Bucknell UNIVERSITY

Virtual Memory

CSCI 315 Operating Systems Design

Department of Computer Science

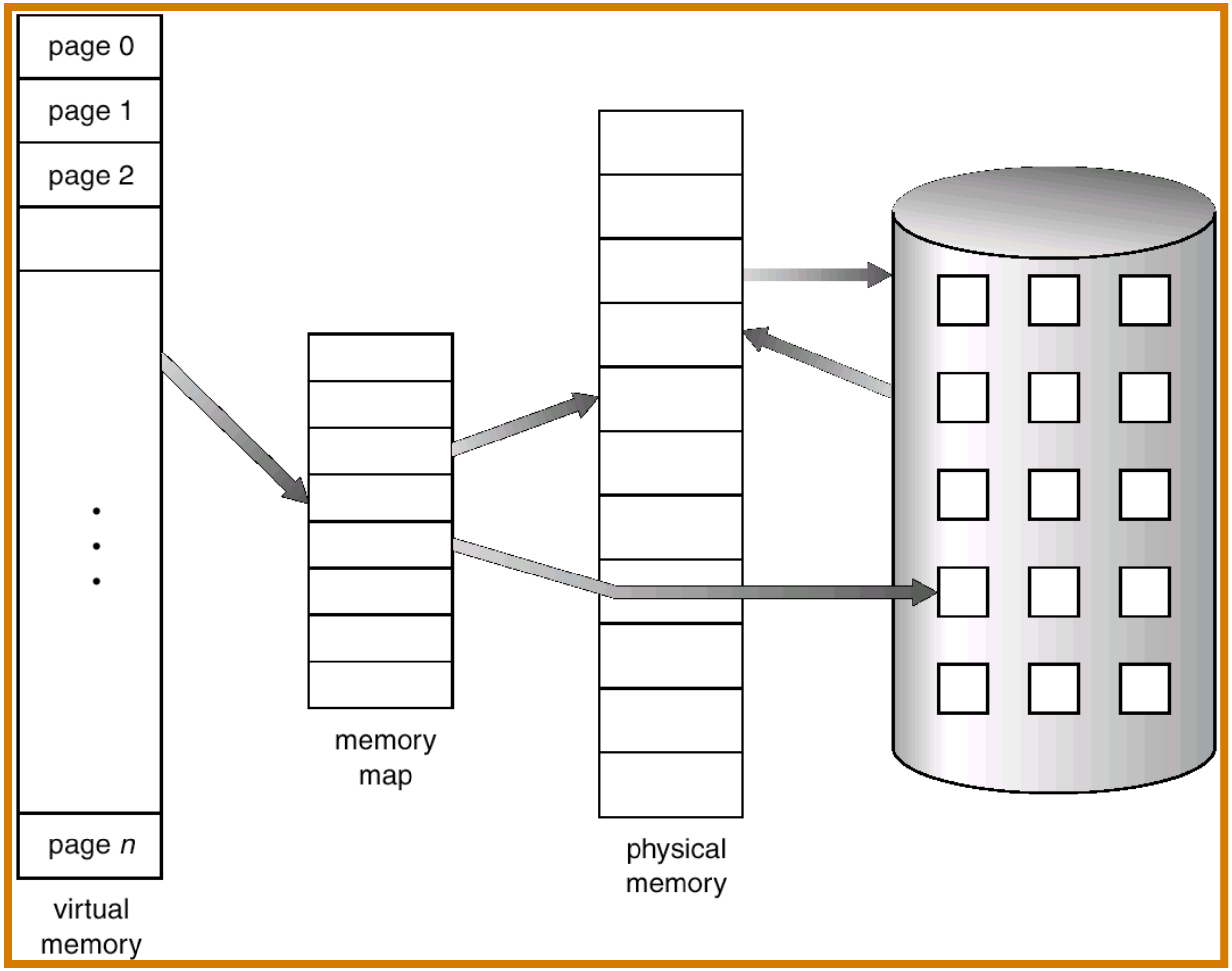
Notice: The slides for this lecture were based on those *Operating Systems Concepts, 9th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.

Virtual Memory

- **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.

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

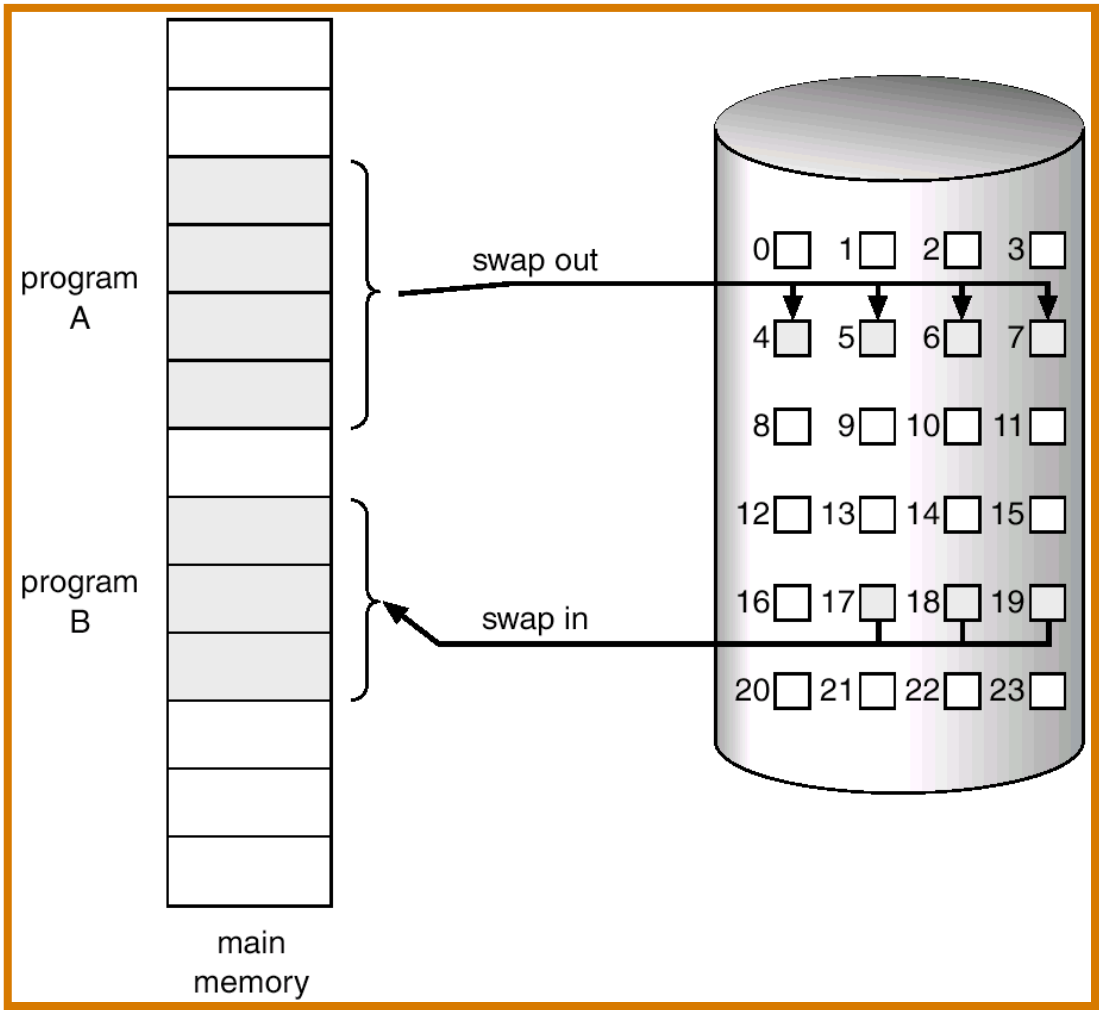
Virtual Memory Larger than Physical Memory



Demand Paging

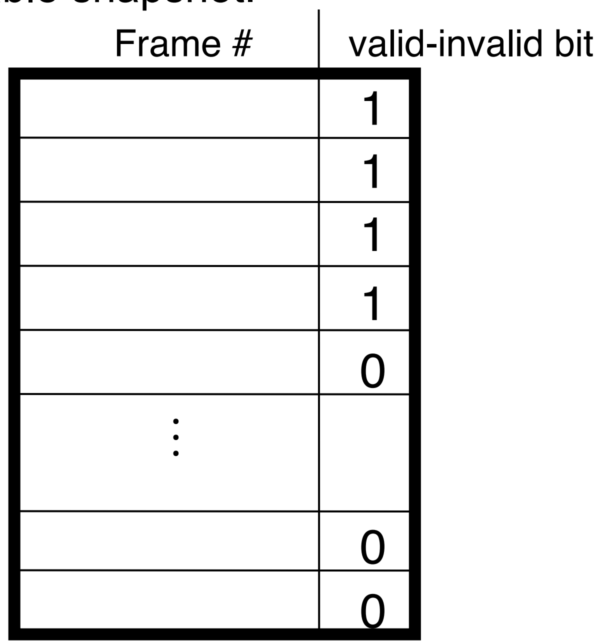
- Bring a page into memory only when it is needed.
  - Less I/O needed.
  - Less memory needed.
  - Faster response.
  - More users.
- Page is needed (there is a reference to it):
  - invalid reference ⇒ abort.
  - not-in-memory ⇒ bring to memory.

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

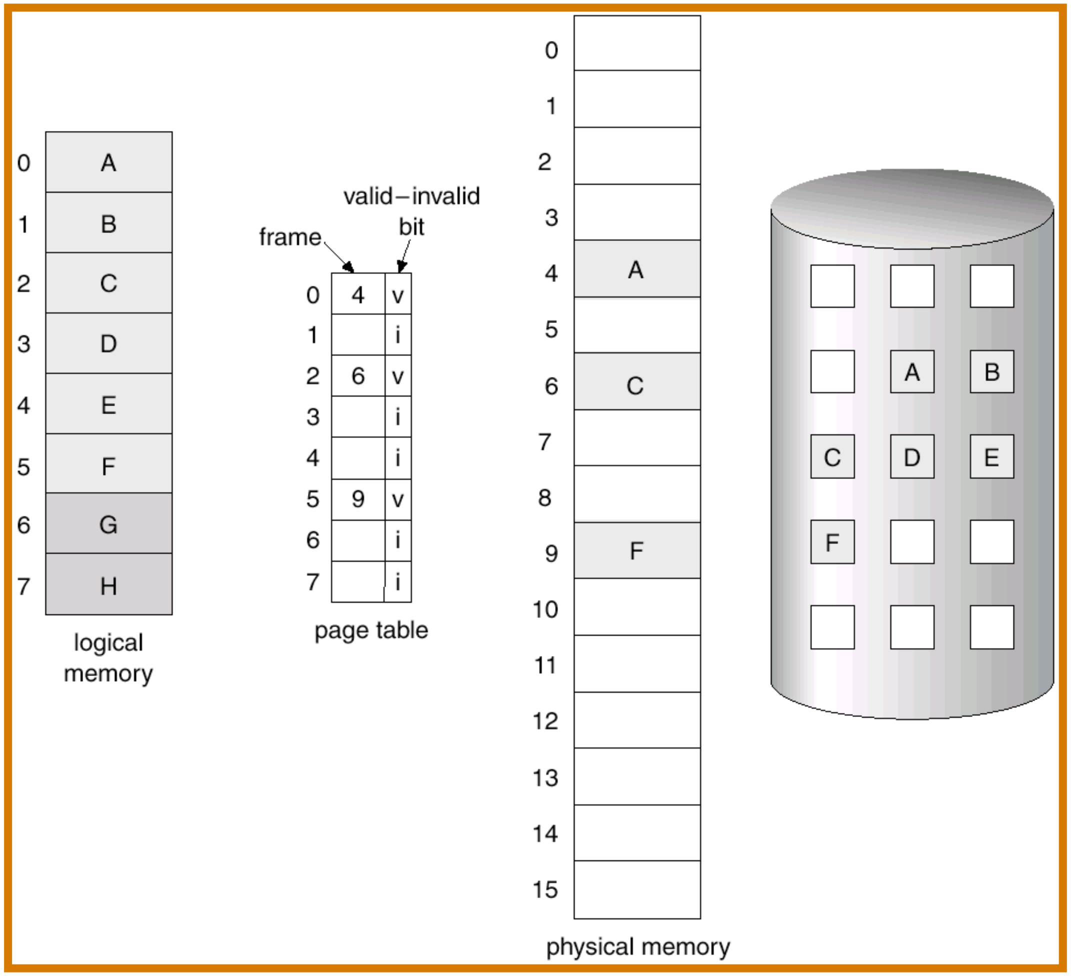
- With each page table entry a valid–invalid bit is associated
(1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.



page table

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault.

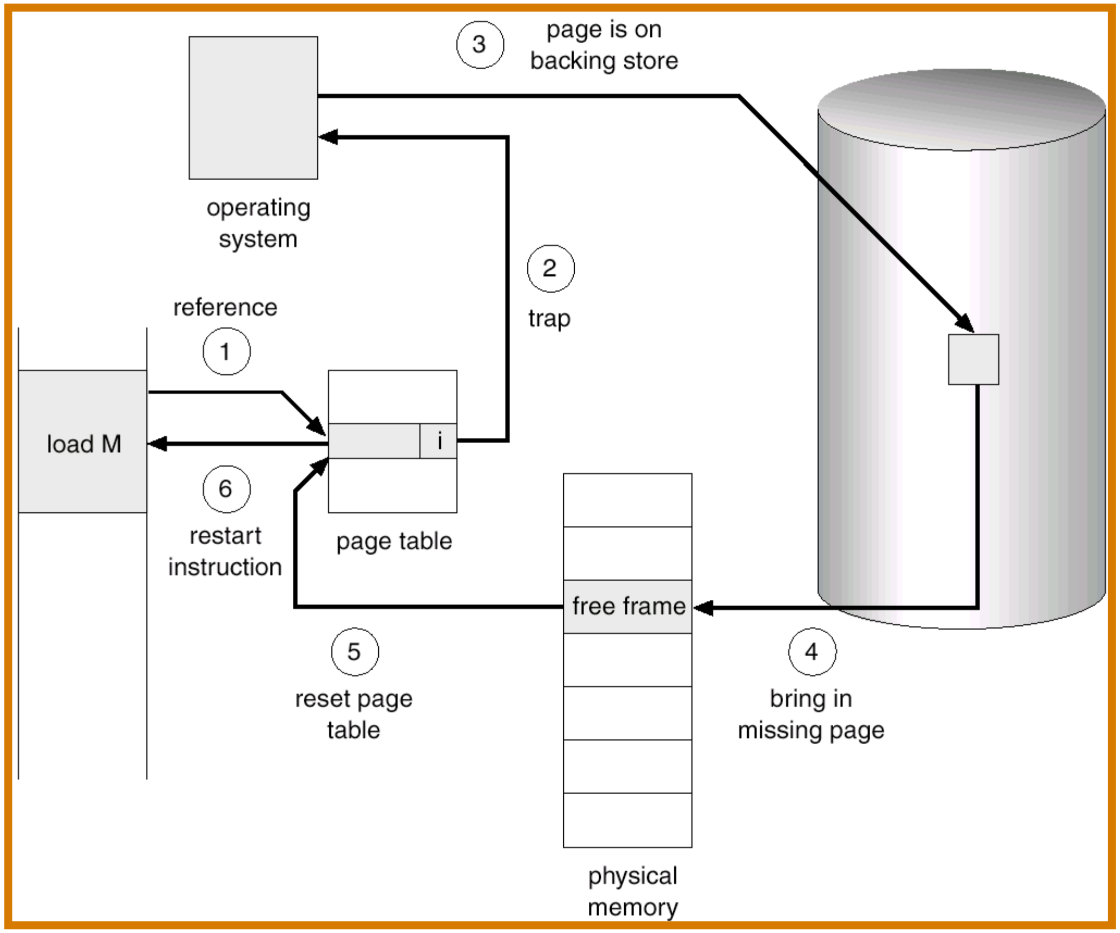
Page Table when some pages are not in Main Memory



Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault.
- OS looks at page table to decide:
  - If it was an invalid reference ⇒ abort.
  - If it was a reference to a page that is not in memory, continue.
- Get an empty frame.
- Swap page into frame.
- Correct the page table and make validation bit = 1.
- Restart the instruction that caused the page fault.

Steps in Handling a Page Fault



What if there is no free frame?

- Page replacement – find some page in memory, that is not *"really"* in use and swap it out.
  - Must define an algorithm to select what page is replaced.
  - Performance: want an algorithm which will result in minimum number of page faults.

- The same page may be brought in and out of memory several times.

No free frame: now what?

- **Page replacement:** Are all those pages in memory being referenced? Choose one to swap back out to disk and make room to load a new page.
  - **Algorithm:** How you choose a victim.
  - **Performance:** Want an algorithm that will result in **minimum** number of page faults.

- Side effect: The same page may be brought in and out of memory several times.

Performance of Demand Paging

- **Page Fault Rate:** $0 \le p \le 1.0$
  - if $p = 0$ no page faults.
  - if $p = 1$, every reference is a fault.

- **Effective Access Time (EAT):**

  EAT = [(1 – *p*) (memory access)] + [*p* (page fault overhead)]

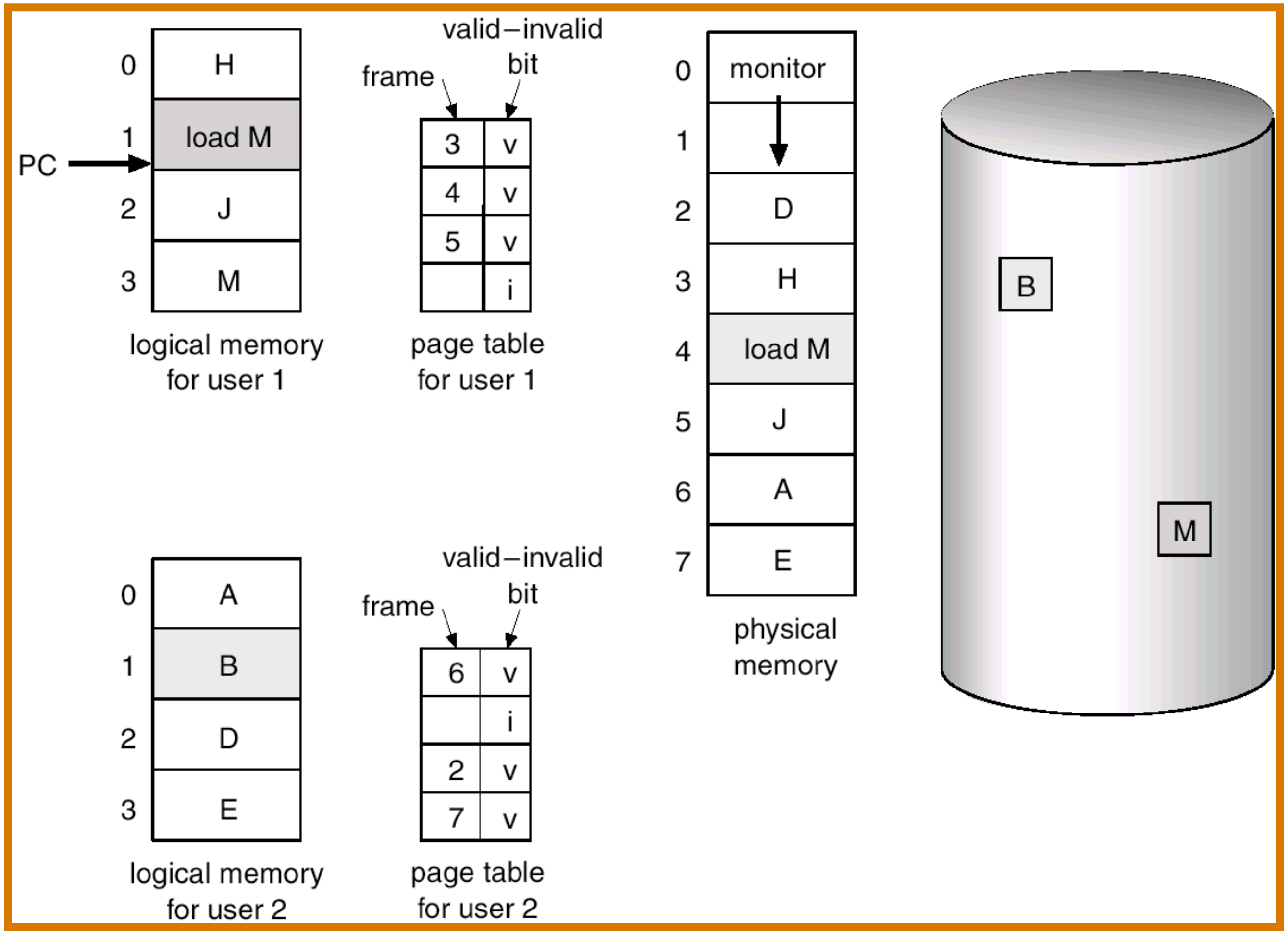
where:

page fault overhead = [swap page out] + [swap page in]
+ [restart overhead]

Page Replacement

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

Need For Page Replacement



Basic Page Replacement

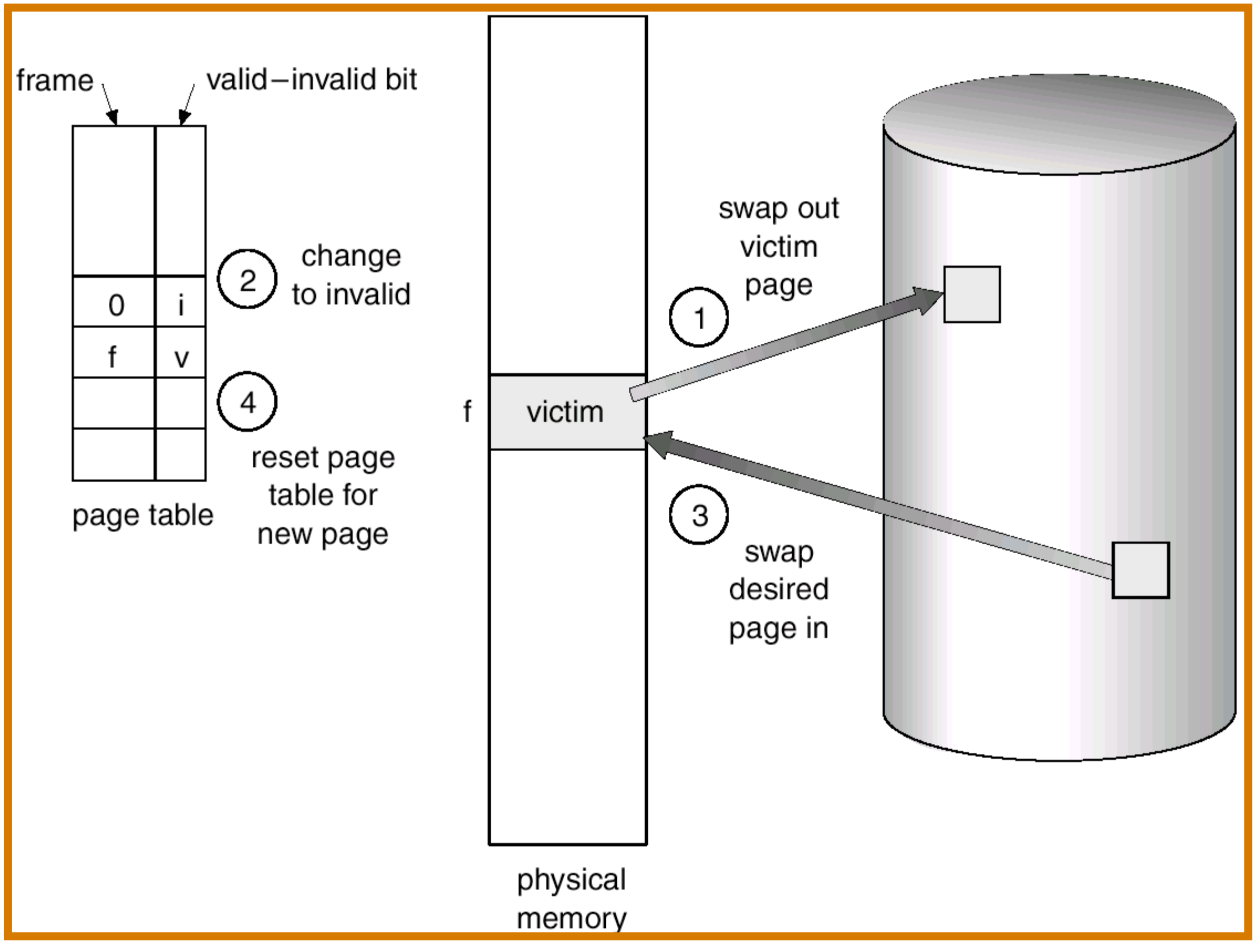
1. Find the location of the desired page on disk.

2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a *victim* frame.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

Page Replacement

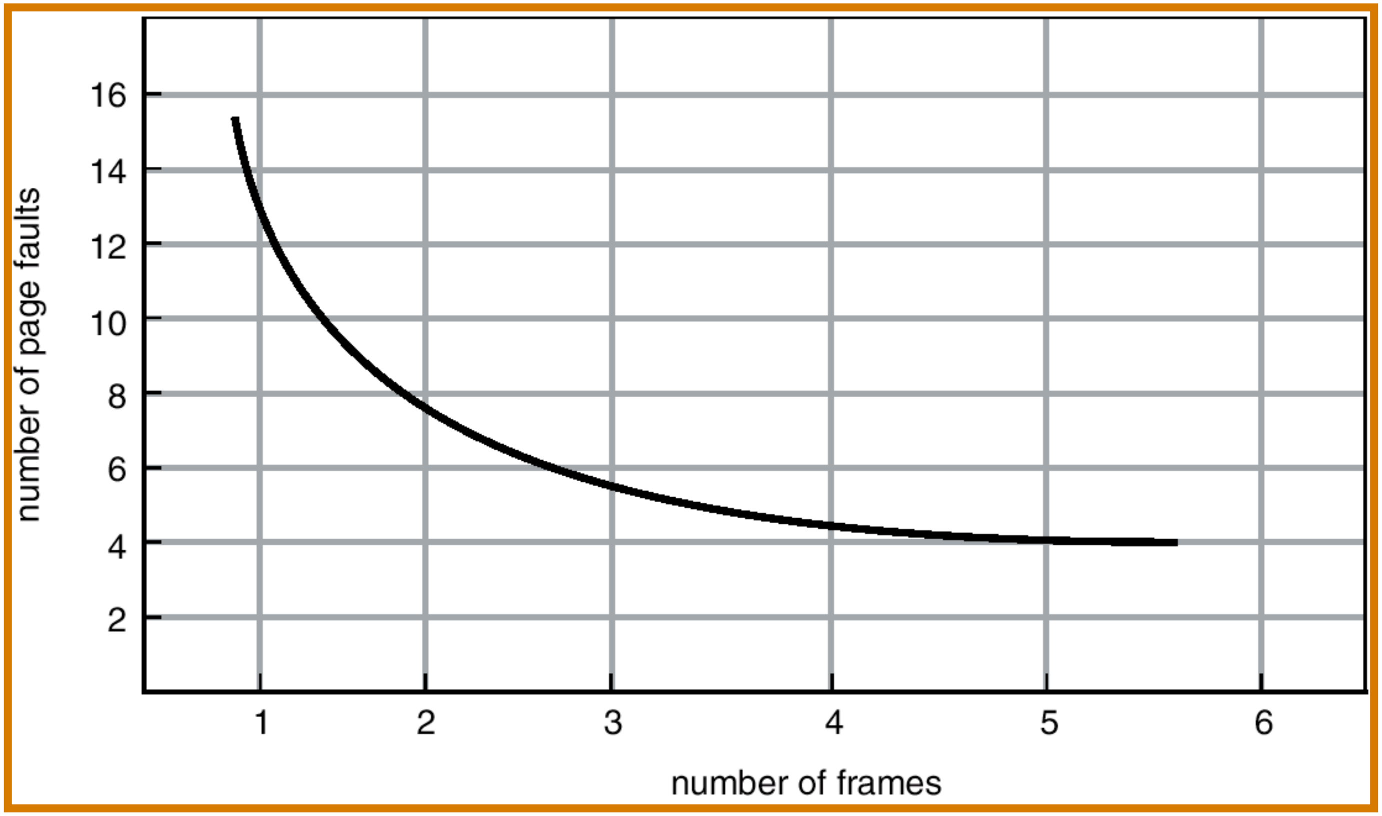


Page Replacement Algorithms

- **Goal:** Produce a low page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (***reference string***) and computing the number of page faults on that string.
- The reference string is produced by tracing a real program or by some stochastic model. We look at every address produced and strip off the page offset, leaving only the page number. For instance:

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

Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- 3 frames (3 pages can be in memory at a time per process)

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

9 page faults

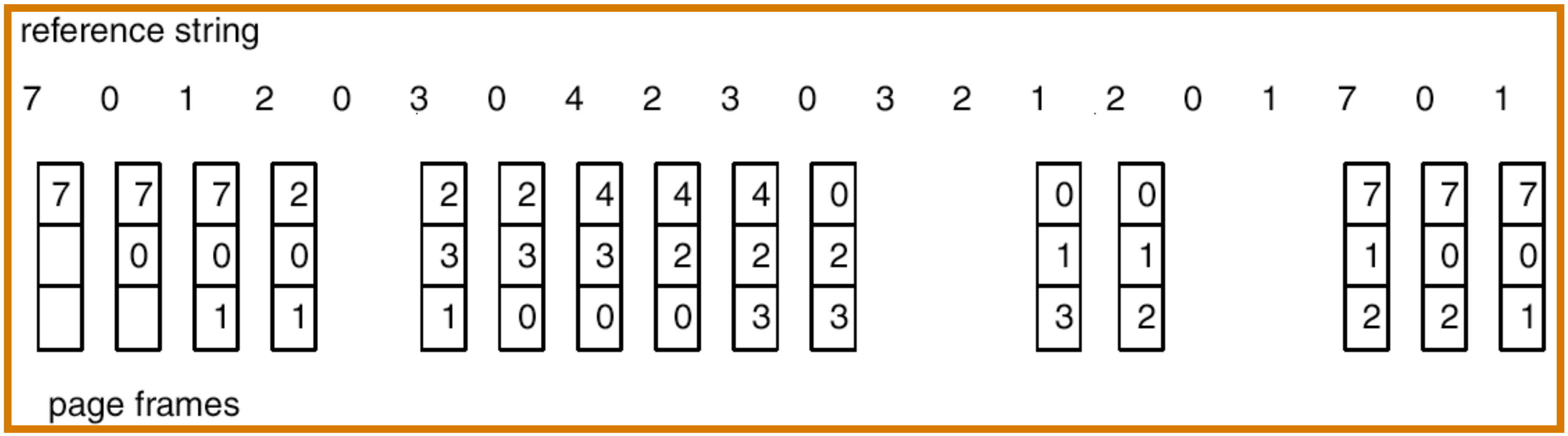
- 4 frames

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

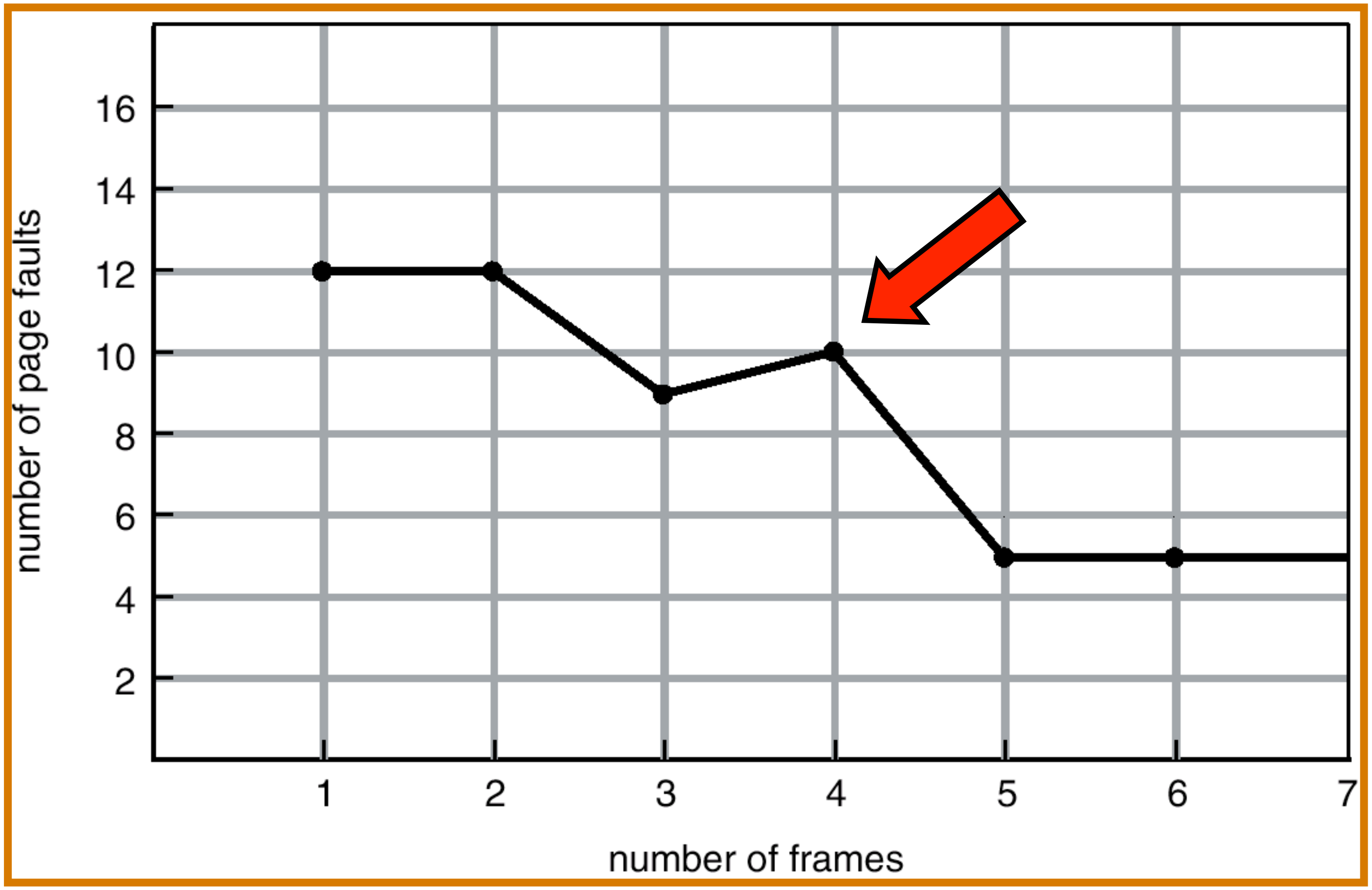
10 page faults

- FIFO Replacement ⇒ **Belady's Anomaly:** more frames, *more* page faults.

FIFO Page Replacement

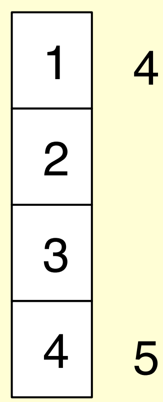


FIFO (Belady's Anomaly)



Optimal Algorithm

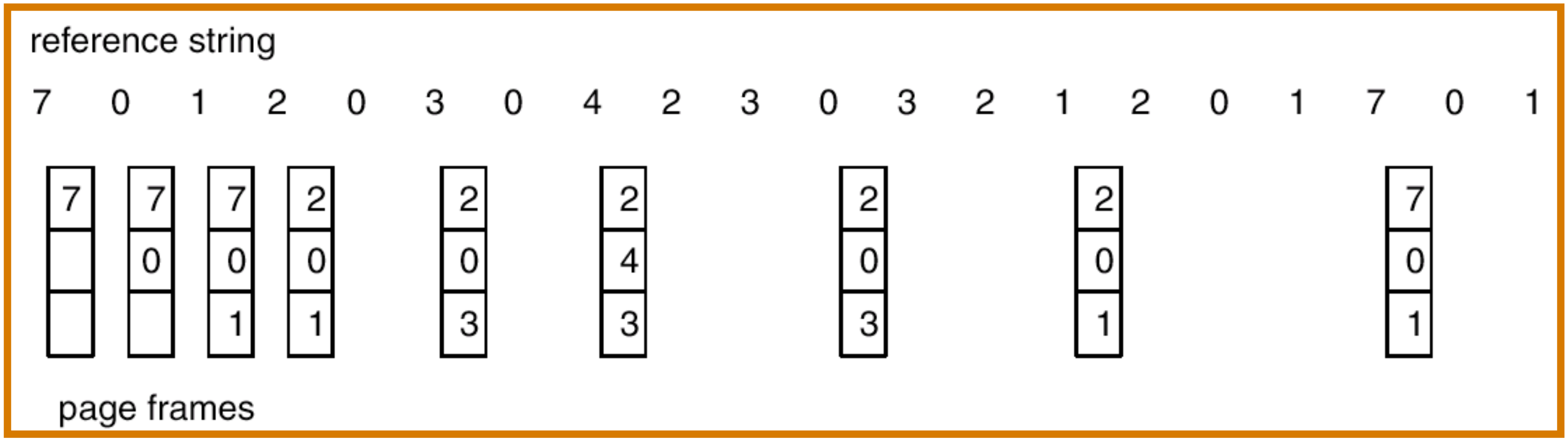
- Replace the page that will not be used for longest period of time. (How can you know what the future references will be?)
- 4 frames example: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



6 page faults

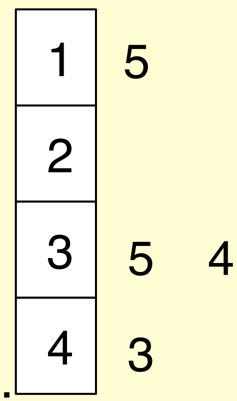
- Used for measuring how well your algorithm performs.

Optimal Page Replacement



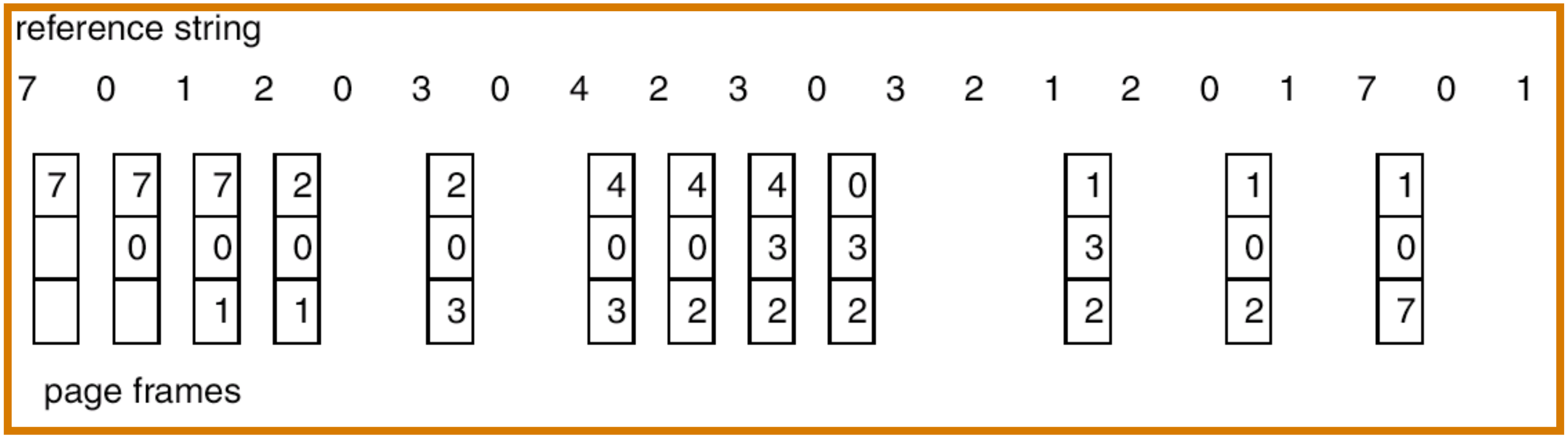
LRU Algorithm

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



- Counter implementation:
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.

LRU Page Replacement



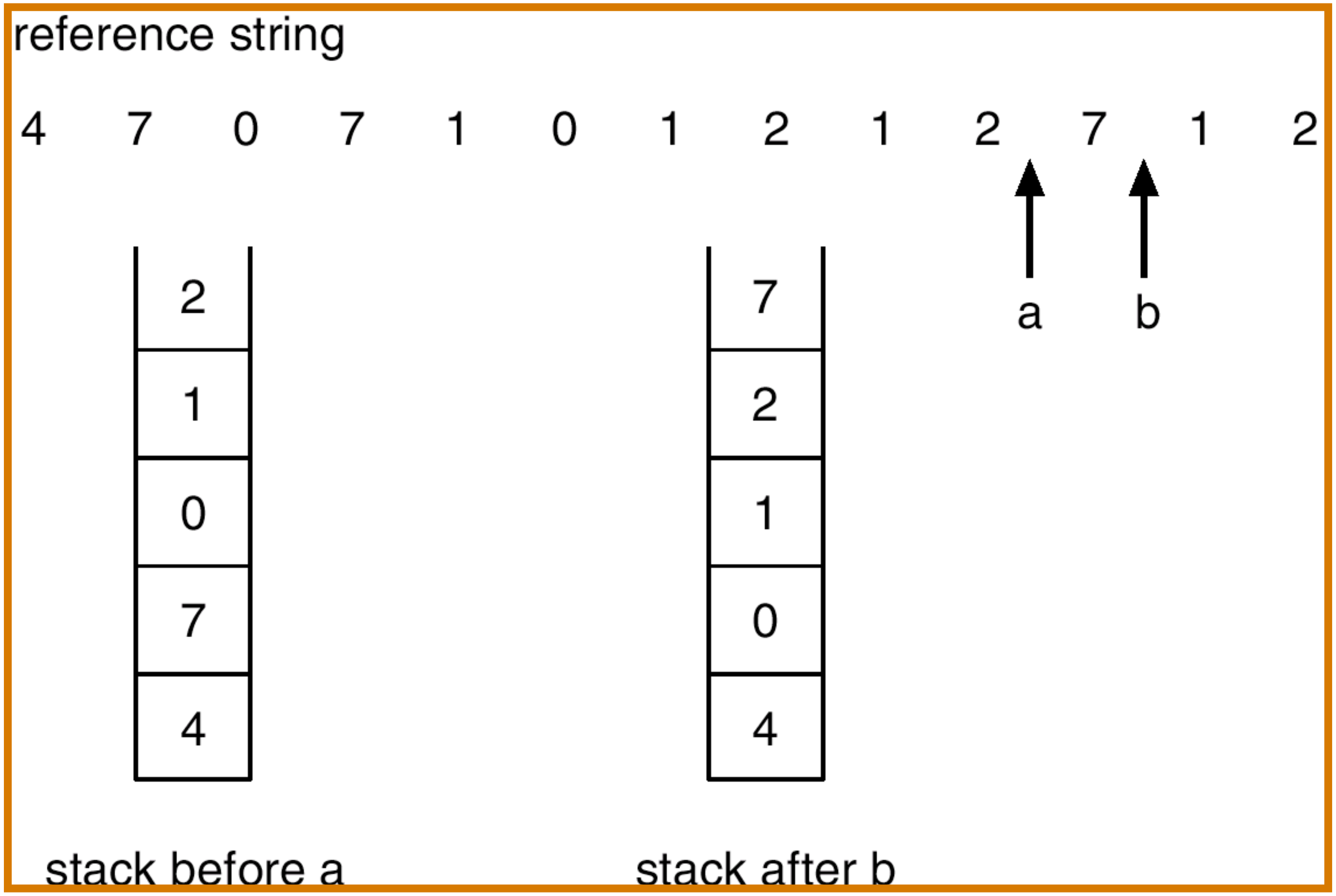
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement.

LRU and Belady's Anomaly

- LRU does not suffer from Belady's Anomaly (OPT doesn't either).
- It has been shown that algorithms in a class called **stack algorithms** can never exhibit Belady's Anomaly.
- A **stack algorithm** is one for which the set of pages in memory for n frames is a subset of the pages that Could be in memory for $n+1$ frames.

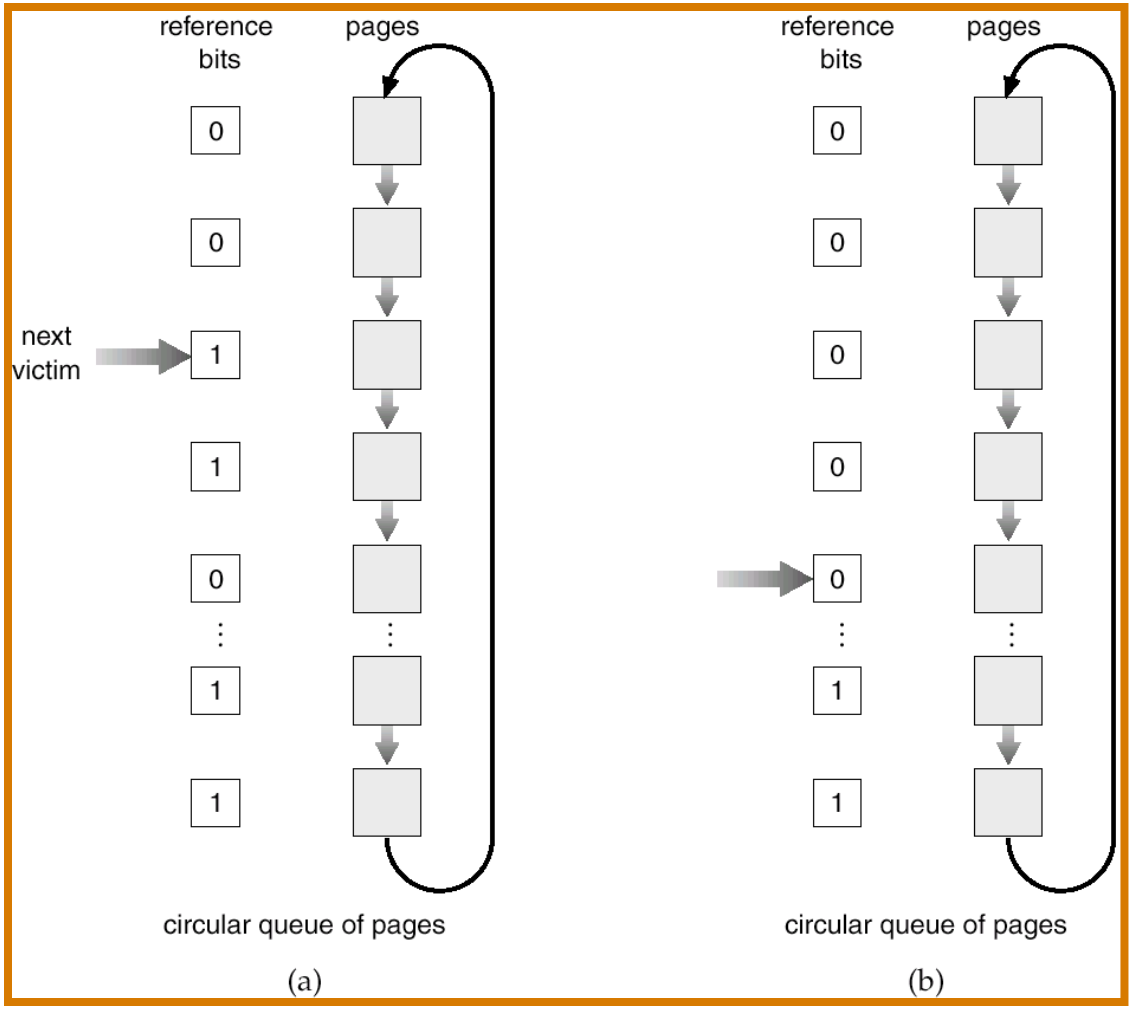
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- **Second chance**
  - Need reference bit.
  - Clock replacement.
  - If page to be replaced (in clock order) has reference bit = 1. then:
    - set reference bit 0.
    - leave page in memory.
    - replace next page (in clock order), subject to same rules.

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **LFU Algorithm:** replaces page with smallest count.
- **MFU Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Allocation of Frames

- Each process needs a **minimum** number of pages.

- There are two major allocation schemes:
  - **fixed allocation**
  - **priority allocation**

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$
\begin{cases}
m = 64 \\
s_i = 10 \\
s_2 = 127 \\
a_1 = \frac{10}{137} \times 64 \approx 5 \\
a_2 = \frac{127}{137} \times 64 \approx 59
\end{cases}
$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

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

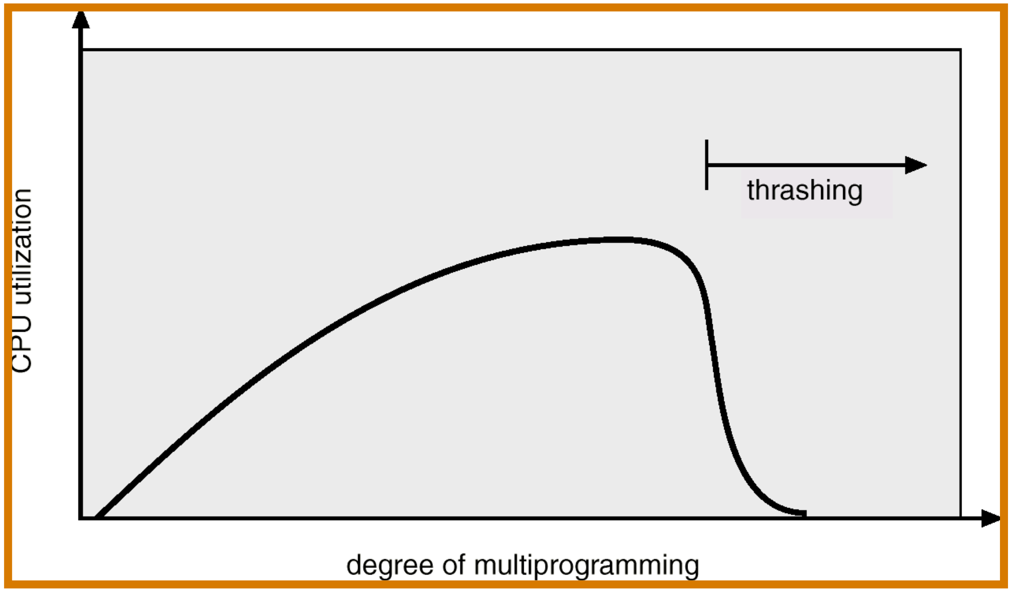
Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local** replacement – each process selects from only its own set of allocated frames.

Thrashing

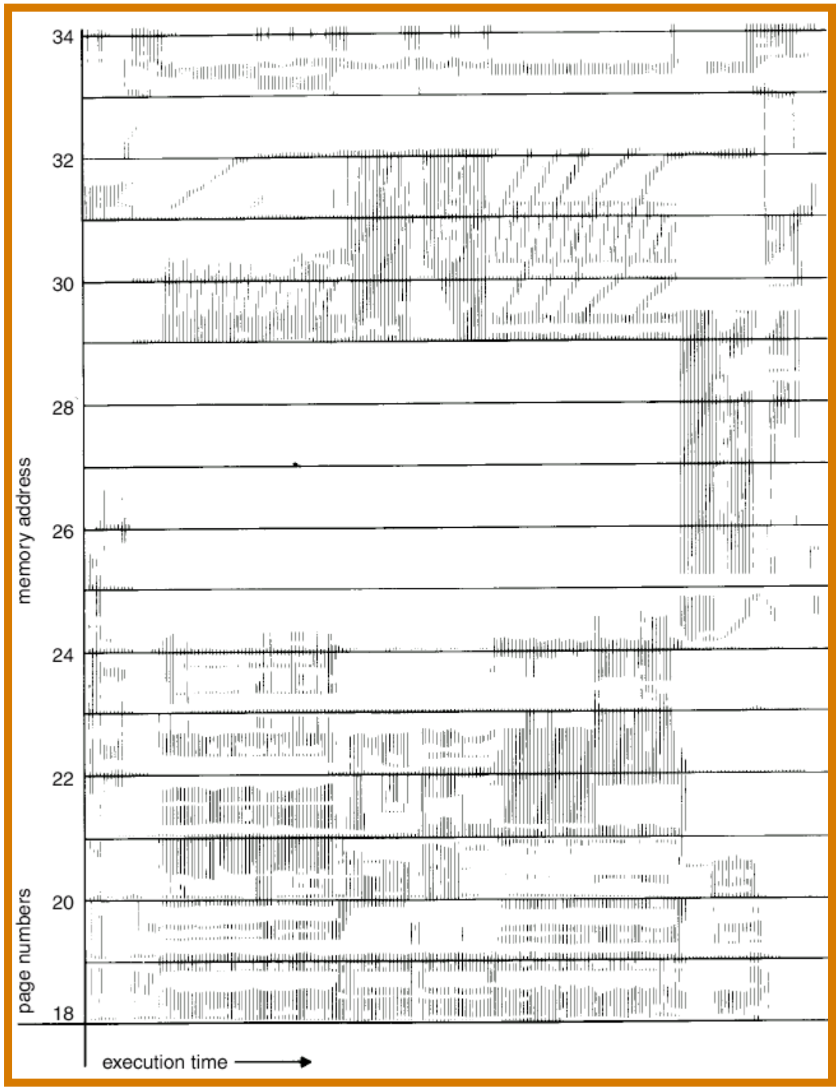
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - **Low CPU utilization**.
  - Operating system thinks that it needs to increase the degree of multiprogramming.
  - Another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in and out.

Thrashing



- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

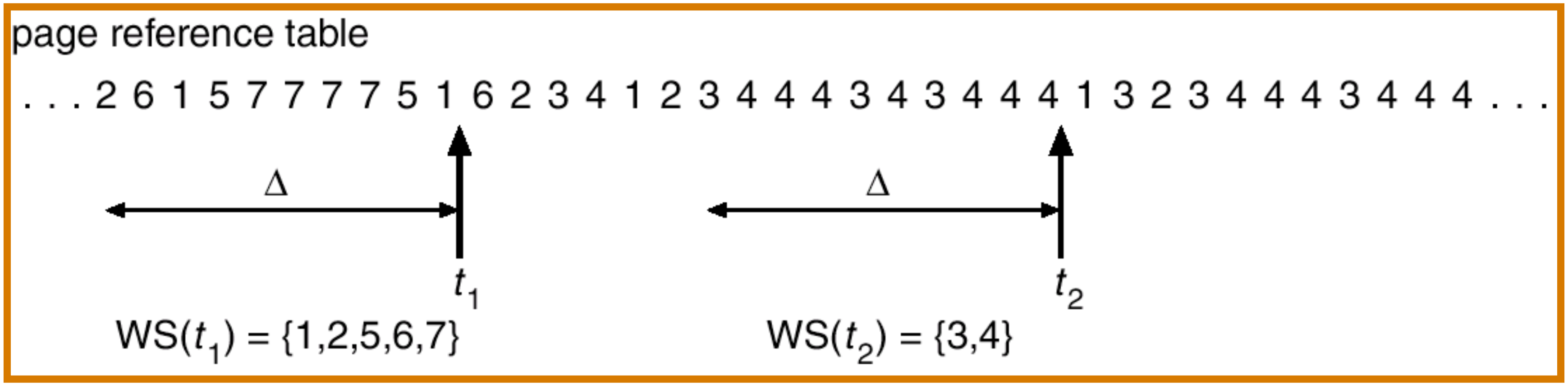
Locality in Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ **working-set window** $\equiv$ a fixed number of page references.
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in 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
- if $D > m \Rightarrow$ **Thrashing**
- Policy if D > m, then suspend one of the processes.

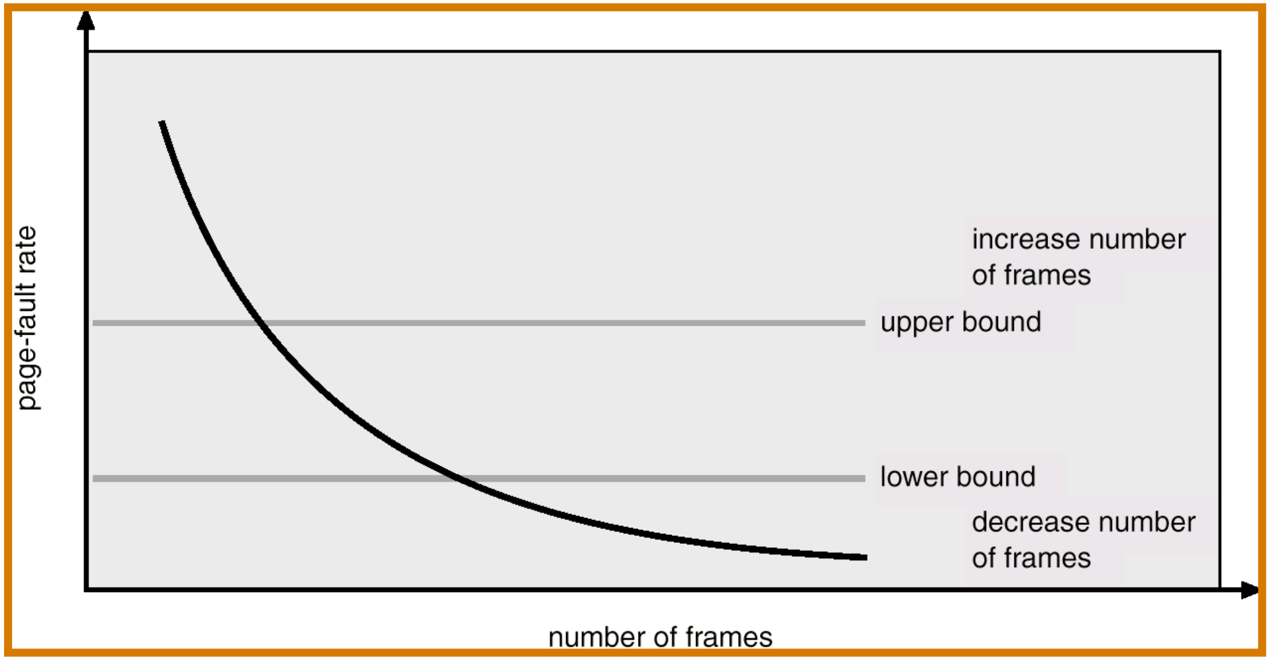
Working-set model



Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

Page-Fault Frequency Scheme



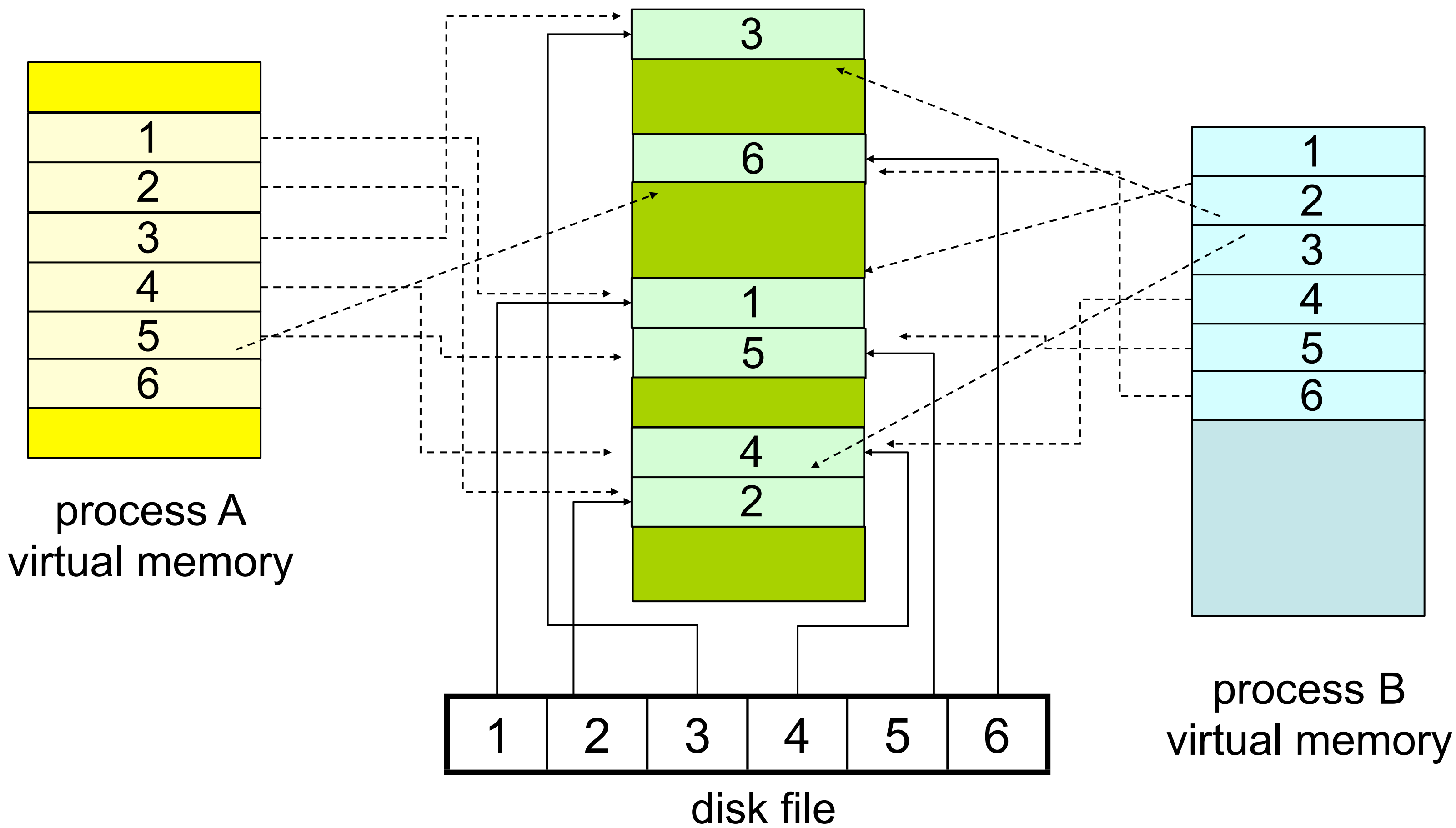
Establish "acceptable" page-fault rate.

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

Memory-mapped Files

- Memory mapping a file can be accomplished by mapping a disk block to one or more pages in memory.
- A page-sized portion of the file is read from the file system into a physical page. Subsequent `read()` and `write()` operations are handled as memory (not disk) accesses.
- Writing to the file in memory is not necessarily synchronous to the file on disk. The file can be committed back to disk when it's closed.

Memory-mapped Files



Prepaging

- **Prepaging**: In order to avoid the initial number of page faults, the system can bring into memory all the pages that will be needed all at once.

- This can also be applied when a swapped-out process is restarted. The smart thing to do is to remember the working set of the process.

- One question that arises is whether all the pages brought in will actually be used…

- Is the cost of prepaging less than the cost of servicing each individual page fault?