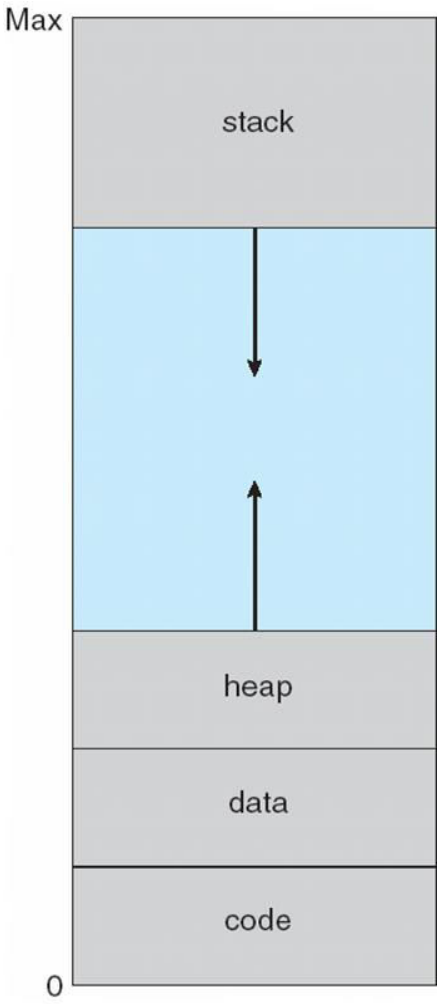
- Memory management hardware must support paging and/or segmentation.
- OS must be able to manage the movement of pages and/or segments between external memory and main memory, including placement and replacement of pages/segments.

Virtual Memory Addressing

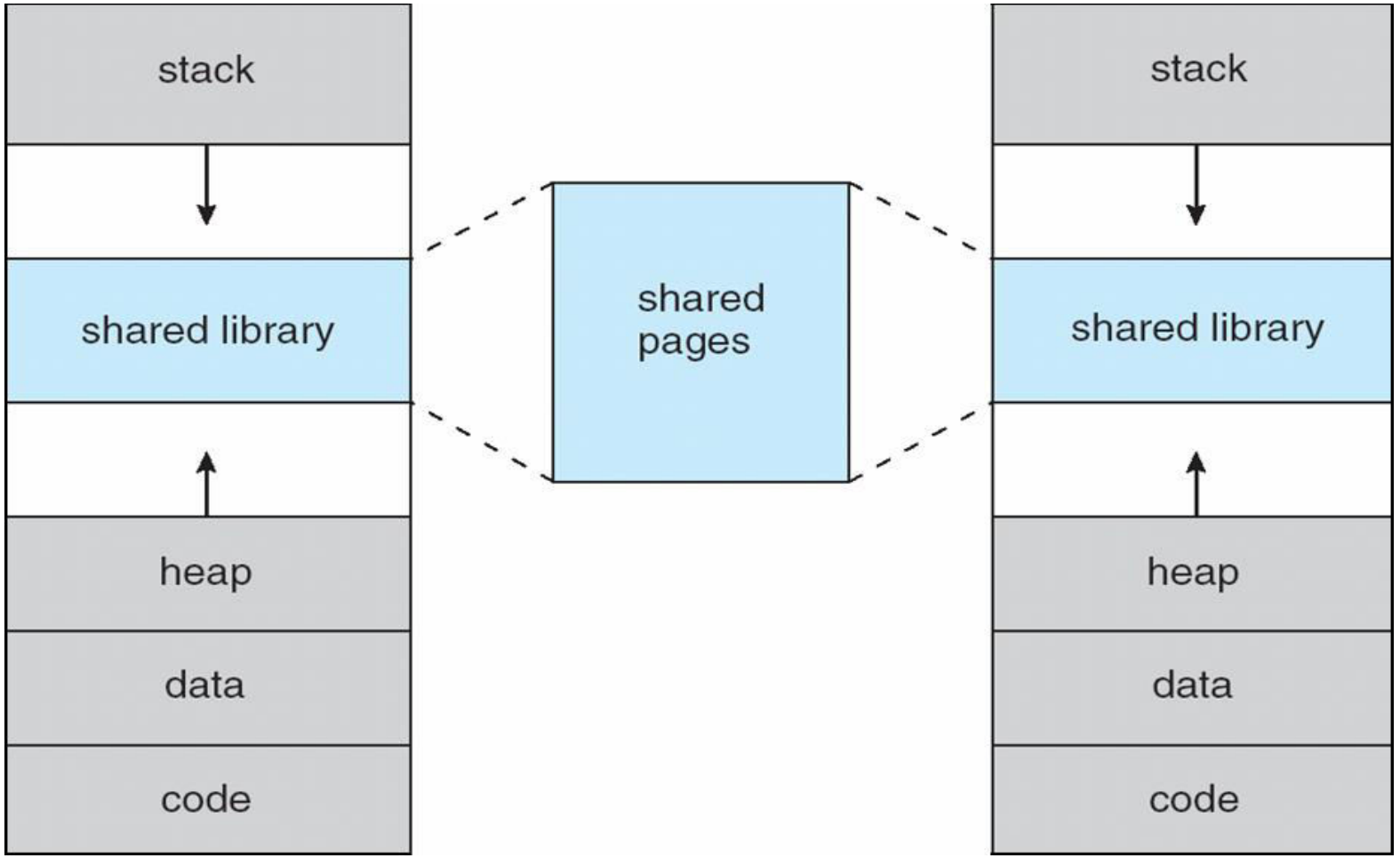


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”:
  - Maximizes address space use.
  - Unused address space between the two is hole.
  - No physical memory needed until heap or stack grows to a given new page.
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space.
- Shared memory by mapping pages read-write into virtual address space.
- Pages can be shared during `fork()`, speeding process creation.



Shared Library using Virtual Memory



Process Execution (1)

- The OS brings into main memory only a few pieces of the program (including its starting point).
- Each page/segment table entry has a valid-invalid bit that is set only if the corresponding piece is in main memory.
- The resident set is the portion of the process that is in main memory at some stage.

Process Execution (2)

- An interrupt (memory fault) is generated when the memory reference is on a piece that is not present in main memory.
- OS places the process in a Blocking state.
- OS issues a disk I/O Read request to bring into main memory the piece referenced to.
- Another process is dispatched to run while the disk I/O takes place.
- An interrupt is issued when disk I/O completes; this causes the OS to place the affected process back in the Ready state.

Demand Paging

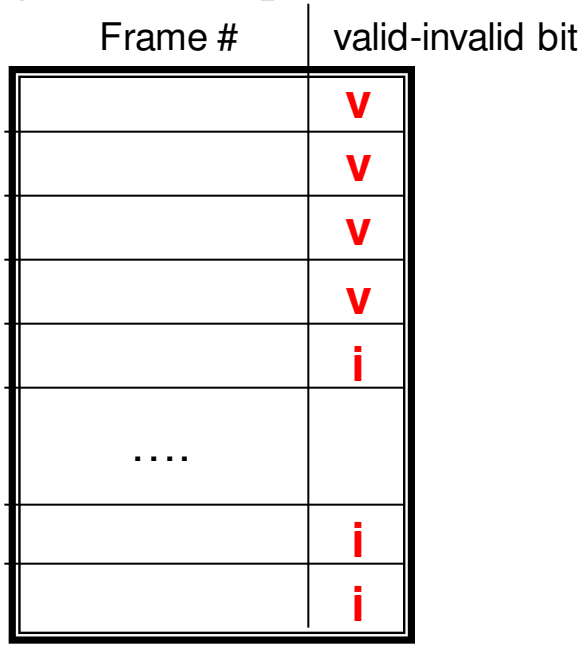
- Bring a page into memory only when it is needed:
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it:
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- Similar to paging system with swapping.
- Lazy swapper – never swaps a page into memory unless page will be needed; Swapper that deals with pages is a pager.

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again.
- Instead, pager brings in 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.

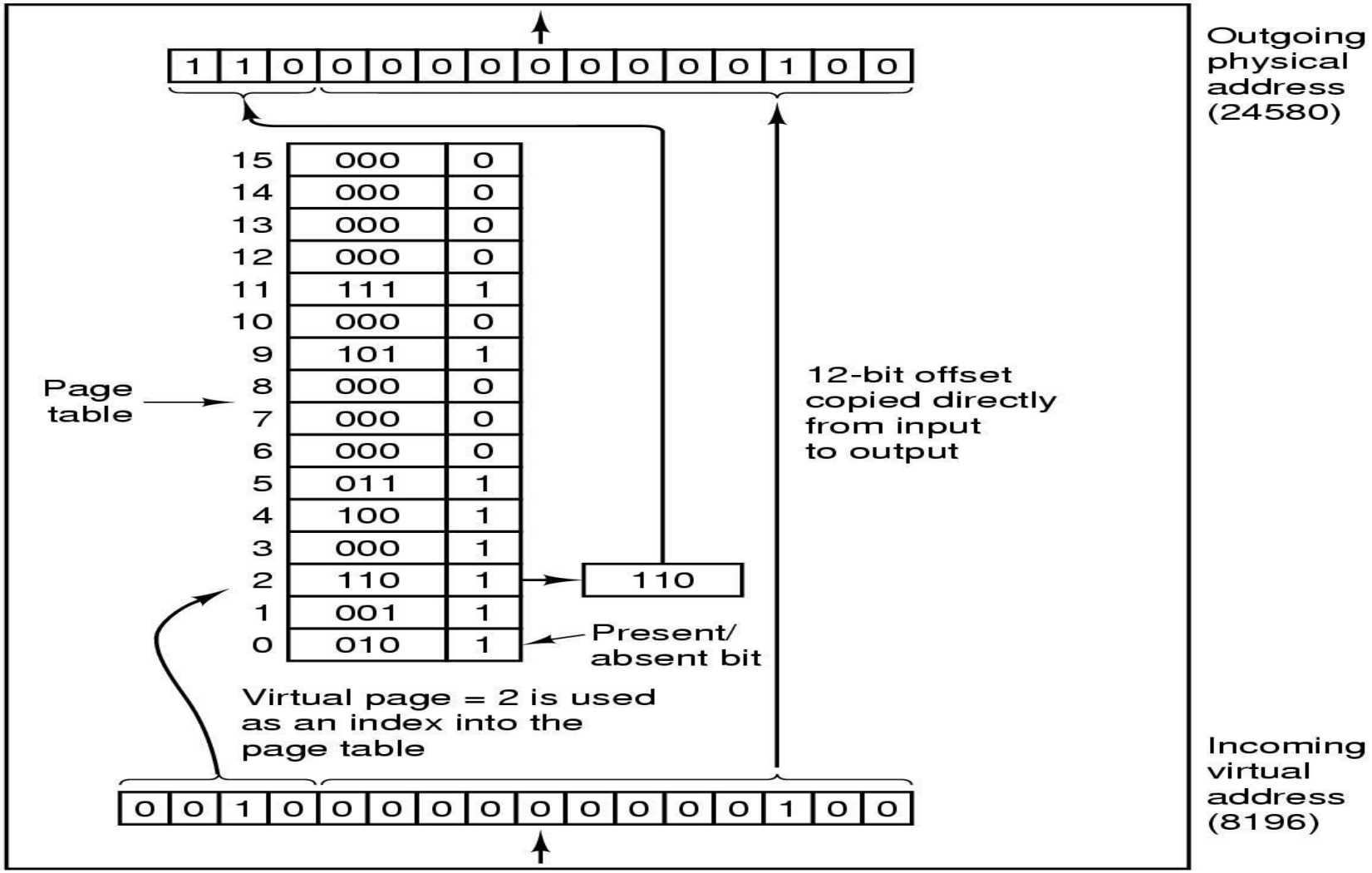
Valid-Invalid Bit

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

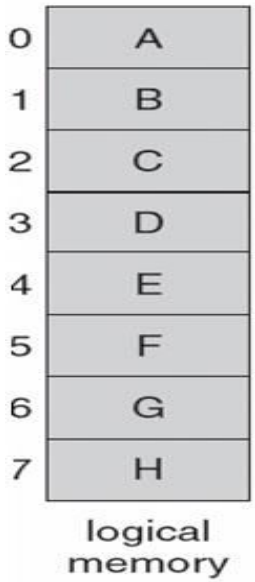


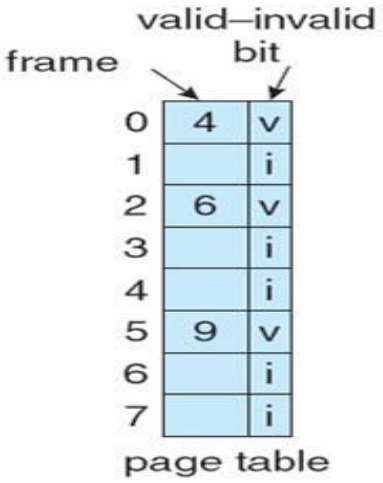
During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault.

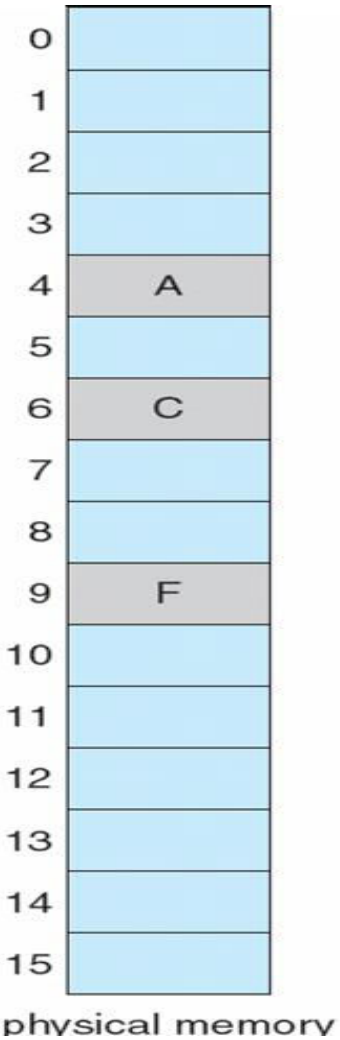
Virtual Memory Mapping Example

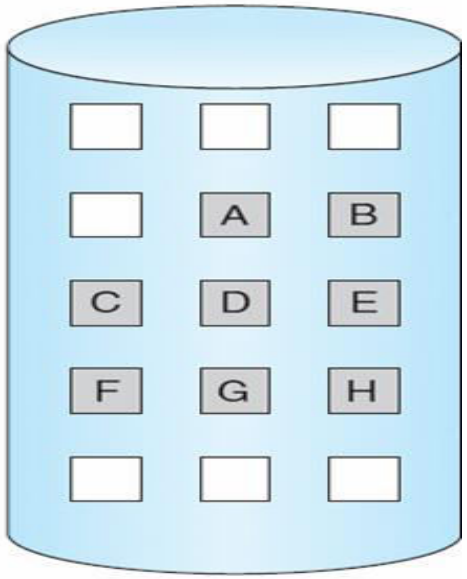


Page Table when some Pages are not in Main Memory



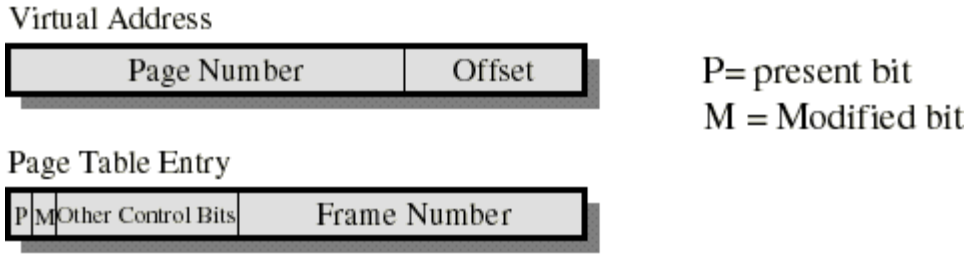






Dynamics of Demand Paging (1)

- Typically, each process has its own page table.

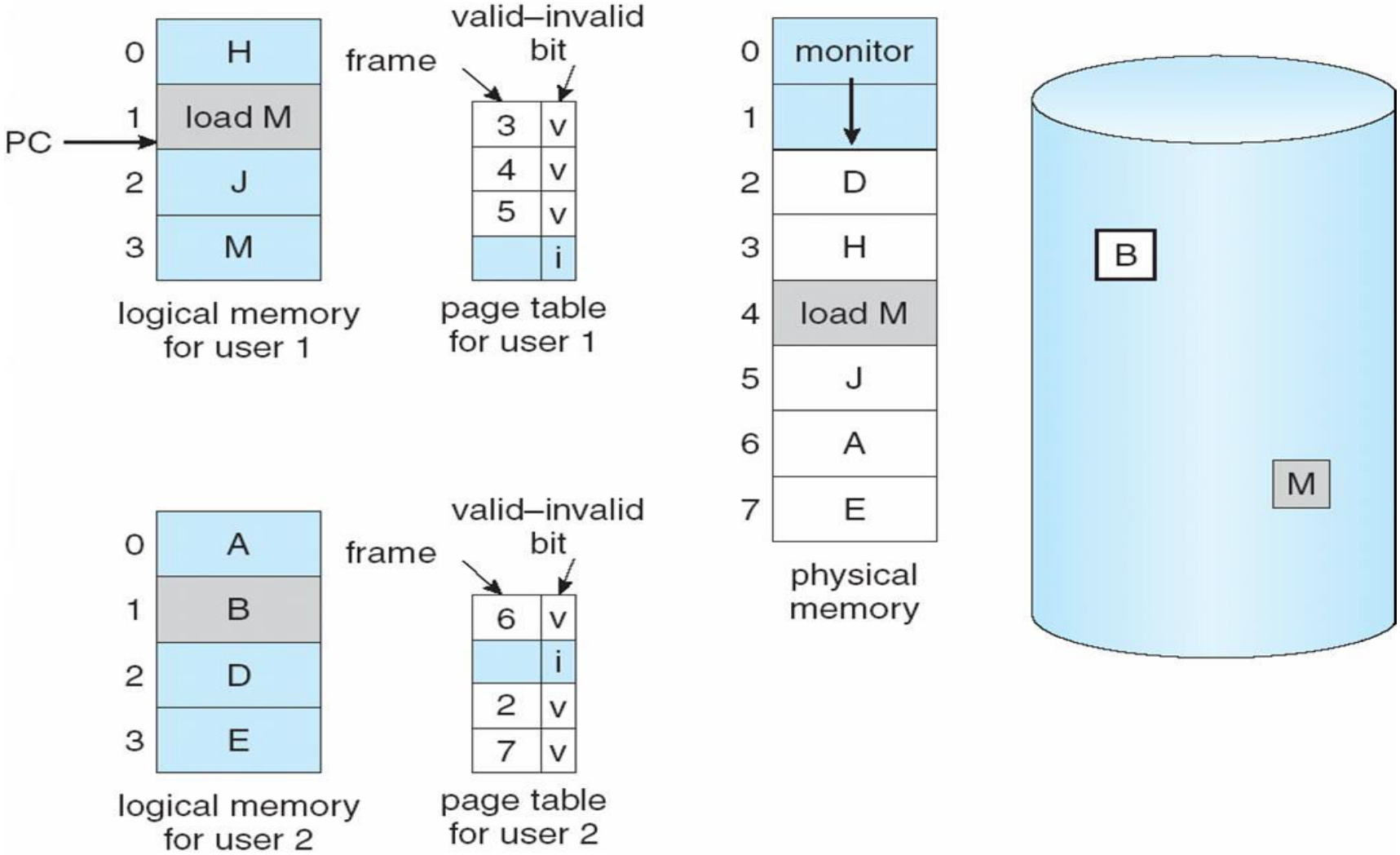


- Each page table entry contains a present (valid-invalid) bit to indicate whether the page is in main memory or not.
  - If it is in main memory, the entry contains the frame number of the corresponding page in main memory.
  - If it is not in main memory, the entry may contain the address of that page on disk or the page number may be used to index another table (often in the PCB) to obtain the address of that page on disk.

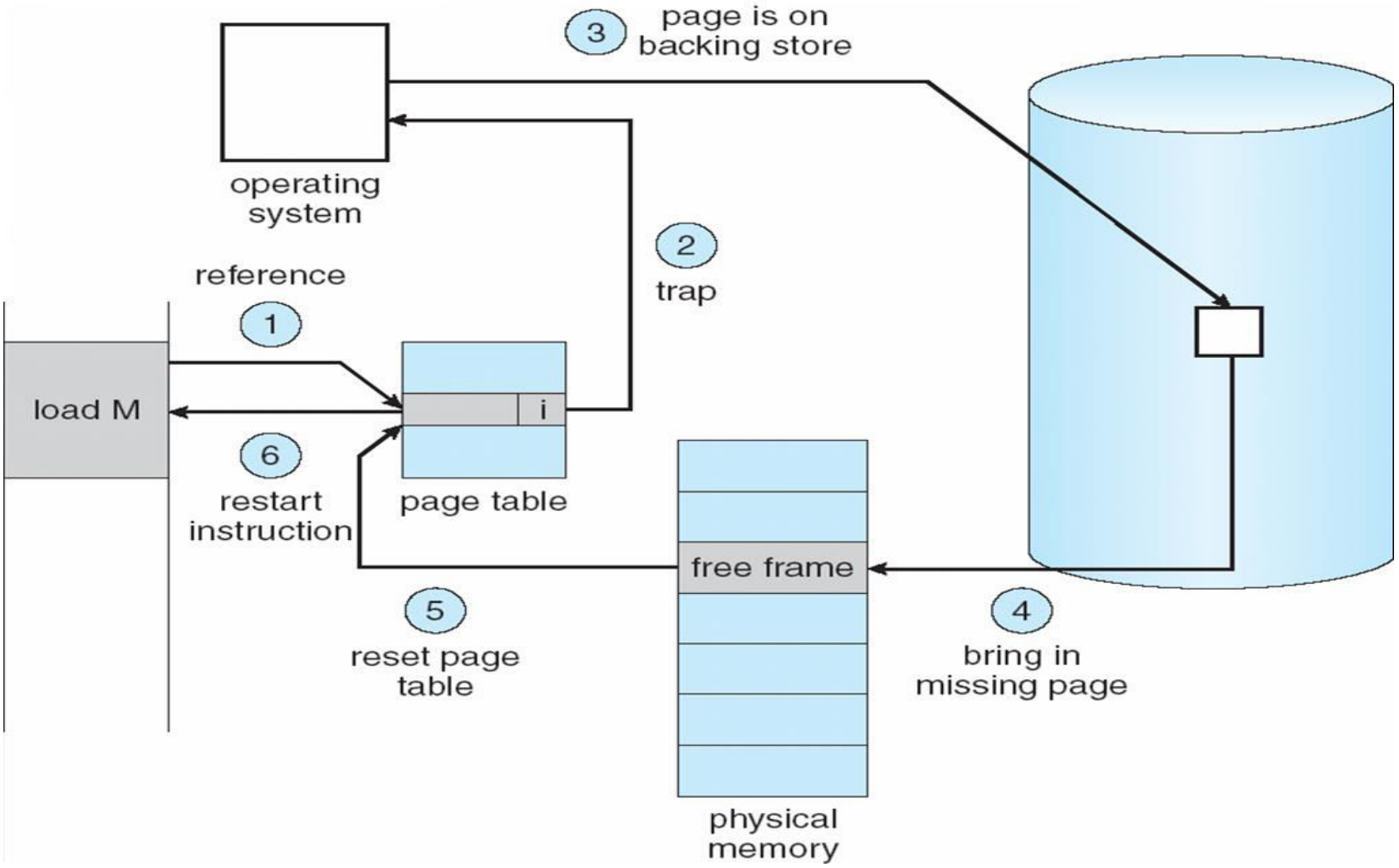
Dynamics of Demand Paging (2)

- A modified bit indicates if the page has been altered since it was last loaded into main memory:
  - If no change has been made, page does not have to be written to the disk when it needs to be swapped out.
- Other control bits may be present if protection is managed at the page level:
  - a read-only/read-write bit.
  - protection level bit: kernel page or user page (more bits are used when the processor supports more than 2 protection levels).

Need For Page Fault/Replacement



Steps in handling a Page Fault (1)



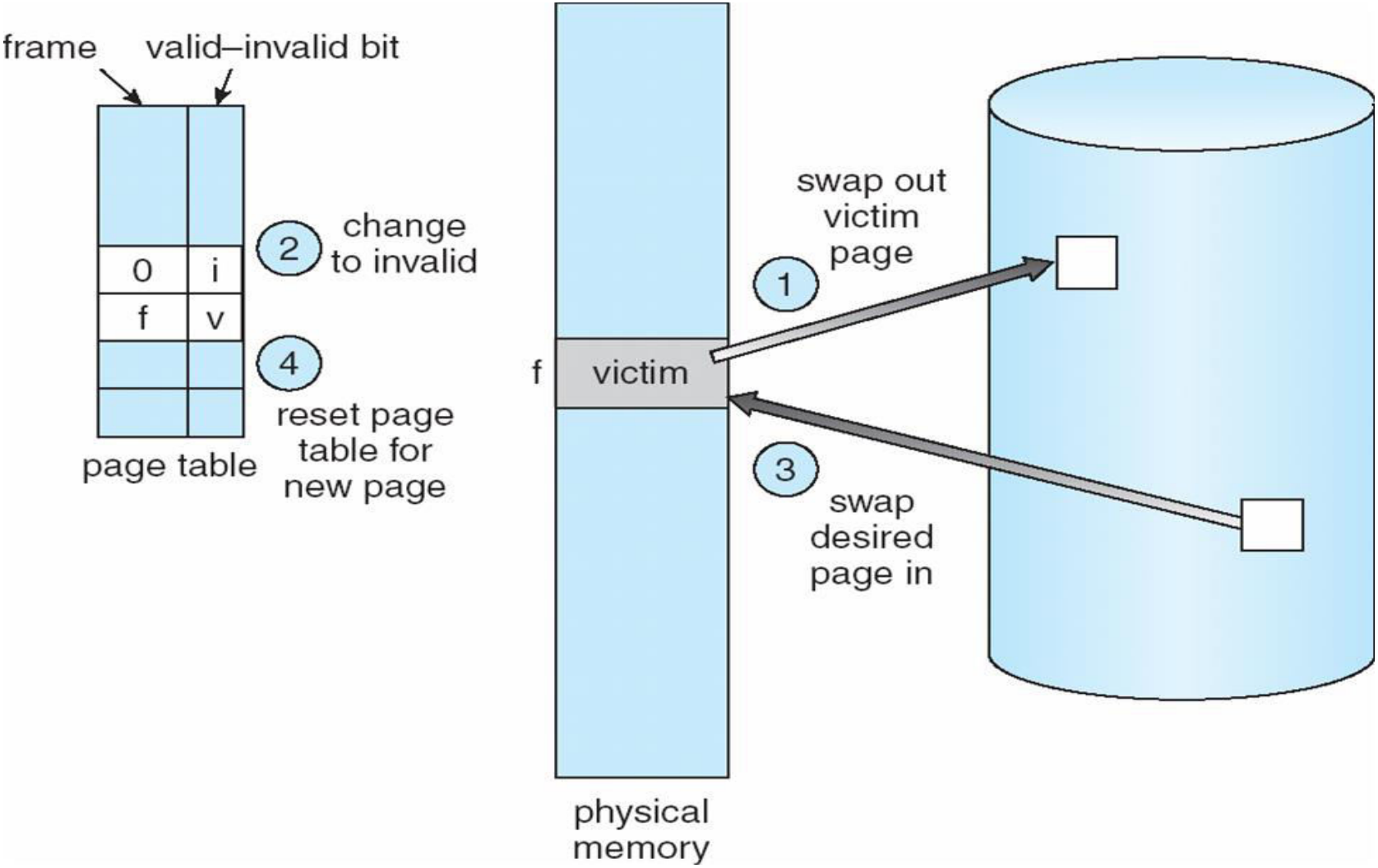
Steps in handling a Page Fault (2)

1. If there is ever a reference to a page not in memory, first reference will cause page fault.
2. Page fault is handled by the appropriate OS service routines.
3. Locate needed page on disk (in file or in backing store).
4. Swap page into free frame (assume available).
5. Reset page tables – valid-invalid bit = v.
6. Restart the instruction that caused the page fault.

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
- Need page replacement algorithm.
- Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Steps in handling a Page Replacement (1)



Steps in handling a Page Replacement (2)

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

Comments on Page Replacement

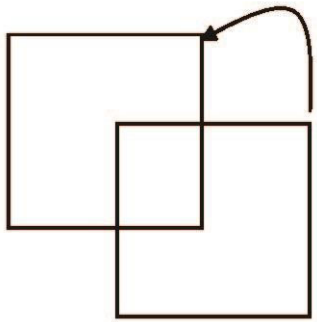
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

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.
  - This is **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.

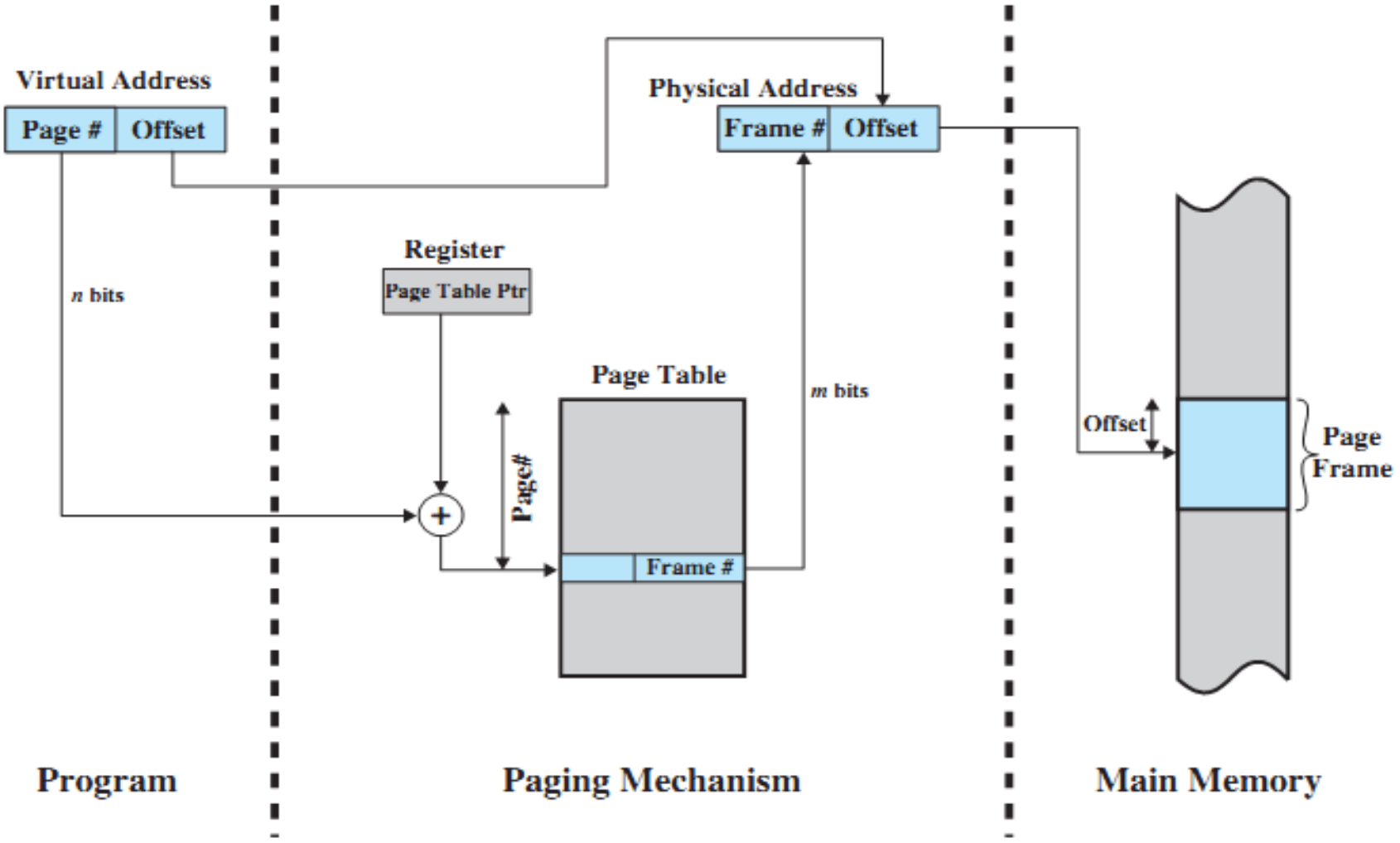
Instruction Restart

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

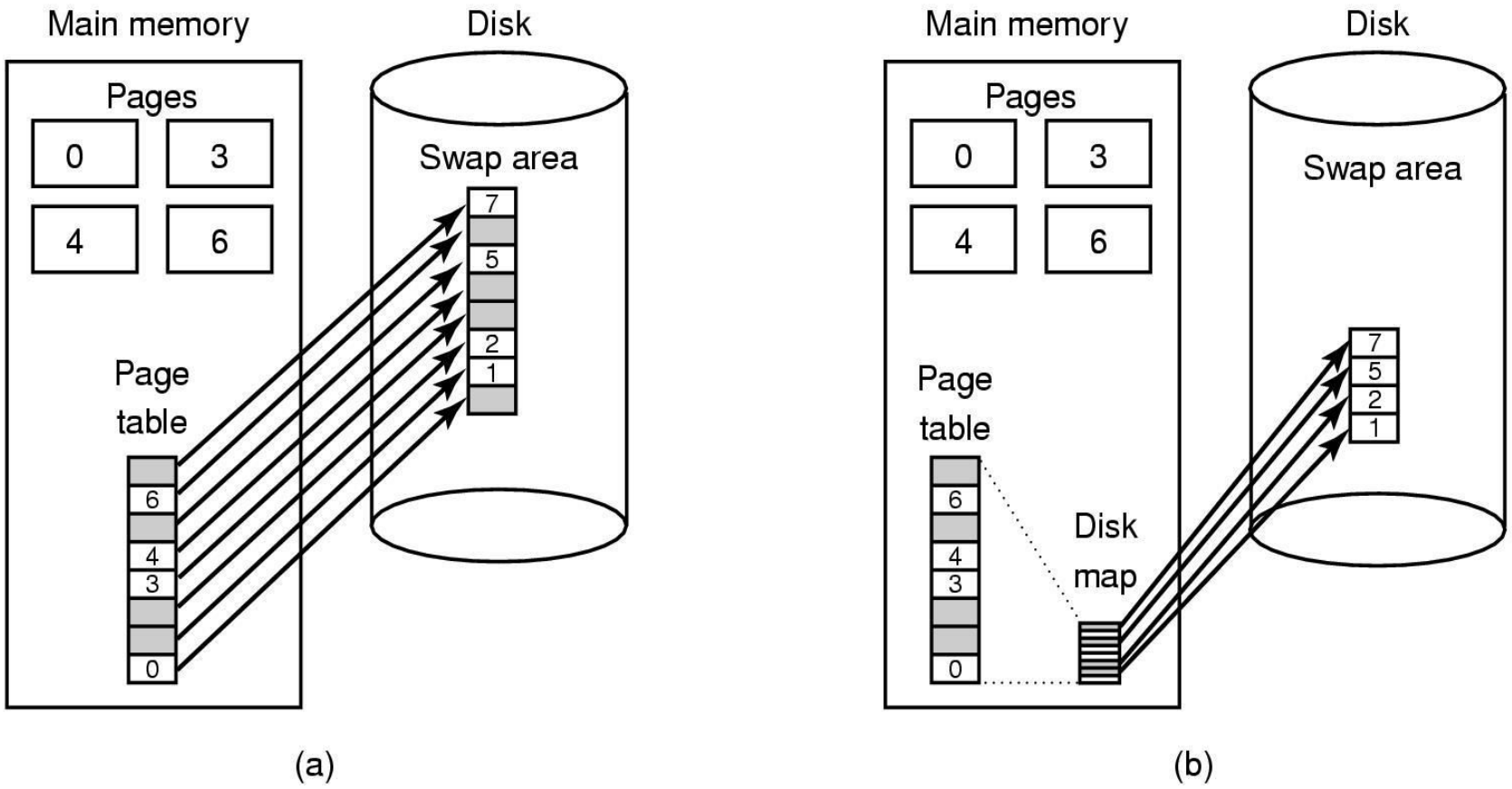


  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Address Translation in a Paging System



Backing/Swap Store



(a) Paging to static swap area. (b) Backing up pages dynamically.

Need for TLB

- Because the page table is in main memory, each virtual memory reference causes at least two physical memory accesses:
  - one to fetch the page table entry.
  - one to fetch the data.
- To overcome this problem a special cache is set up for page table entries, called the TLB (Translation Look-aside Buffer):
  - Contains page table entries that have been most recently used.
  - Works similar to main memory cache.

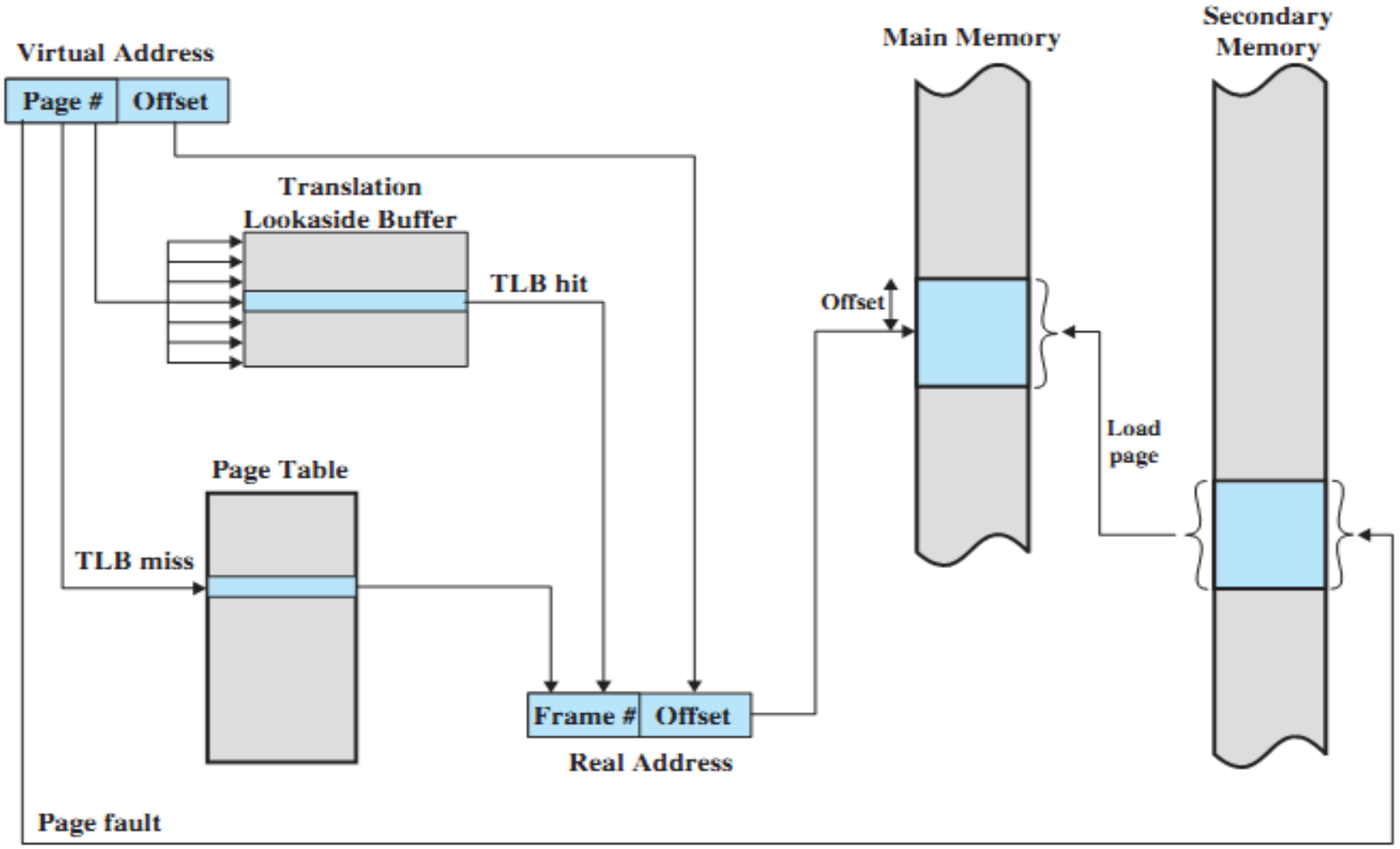
Example TLB

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

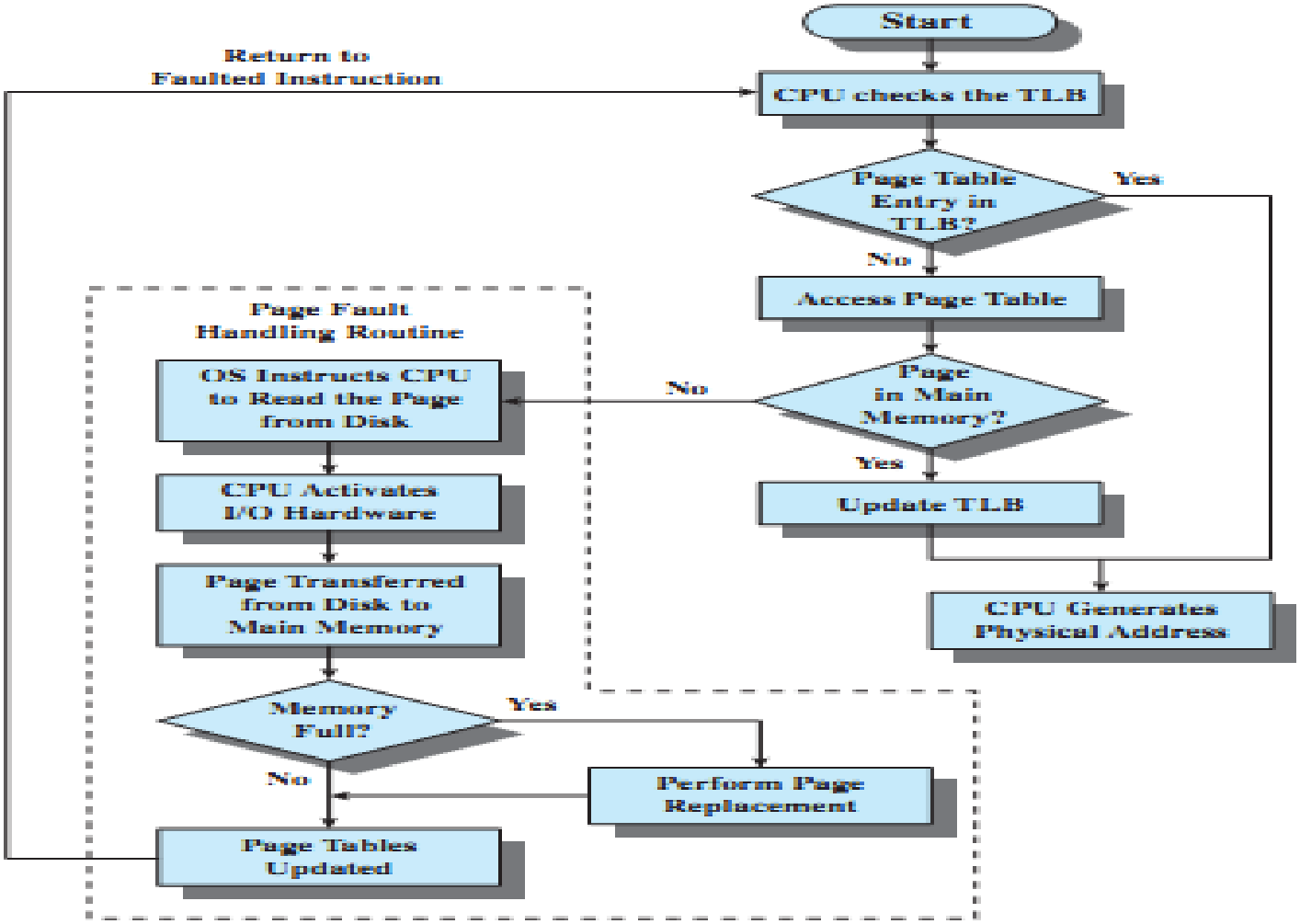
TLB Dynamics

- Given a logical address, the processor examines the TLB.
- If page table entry is present (a hit), the frame number is retrieved and the real (physical) address is formed.
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table:
  - if valid bit is set, then the corresponding frame is accessed.
  - if not, a page fault is issued to bring in the referenced page in main memory.
- The TLB is updated to include the new page entry.

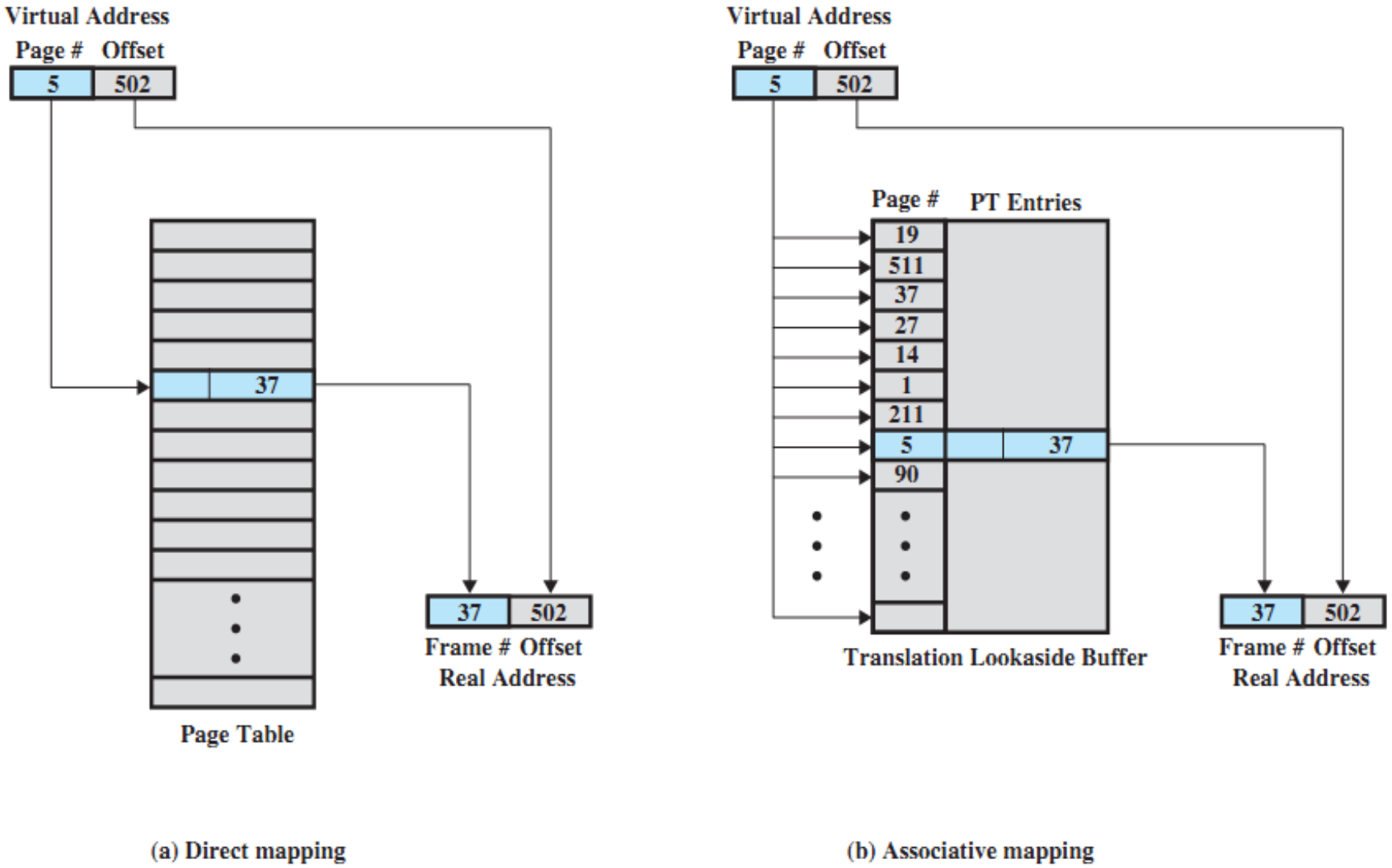
Use of a Translation Look-aside Buffer



Operation of Paging and TLB



Direct vs. Associative Lookup for Page Table Entries



TLB: further comments

- TLB use associative mapping hardware to simultaneously interrogates all TLB entries to find a match on page number.
- The TLB must be flushed each time a new process enters the Running state.
- The CPU uses two levels of cache on each virtual memory reference:
  - first the TLB: to convert the logical address to the physical address.
  - once the physical address is formed, the CPU then looks in the regular cache for the referenced word.

Stages in Demand Paging (worse case)

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine location of page on the disk.
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced.
    2. Wait for the device seek and/or latency time.
    3. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user.
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Performance of Demand Paging

- 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) –

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

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

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p

    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses.