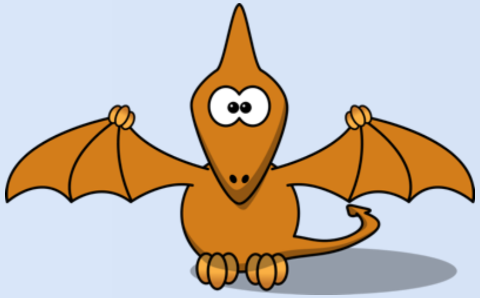
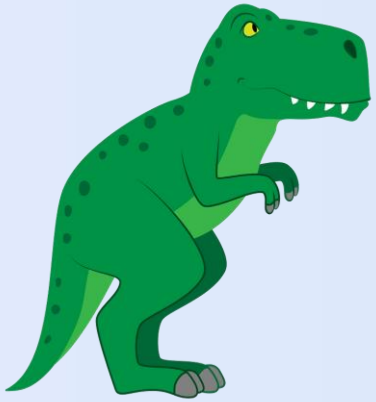


Chapter 10.

Virtual Memory



Operating System Concepts (10th Ed.)

10.1 Background

- ***Virtual Memory***
  - a technique to allow the execution of processes
    - that are not completely in memory
    - so, programs can be larger than physical memory.
  - abstracts main memory into an extremely large array of storage,
    - separating logical memory from physical memory.
  - provides an efficient mechanism
    - for sharing files and libraries and process creation.

10.1 Background

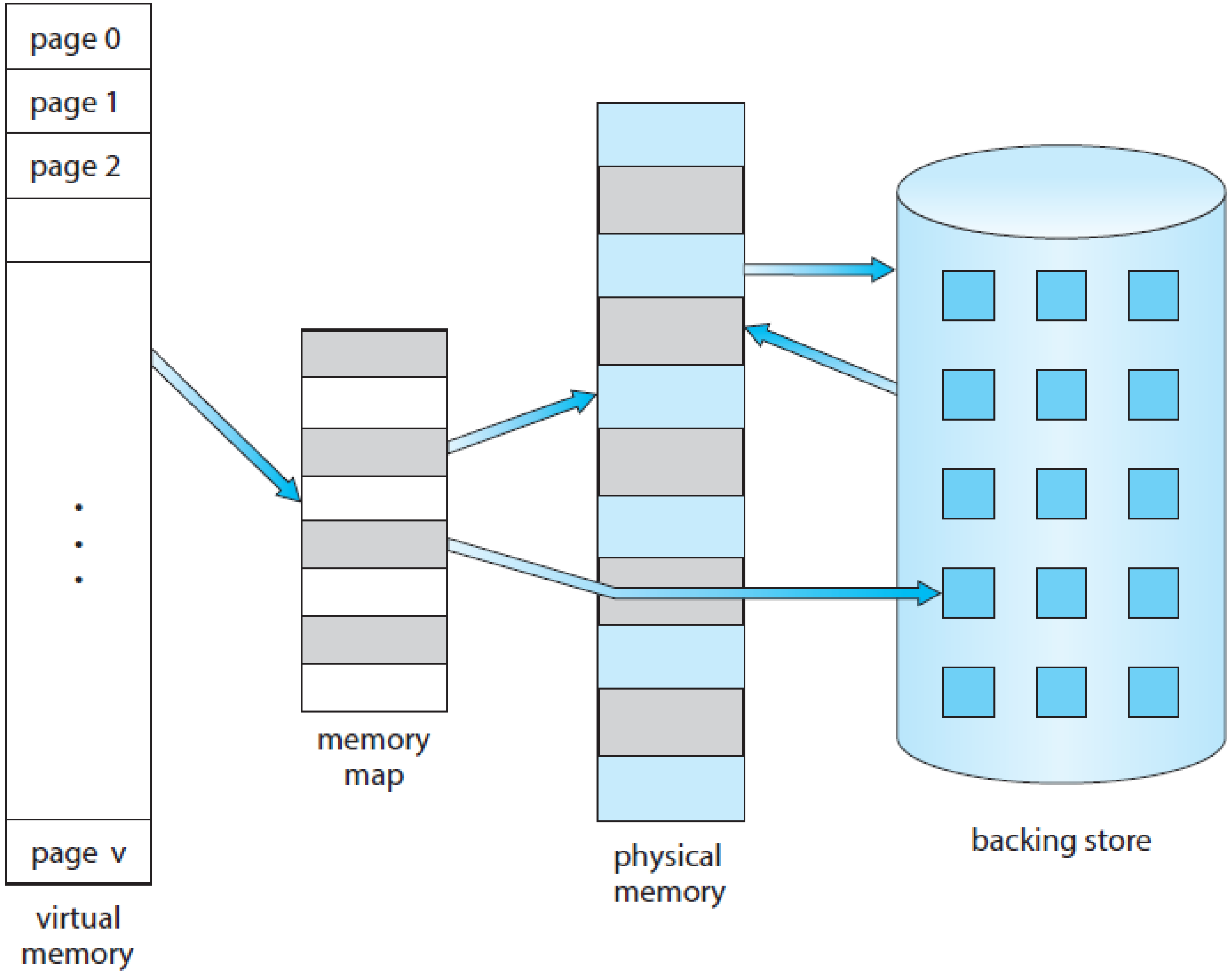


Figure 10.1 *Diagram showing virtual memory that is larger than physical memory.*

10.1 Background

- *Virtual Address Space*
  - the logical (or virtual) view of how a process is stored in memory.
  - Typically, begins a certain logical address, to say, address 0,
  - and exists in contiguous memory.

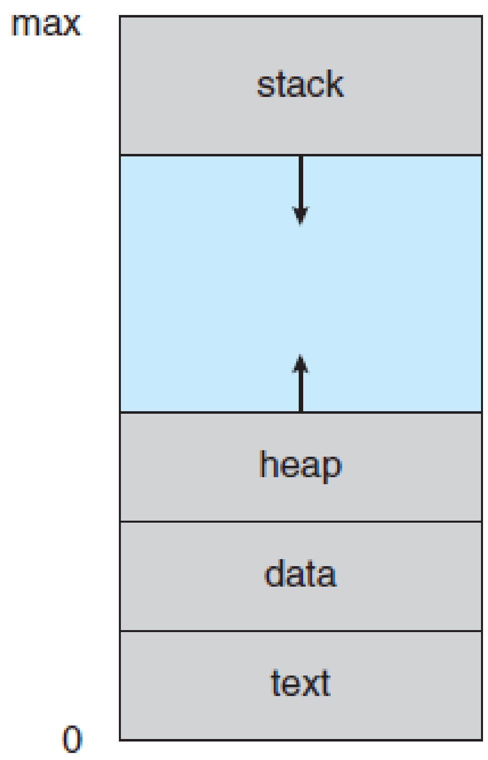


Figure 10.2 *Virtual address space of a process in memory.*

10.1 Background

- Virtual Memory
  - allows *files* and *memory* to be shared by two or more processes
    - through *page sharing*.

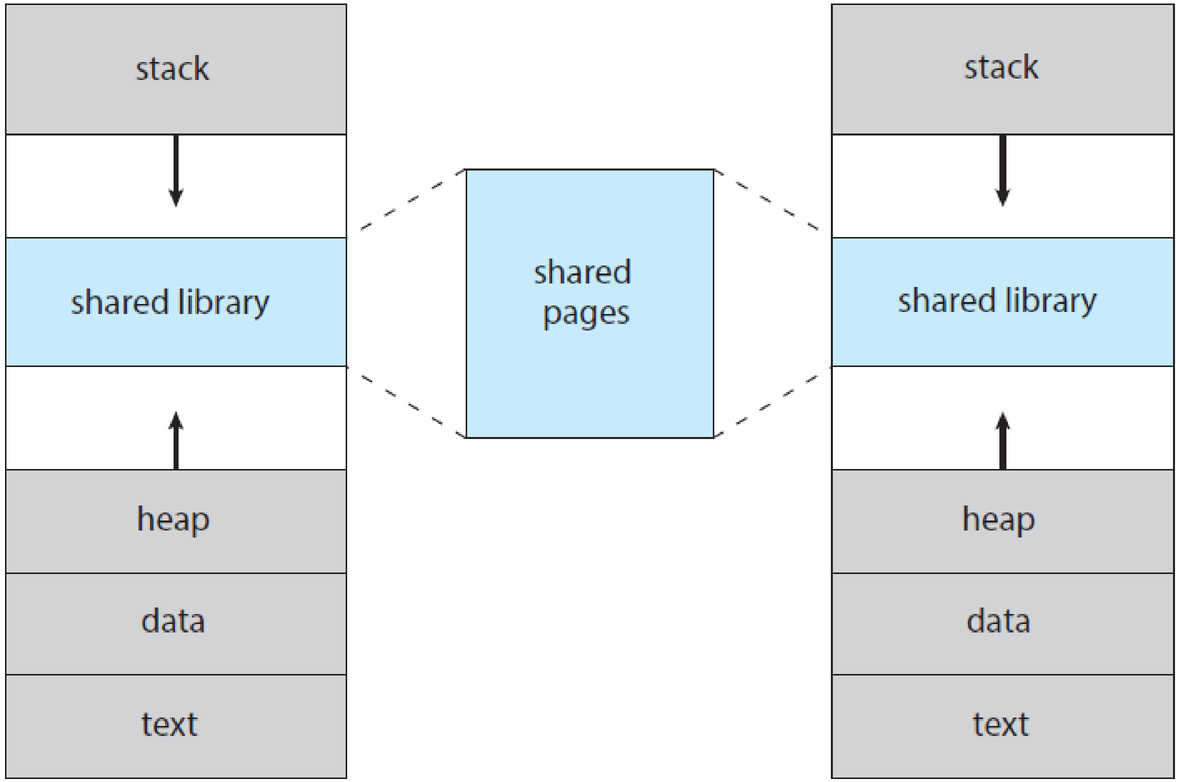


Figure 10.3 *Shared library using virtual memory.*

10.2 Demand Paging

- Consider how an executable program
  - might be loaded from secondary storage into memory.
  - One option is to load the entire program in physical memory
  - The ***demand paging*** is an alternative strategy
    - to load pages only as they are needed.
    - commonly used in virtual memory systems.
  - With *demand-paged virtual memory*,
    - pages are loaded only when they are *demanded* during execution.

10.2 Demand Paging

- Basic Concepts of **the Demand Paging**:
  - While a process is executing,
    - some pages will be *in memory* and some will be *in secondary storage*.
  - To *distinguish* between these two situations,
    - the *valid-invalid bit* scheme can be used.
    - *valid*: the page is both legal and in memory.
    - *invalid*: the either is not valid or currently in secondary storage.

10.2 Demand Paging

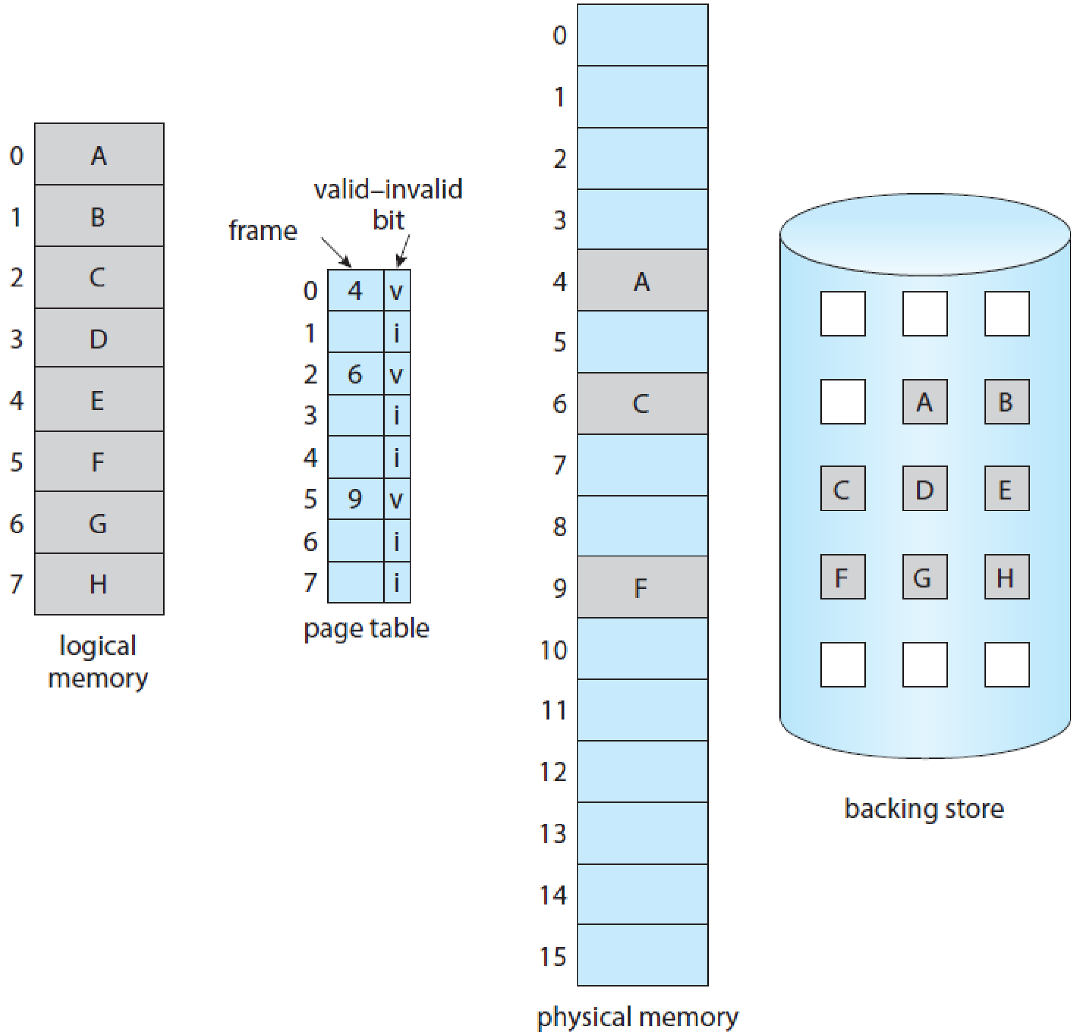


Figure 10.4 *Page table when some pages are not in main memory.*

10.2 Demand Paging

- The procedure for handling the ***Page Fault***:
  1. *Check an internal table* for the process to determine
     - whether the reference was *valid* or *invalid* memory access.
  2. If the reference was valid, terminate the process,
     - or valid but page fault, we now *page it in*.
  3. Find a *free frame* (by taking one from the free-frame list)
  4. Schedule a secondary storage operation
     - to *read the desired page* into the newly allocated frame.
  5. When the storage read is complete, *modify the internal table and*
     - *the page table* to indicate that the page is now in memory.
  6. *Restart the instruction* that was interrupted by the trap.

10.2 Demand Paging

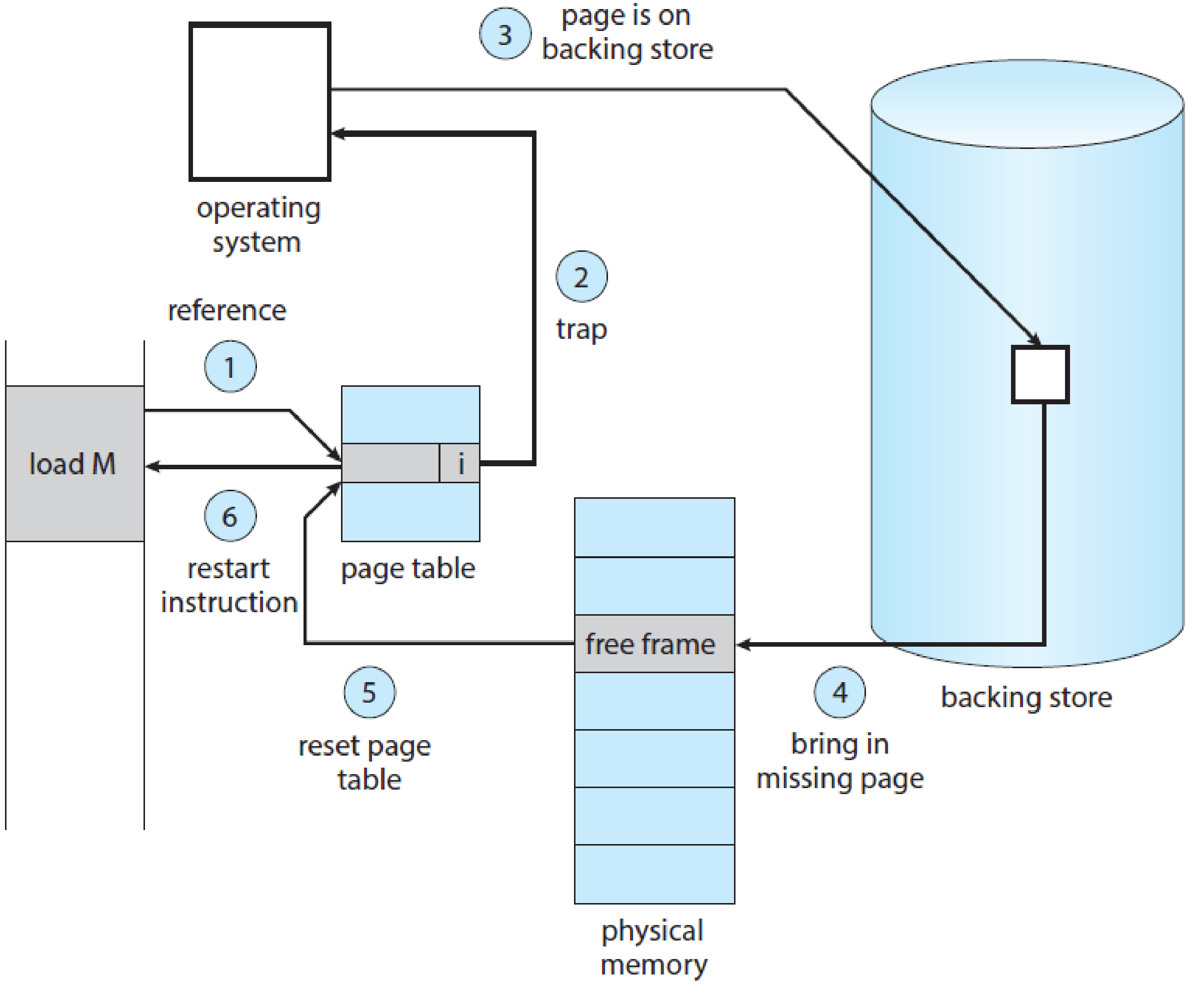


Figure 10.5 *Steps in handling a page fault.*

10.2 Demand Paging

- ***Pure*** Demand Paging:
  - *never bring a page* into memory *until it is required*.
  - With the scheme of pure demand paging,
    - we can start executing a process with *no pages* in memory.
  - When the OS sets the instruction pointer
    - to the *first instruction* of the process with a *page fault*,
    - the page of the process would be paged in.

10.2 Demand Paging

- ***Locality of Reference***:
  - If a program accesses *several new pages* with *each instruction*,
    - to say, *one page* for *the instruction* and *many pages* for *data*,
    - possibly causes *multiple page faults per instruction*.
  - Fortunately, analysis running processes
    - show that this behavior is exceedingly unlikely.
  - Programs tend to have *the locality of reference*,
    - which results in *reasonable performance* from *demand paging*.

An Example of Program Structure ($10.9.5)

- Let the *page size* to be 128 and an *array* to be 128 × 128,
  - then compare the following two codes.

```
int i, j;
int[128][128] data;

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

```
int i, j;
int[128][128] data;

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

- Careful selection of data structures and programming structures
  - can increase *the locality of code or data*
  - hence, *lower the page-fault* rate and *enhance the system performance*.

10.2 Demand Paging

- Hardware Support to Demand Paging:
  - *Page table*:
    - has the ability to mark valid or invalid. (with a valid-invalid bit)
  - *Secondary memory*: (=***swap space***)
    - holds those pages that are not present in main memory.
    - usually a high-speed disk or NVM device

10.2 Demand Paging

- Instruction Restart:
  - A crucial requirement for demand paging.
    - the ability to restart any instruction after page fault.
  - When the page fault occurs,
    - the state of interrupted process (registers, condition code, instruction counter, etc.) is saved.
  - Therefore, *restart* the process in *exactly the same place and state*.
  - If a *page fault* occurs *on the instruction fetch*,
    - restart by fetching the instruction again.
  - If a *page fault* occurs while we are *fetching an operand*,
    - fetch and decode the instruction again and then fetch the operand.

10.2 Demand Paging

- As a worst-case example:
  - `ADD  A, B, C`; three address instruction, adding A and B into C.
    1. Fetch and decode the instruction (ADD)
    2. Fetch A
    3. Fetch B
    4. ADD A and B
    5. Store the sum in C

10.2 Demand Paging

- **Free Frame List**:
  - When a page fault occurs,
    - OS must bring the desired page from secondary storage into memory.
  - To resolve page faults, OS maintains the *free frame list*:
    - a pool of free frames for satisfying such requests.
  - Free frames must also be allocated
    - when the stack or heap segments from a process expand.

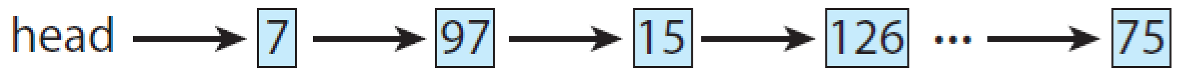


Figure 10.6 *List of free frames.*

10.2 Demand Paging

- Performance of Demand Paging:
  - How to compute the *effective access time* for a *demand-paged* memory?
  - Let *ma* to denote the *memory-access time*.
  - Let *p* be the *probability* of a *page fault*.
    - EAT $= (1 - p) \times ma + p * (page\ fault\ time)$.
  - How much time is needed to service a page fault?
    - three major activities:
    - Service the page-fault interrupt.
    - Read in the page.
    - Restart the process.

10.2 Demand Paging

- Consider a system:
  - with an average page-fault service time of 8 *milliseconds*
    - and a memory access time of 200 *nanoseconds*,
    - $\text{EAT} = (1 - p) \times 200 + p \times 8{,}000{,}000$
      $= 200 + 7{,}999{,}800 \times p$
  - If one access out of 1,000 causes a page fault ($p = 0.001$),
    - $\text{EAT} = 200 + 7999.8 = 8199.8$ *nanoseconds*
      $\cong 8.2$ *microseconds*

10.3 Copy-on-Write

- **Copy-on-Write**:
  - *copy* a shared page *only when* a process *writes* to a shared page.
  - Recall the process creation with **fork()** and **exec()**.

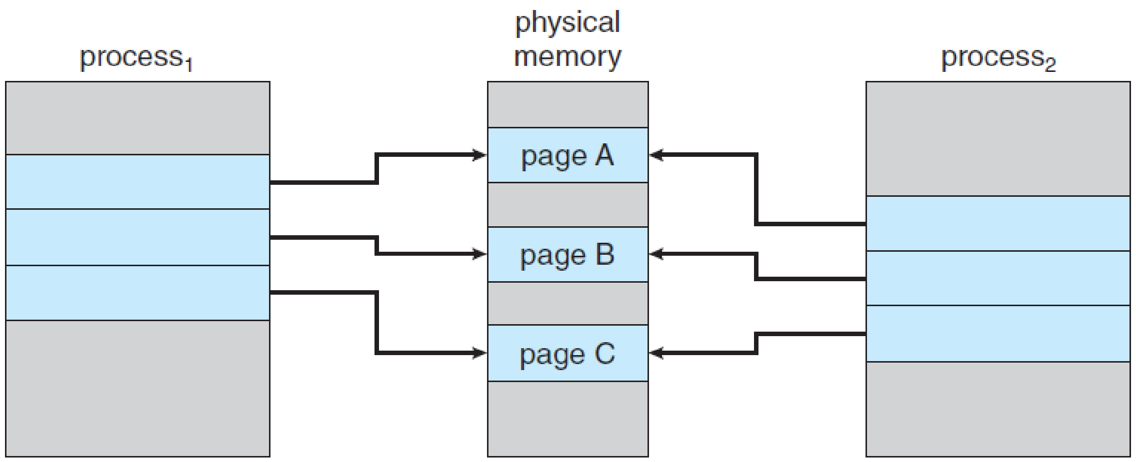


Figure 10.7 *Before process 1 modifies page C.*

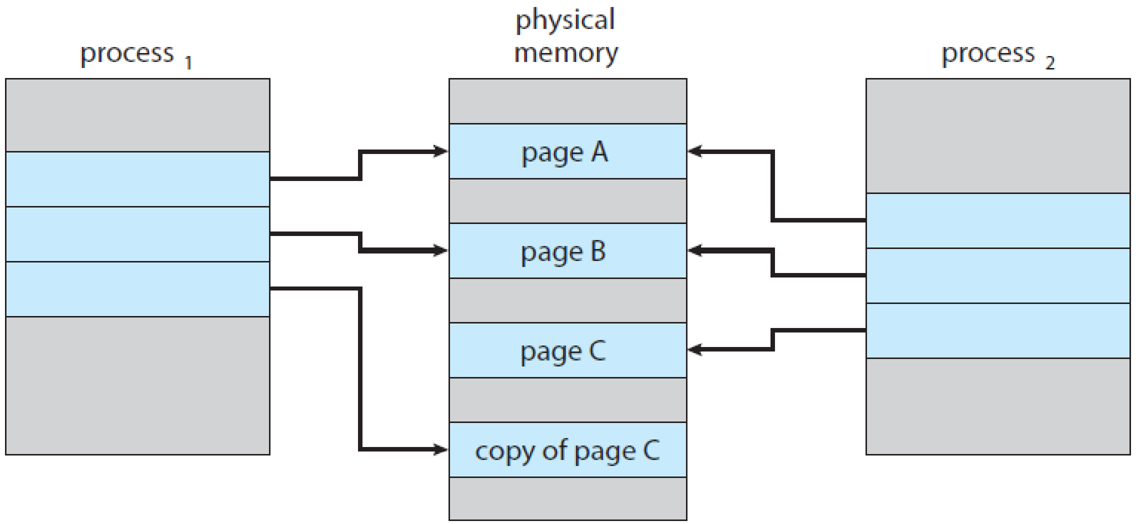


Figure 10.8 *After process 1 modifies page C.*

10.4 Page Replacement

- What happens if there is *no free frames*?
  - If we increase our degree of multiprogramming,
    - we are over-allocating memory.
  - If we have 40 frames and run 6 processes,
    - each of which is 10 pages in size, but actually uses only 5 pages.
    - Then, we manage to demand-paged system with 10 frames spared.
  - However, what if the processes *suddenly want to use all 10 pages*,
    - or need a huge buffer *consuming more pages than available* ones?

10.4 Page Replacement

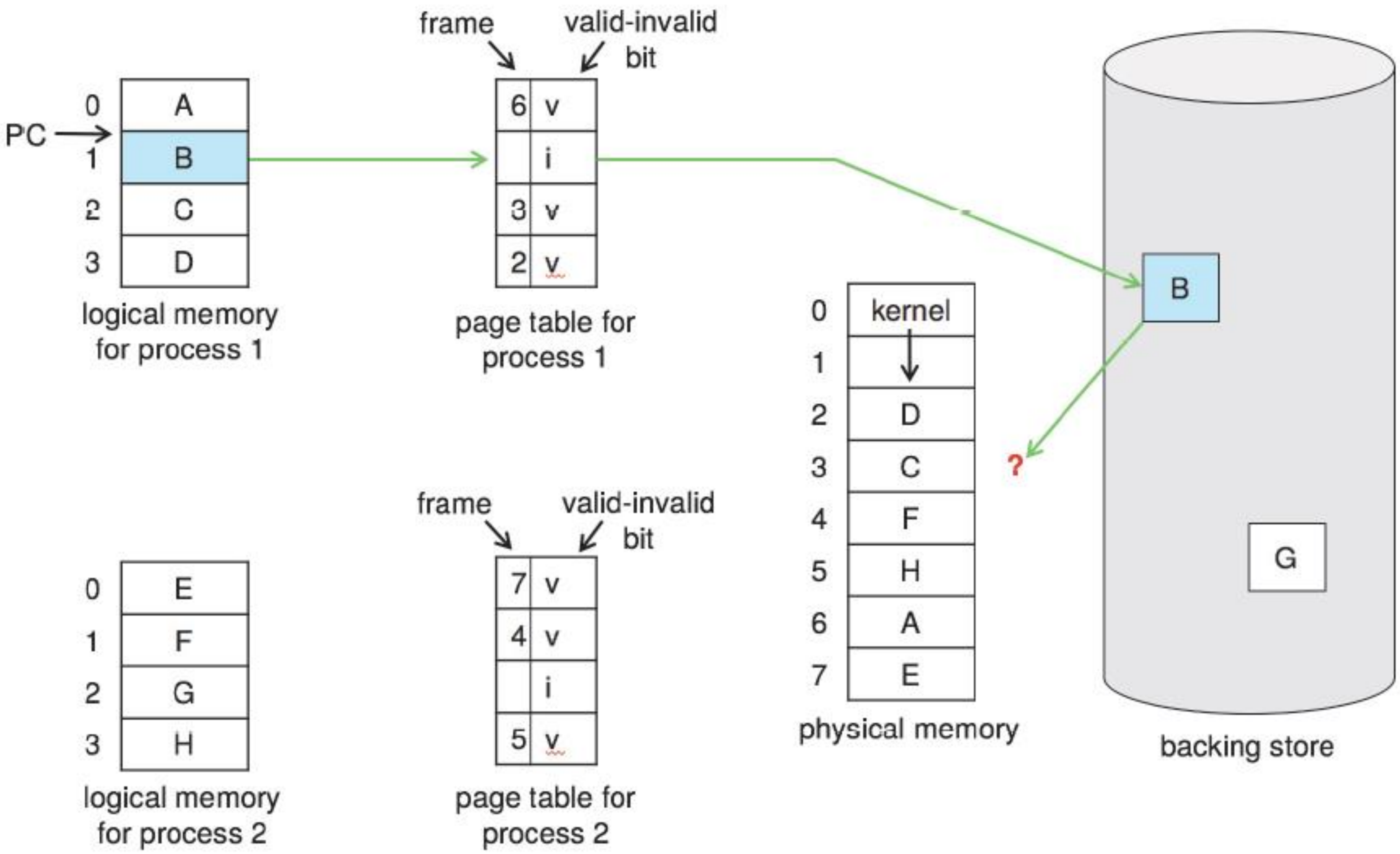


Figure 10.9 *Need for page replacement.*

10.4 Page Replacement

- **Page Replacement**:
  - If no frame is free,
    - find one that is not currently being used and free it.
  - Free a frame by writing its contents to swap space
    - and changing the page table
    - to indicate that the page is no longer in memory (invalid or dirty).
  - Now, use the freed frame to hold page
    - for which the process faulted.

10.4 Page Replacement

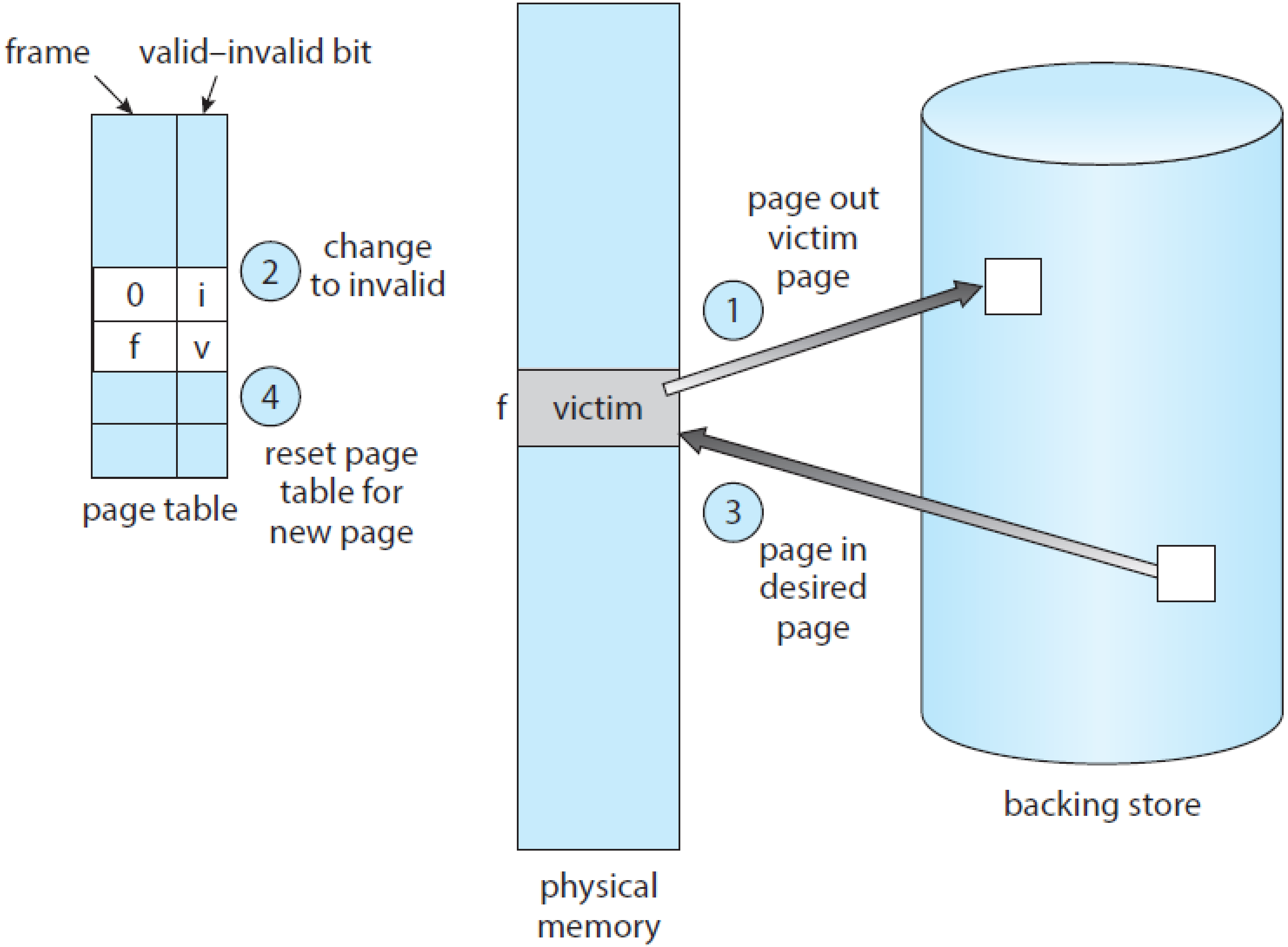


Figure 10.10 *Page replacement.*

10.4 Page Replacement

- Page Fault Service Routine incl. Page Replacement:
  1. Find the location of the desired page on secondary storage.
  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 the victim frame to secondary storage; change the page and frame tables accordingly.
  - Read the desired page into the *newly freed frame*
     - change the page and frame tables.
  - Continue the process from where the *page fault occurred*.

10.4 Page Replacement

- Two major problems to implement demand paging:
  - ***Frame-allocation algorithm***:
    - how many frames to allocated to each process?
  - ***Page-replacement algorithm***:
    - select the frames that are to be replaced.
  - Since the secondary storage I/O is *so expensive*,
    - even *slight improvements* in demand-paging methods
    - can yield *large gains* in system performance.

10.4 Page Replacement

- Evaluation of Page Replacement Algorithms:
  - reference string: a string of memory references.
  - Evaluate an algorithm by running it on a reference string
    - and computes *the number of page faults*. (***minimize it!***)
  - What about the number of page frames?
    - Obviously, the more frames, the less page faults.

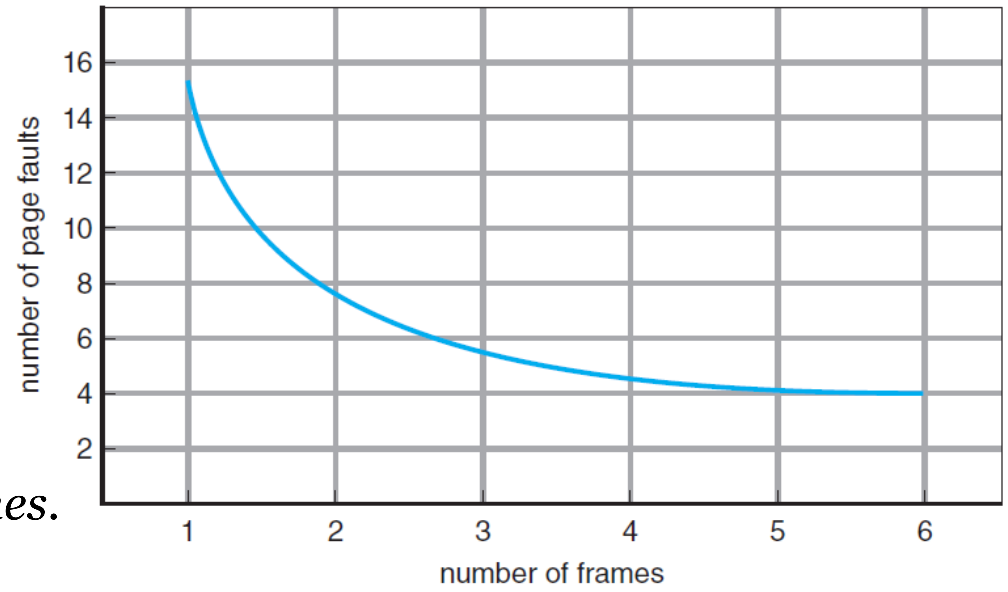
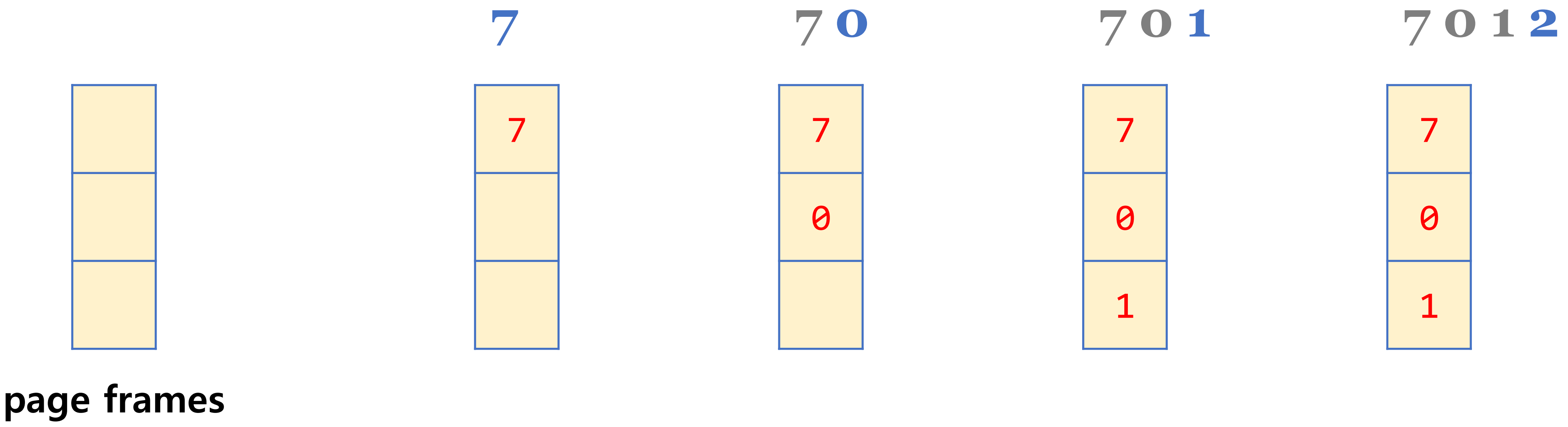


Figure 10.11 *Graph of page faults versus number of frames.*

10.4 Page Replacement

- An example to evaluate algorithms:
  - reference string: 7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1
    - in a memory with ***three*** frames.



10.4 Page Replacement

- **FIFO** Page Replacement:
  - FIFO: First-In-First-Out: the simplest algorithm.
  - Choose *the oldest page* when a page must be replaced.
  - There are *15 page faults* with our example.

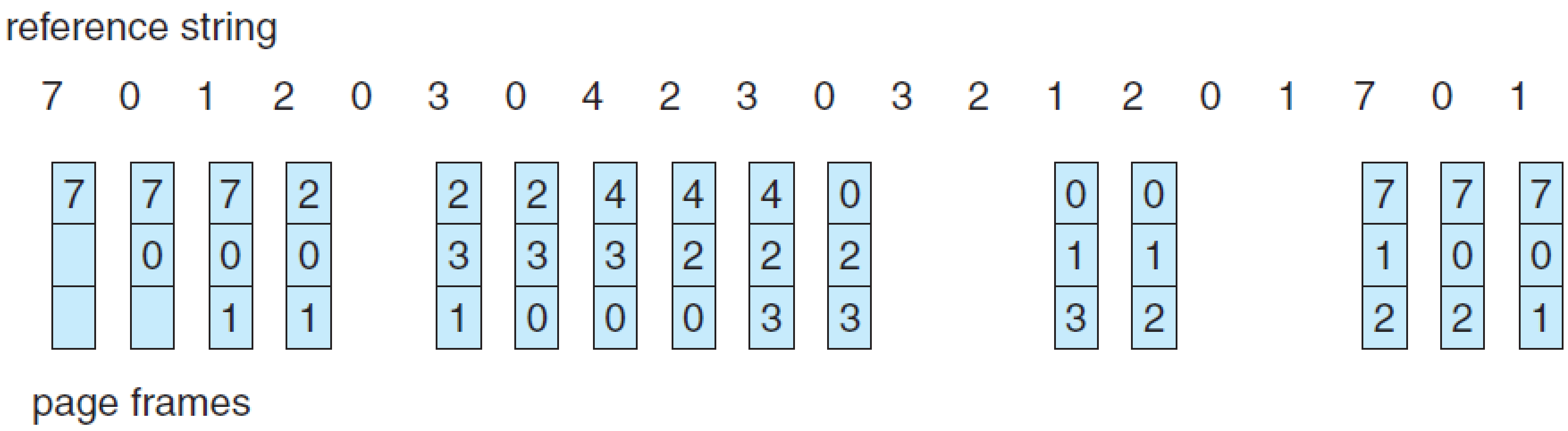
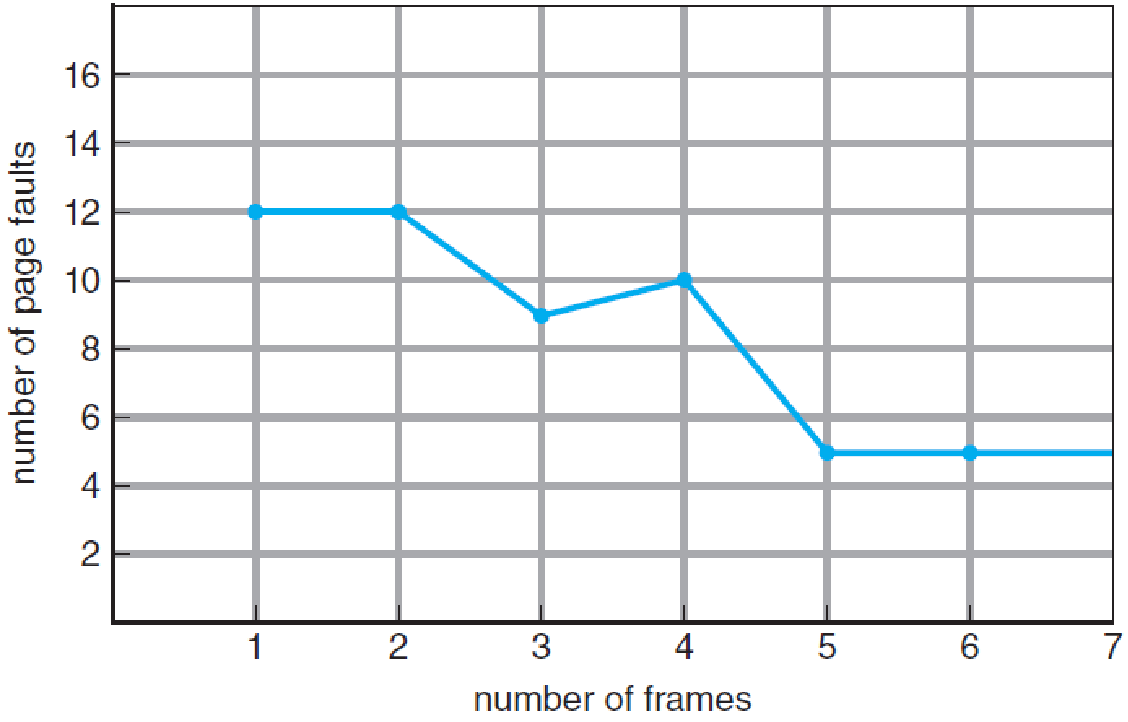


Figure 10.12 *FIFO page-replacement algorithm.*

10.4 Page Replacement

- ***Belady's Anomaly***:
  - The page-fault rate *may increase*
    - as *the number of allocated frames increases*.



- **reference string**:

  1 2 3 4 1 2 5 1 2 3 4 5

Figure 10.13 *Page-fault curve of FIFO replacement on a reference string.*

10.4 Page Replacement

- **Optimal** Page Replacement:
  - Seeking for an *optimal* algorithm that has the *lowest page-fault rate*,
    - and *never suffers from* Belady's anomaly.
  - OPT or MIN:
    - replace the page that *will not be used* for the *longest period of time*.
  - OPT will guarantee the lowest possible page-fault rate.

- **The difficulty of implementing OPT:**
  - There are *9 page faults* with our example.
  - OPT requires *future knowledge* of the reference string.
    - used mainly for comparison studies.

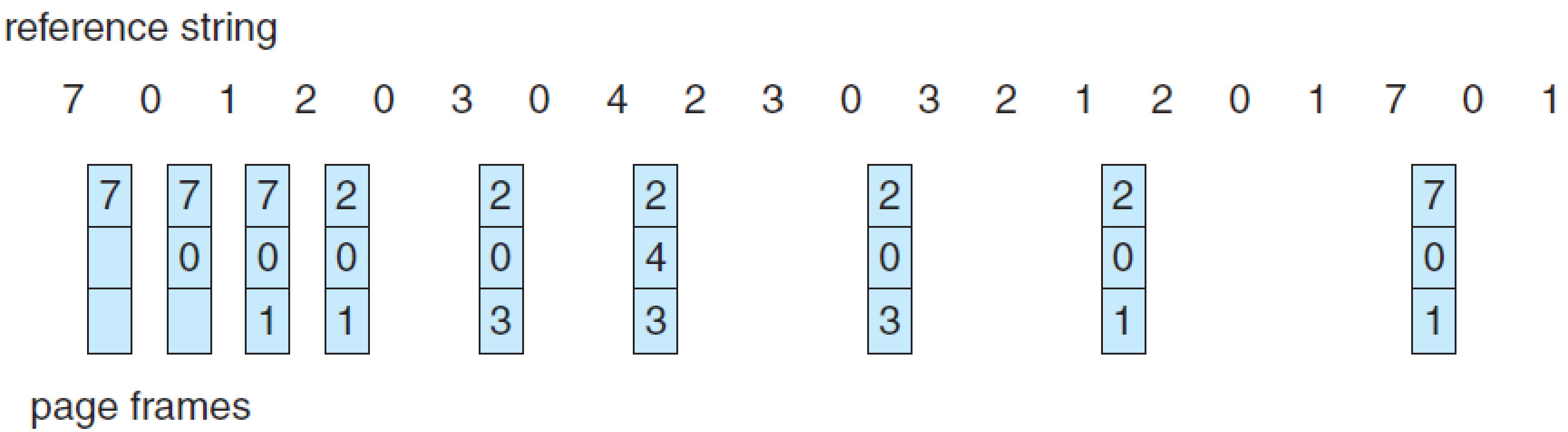


Figure 10.14 *Optimal page-replacement algorithm.*

10.4 Page Replacement

- Recall the Shortest-Job-First CPU scheduler.
  - The key distinction between the FIFO and the OPT:
    - *looking backward*: when a page was *brought in*?
    - *looking forward*: when a page to *be used*?
  - If we use the *recent past*
    - as an *approximation* of the *near future*,
  - then we can replace the page
    - that has *not been used* for the *longest period of time*.

10.4 Page Replacement

- **LRU** Page Replacement:
  - LRU: Least Recently Used
  - Associates with each page *the time of that page's last use*, and
    - choose the page that has *not been used* for the *longest period* of time.
    - There are *12 page faults* with our example.

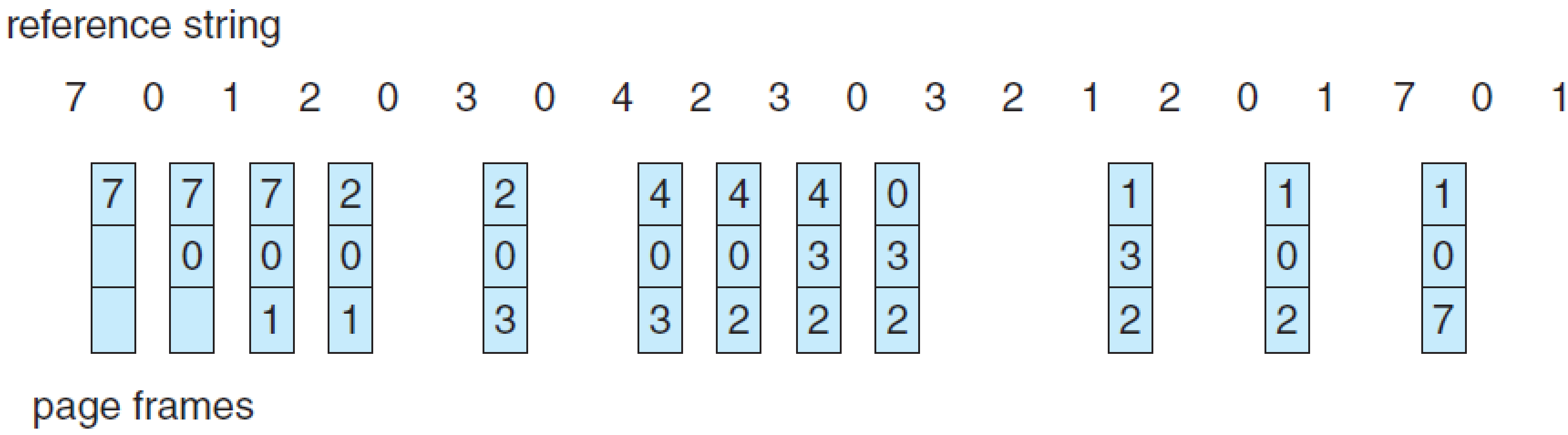


Figure 10.15 *LRU page-replacement algorithm.*

- LRU policy
  - is considered to be *good* and is *often used*.
  - However, the problem to solve for the implementation of LRU is
    - to determine an order for the frames defined by the time of last use?
  - It may require substantial *hardware assistance*.
    - Two implementations are possible: *counter* and *stack*
  - LRU *does not suffer* from Belady's anomaly like OPT.

10.4 Page Replacement

- Two implementation methods for the LRU:
  - *Counter* implementation:
    - Whenever a page is referenced, copy the counter (or the clock).
    - Replace the page with the smallest value.
  - *Stack* implementation:
    - Keep a stack of page numbers.
    - Note that entries must be removed from the middle of the stack.

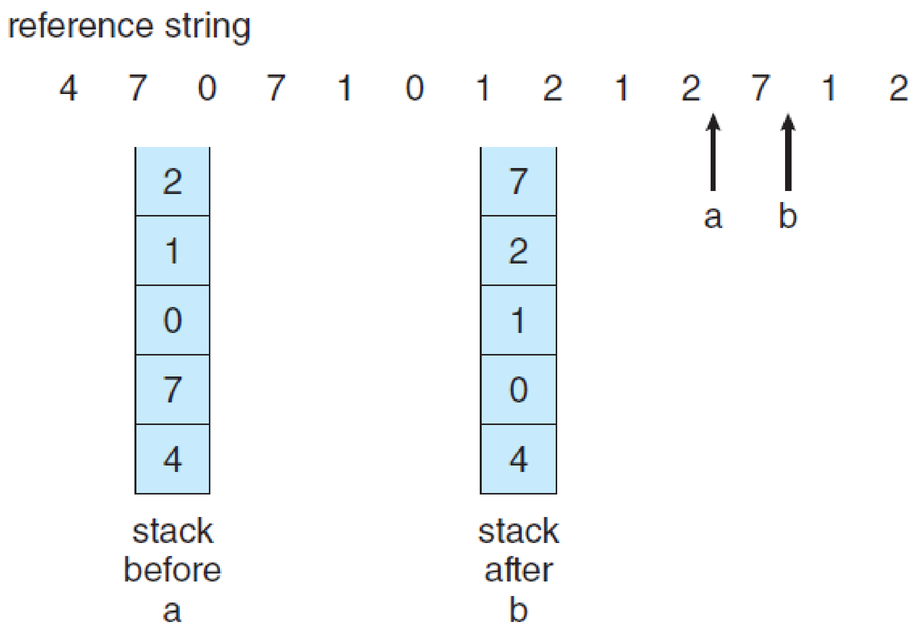


Figure 10.16 *Use of a stack to record the most recent page reference.*

- LRU-Approximation
  - LRU needs hardware support,
    - however, many systems provide some help with a *reference bit*.
  - ***reference bit:***
    - initially 0, a bit associated with each page.
    - when a page is referenced, set to 1.
    - replace any with reference bit = 0 (if any).

- Second-Change Algorithm
  - Use a FIFO replacement algorithm.
  - However, inspect its reference bit, when a page has been selected.
    - If the value is 0, proceed to replace it.
    - If the value is 1, give the page a second change and move on to select the next FIFO page.
  - When a page gets a second change,
    - its reference bit is cleared,
    - and its arrival time is reset to the current time.

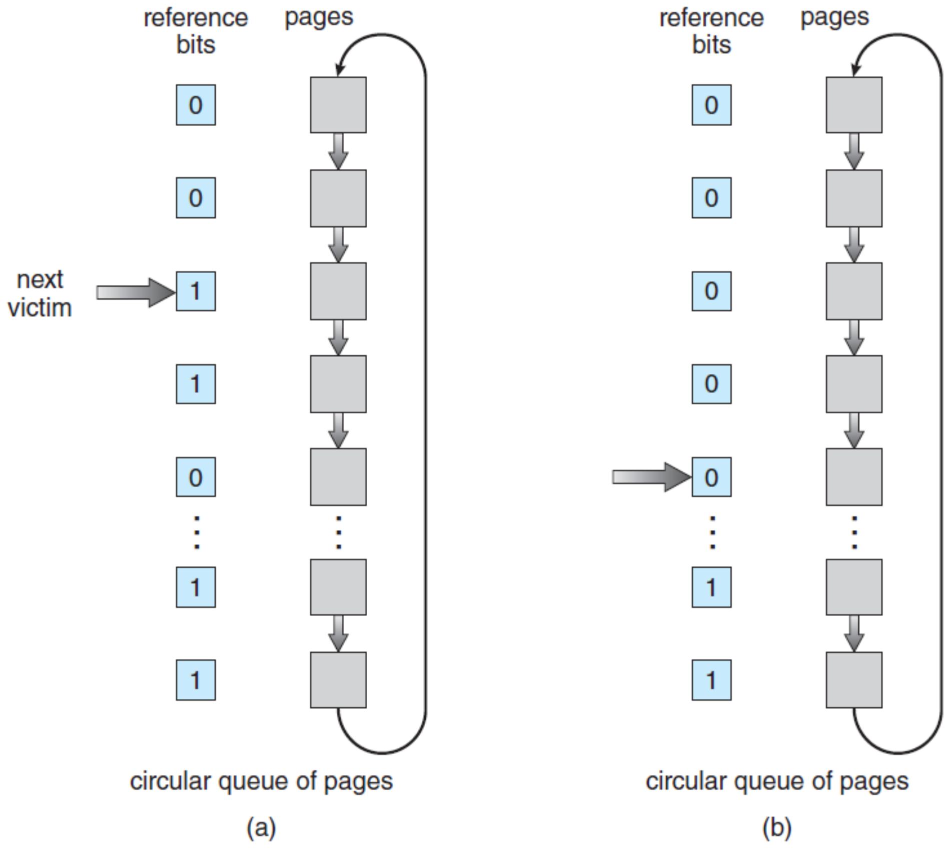


Figure 10.17 *Second-chance (clock) page-replacement algorithm.*

10.5 Allocation of Frames

- The Issues for Frame Allocation:
  - Consider a simple case of a system with 128 frames.
    - OS may take 35, leaving *93 frames* for the user process.
  - Using the *pure-demand-paging*,
    - 93 frames would be put on the *free-frame list*.
  - The first 93 page faults would get free frames.
    - 94th *page faults* would cause a *page replacement*.
  - Then, if we have *two processes*,
    - how do we *allocate 93 frames* to these *two processes*?

10.5 Allocation of Frames

- The strategies for frame allocation
  - Equal .vs. Proportional:
    - *equal allocation*: give every process an equal share.
    - *proportional allocation*: allocate according to the size of process.
  - Global .vs. Local:
    - *local replacement*: selects from *only its own* set of allocated frames.
    - *global replacement*: selects a replacement frame from the set of all the frames in the system.

10.6 Thrashing

- **Thrashing:**
  - a situation that a process is busy swapping pages in and out.
  - If a process does *not* have *enough pages*,
    - the page-fault rate is very high.

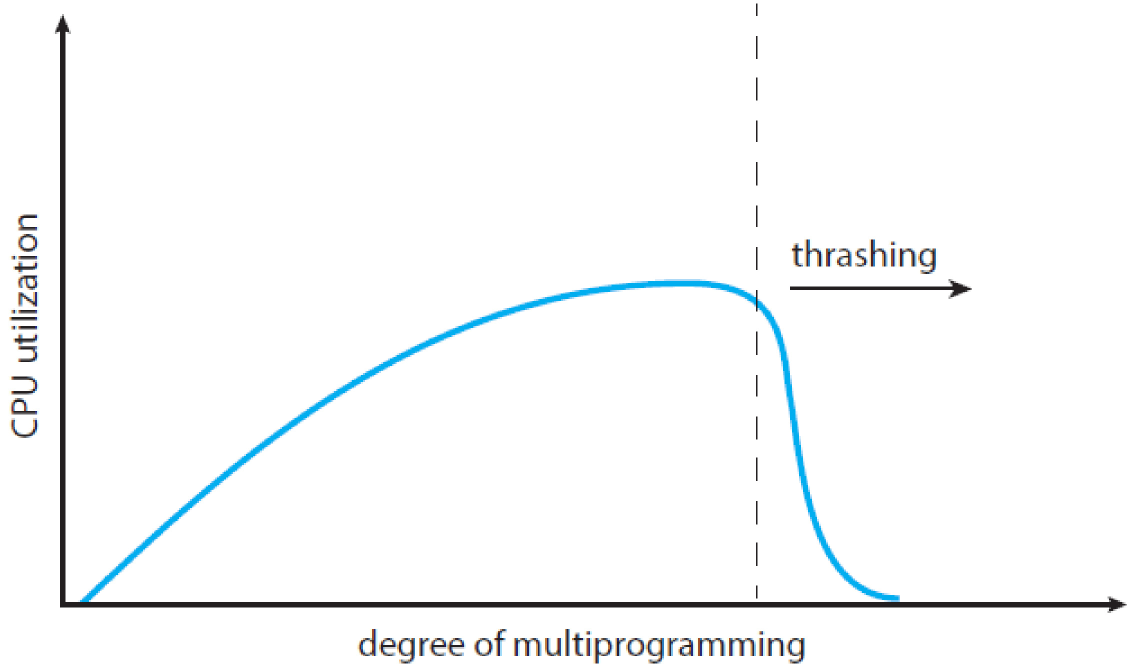


Figure 10.20 *Thrashing.*

10.6 Thrashing

- **Working-Set Model**:
  - based on the assumption of *locality*.
  - define the ***working-set window*** with a parameter Δ.
  - The idea is to examine the most recent Δ page references.
    - **working-set**: the set of pages in the most recent Δ page references
  - If a page is in active use,
    - it will be in the working set.
  - If it is no longer being used,
    - it will drop from the working set time units after its last reference.

10.6 Thrashing

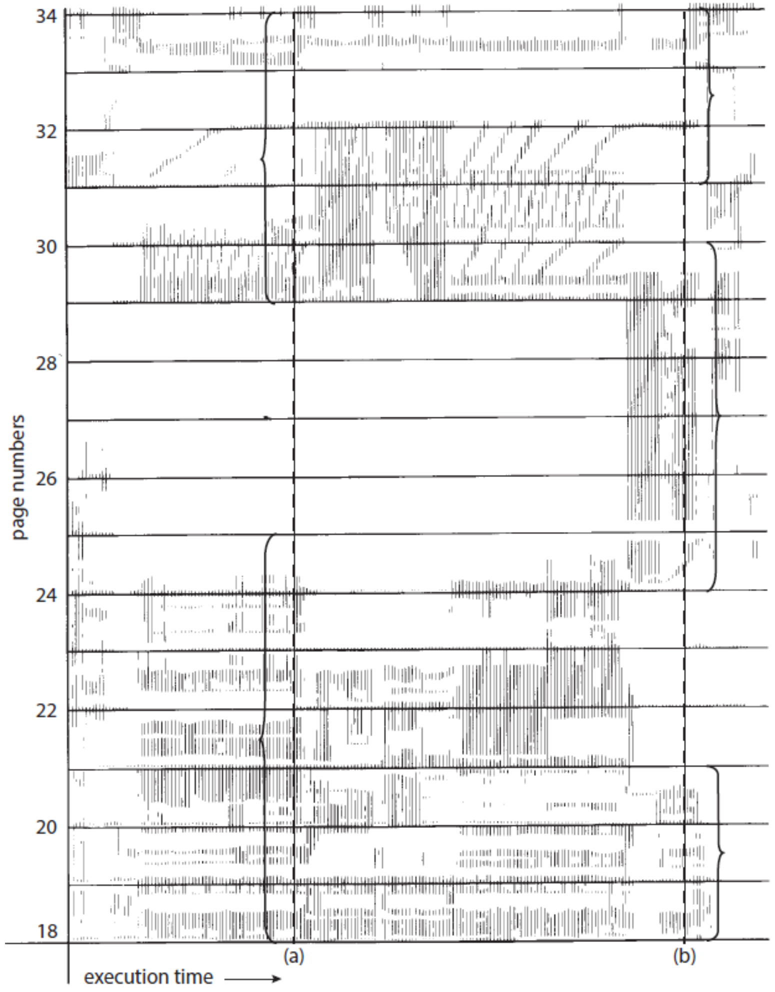


Figure 10.21 *Locality in a memory-reference pattern.*

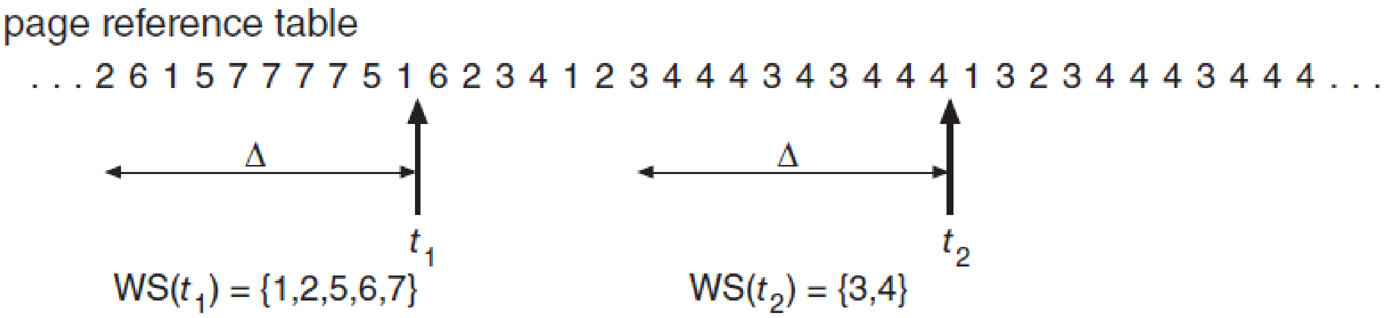


Figure 10.22 *Working-set model.*

Any Questions?