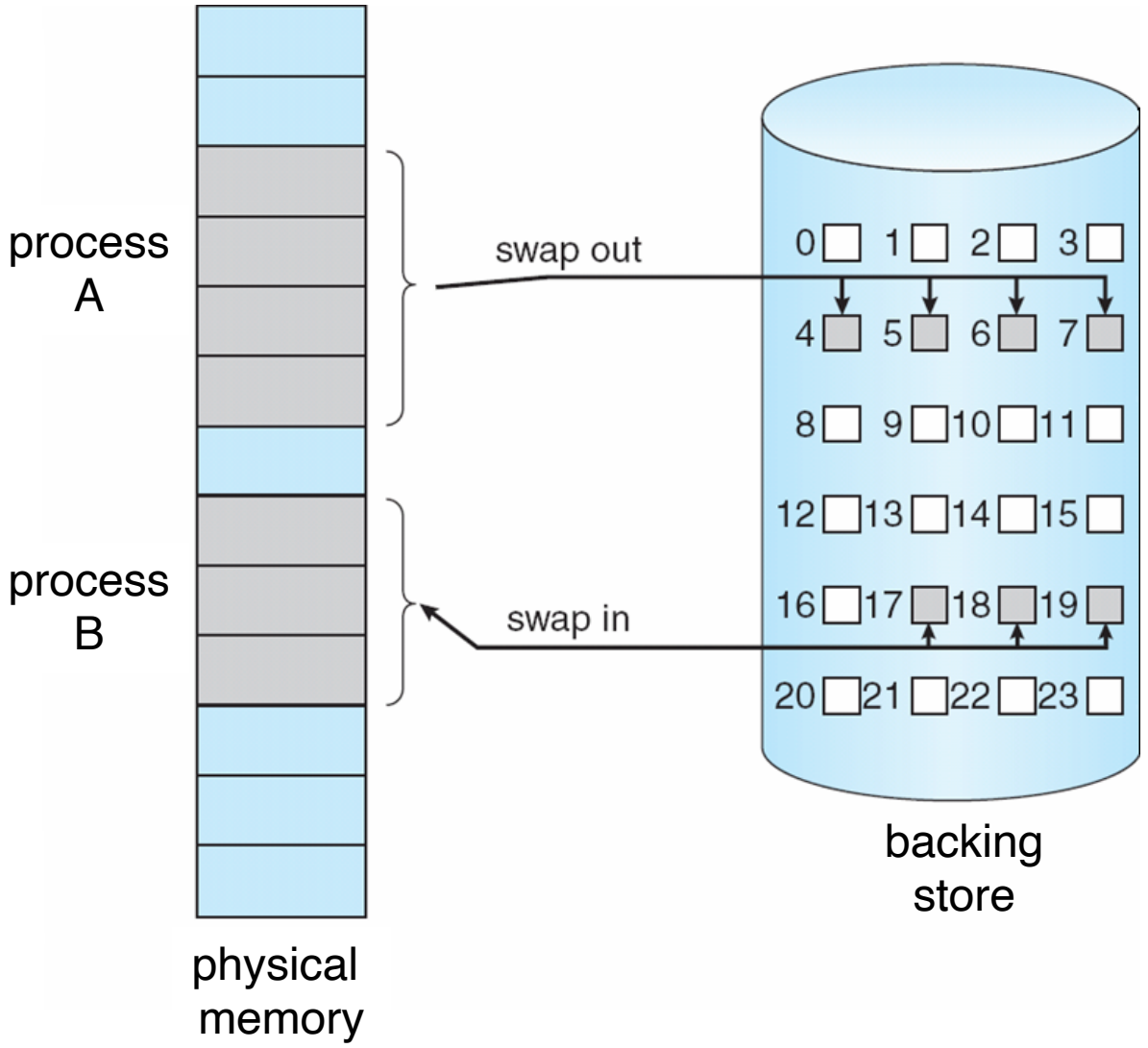
Part 8: Virtual Memory

- Introduction
  - Swapping, Virtual Memory Support
- Demand Paging
  - Page Fault, Performance of Demand Paging, Page Replacement
- Page-Replacement Algorithms
  - FIFO, Optimal, LRU, LRU Implementation, and LRU Approximation
- Allocation of Frames
  - Fixed vs. Variable Allocation, Local vs. Global Replacement
- Thrashing
  - Cause of Thrashing, Working-Set Model, Page-fault Frequency Scheme

Introduction - Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Swapping can be used to
  - Make space available for processes that require more memory
  - Increase the *degree of multiprogramming*

Introduction – Swapping (cont.)



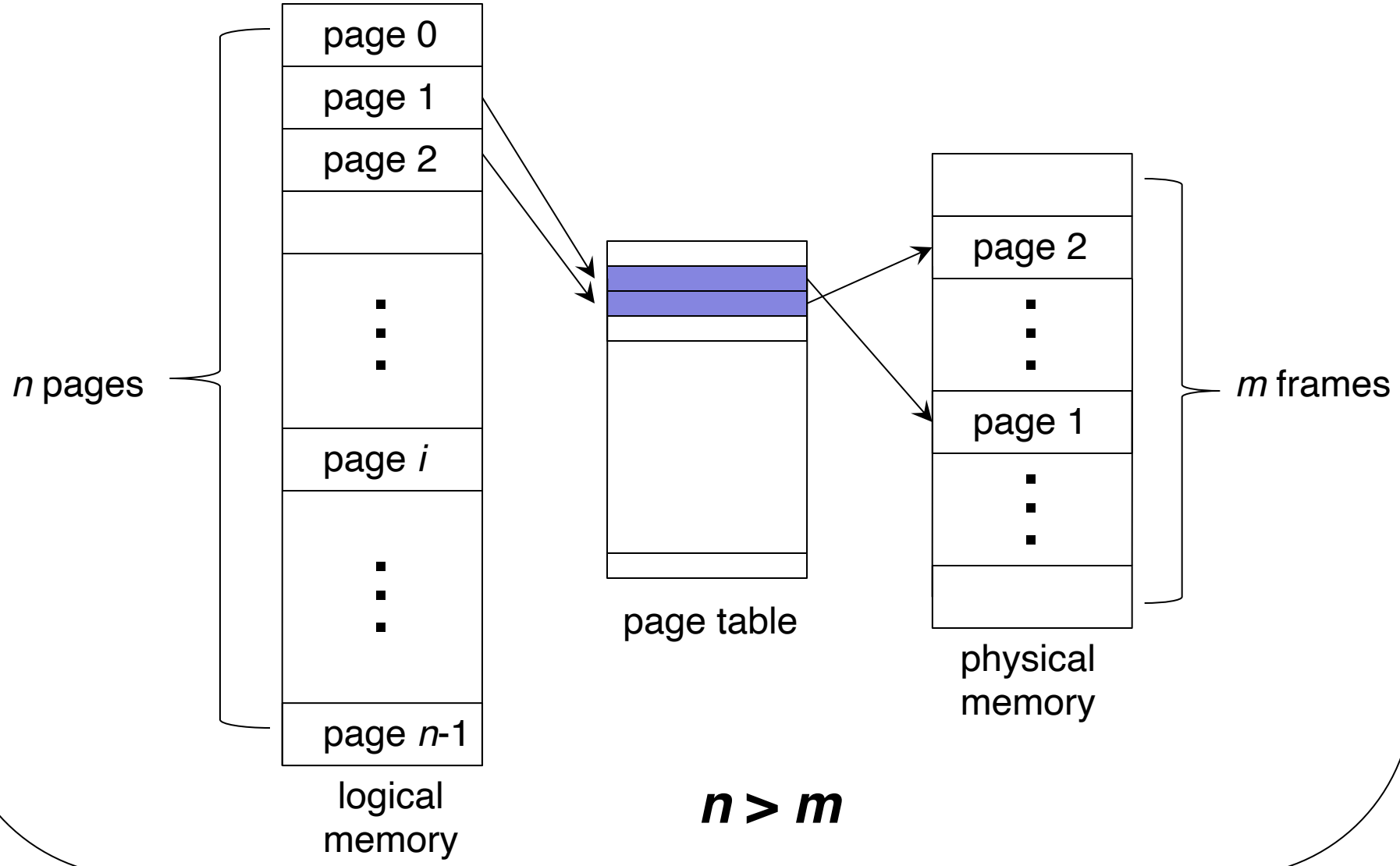
Introduction – Virtual Memory Support

- So far we assumed that the whole program is loaded into the memory, at least at some point of time.
  Restriction: Overall program size, including all library routines that may be linked at runtime, must be restricted to the size of physical memory

- To remove this restriction, the logical memory size requirement of the user program must be de-coupled from the size of the physical memory space

- This is what virtual memory aims to achieve
  - The size of virtual memory is limited by the address scheme of a computer, not by the actual size of the physical memory.

Introduction – Virtual Memory Support (Cont.)



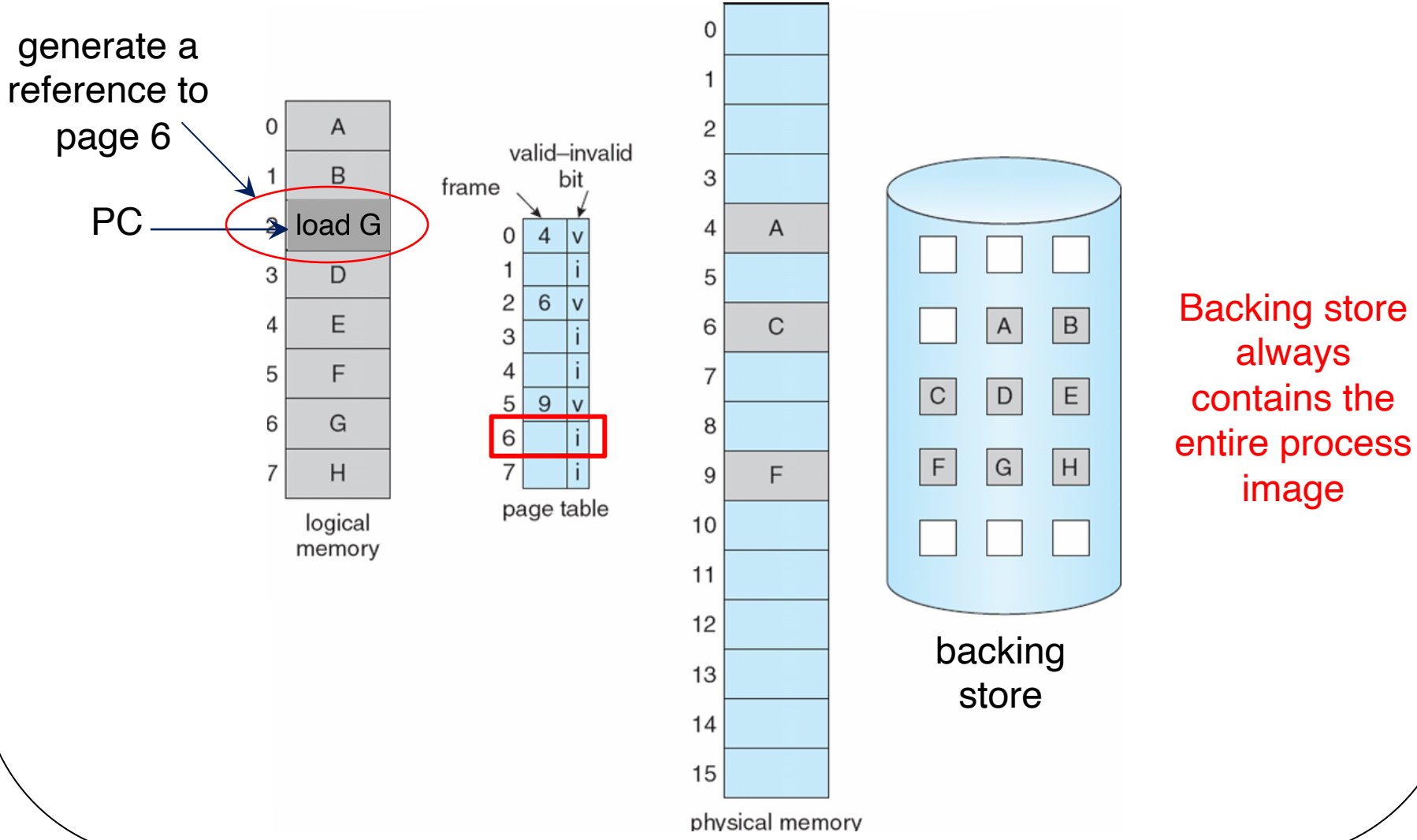
Introduction – Virtual Memory Support (cont.)

- Virtual Memory Support: separation of user logical memory from the physical memory
- Implications of Virtual Memory Support:
  - Logical address space can be much larger than available memory
  - Only part of the program needs to be in memory for execution $\Rightarrow$ greater degree of multiprogramming $\Rightarrow$ increased CPU utilization
  - Need to allow pages to be paged in and out
- Virtual memory is implemented (mostly) using *demand paging* (demand segmentation is also used)

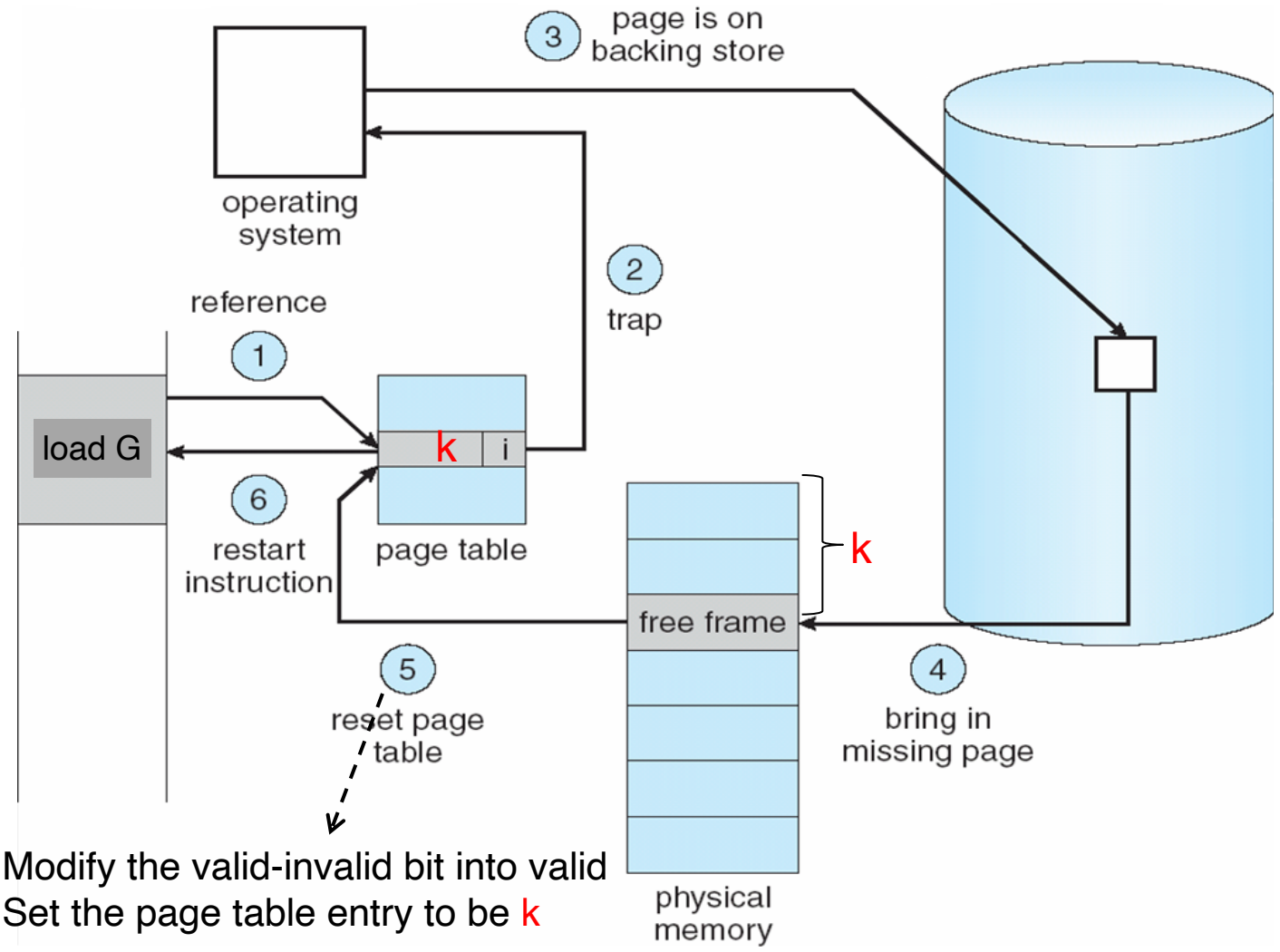
Demand Paging

- Memory requirement of a process is divided into pages
- logical address <page no., offset> $\Rightarrow$ a reference generated for the page $\Rightarrow$ page is needed
- With each page table entry, a *valid–invalid bit* is associated
  - **v** $\Rightarrow$ in-memory, page table entry contains location of the page in memory (i.e., frame number)
  - **i** $\Rightarrow$ not-in-memory, page table entry contains the address of the page on disk
  - Initially valid–invalid bit is set to **i** on all entries
  - During address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ *page fault* (to bring the page into memory)

Demand Paging (Cont.)



Page Fault



(5.1) Modify the valid-invalid bit into valid
(5.2) Set the page table entry to be k

Page Fault (Cont.)

1. If there is a reference to a page, page table is checked to determine if the page is in the memory.
2. If the page is not in the memory, trap to OS to bring in the page. The state of interrupted process is saved.
3. OS uses the page table entry to locate the page in backing store.
4. OS finds a free frame (i.e., using free frame list & page replacement algorithm), and schedules a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, page table entry is modified.
6. The state of interrupted process is restored, and the instruction is now re-started.

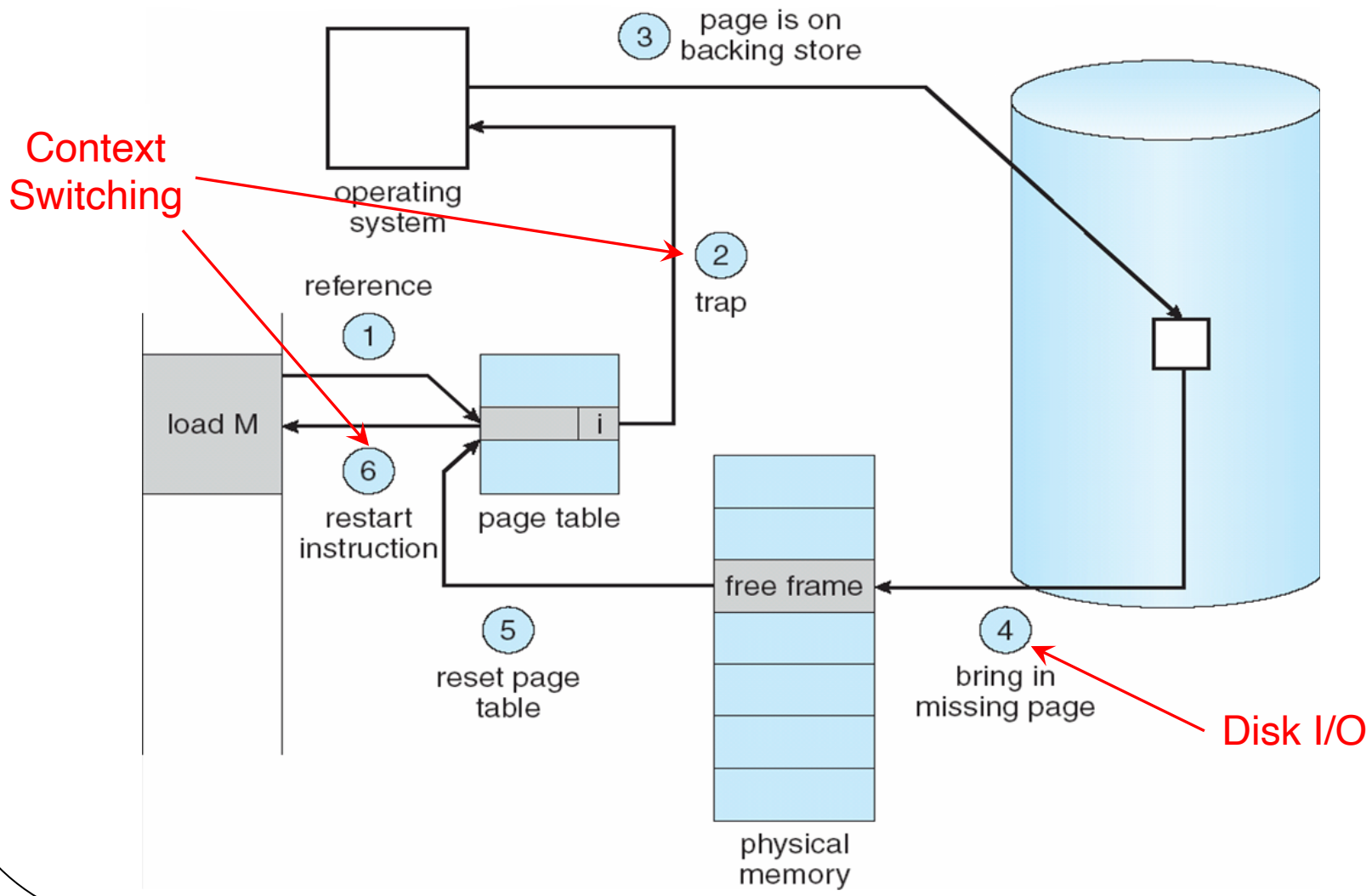
Performance of Demand Paging

- Let p = probability of Page Fault, $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference causes a fault
- Effective Access Time (EAT)

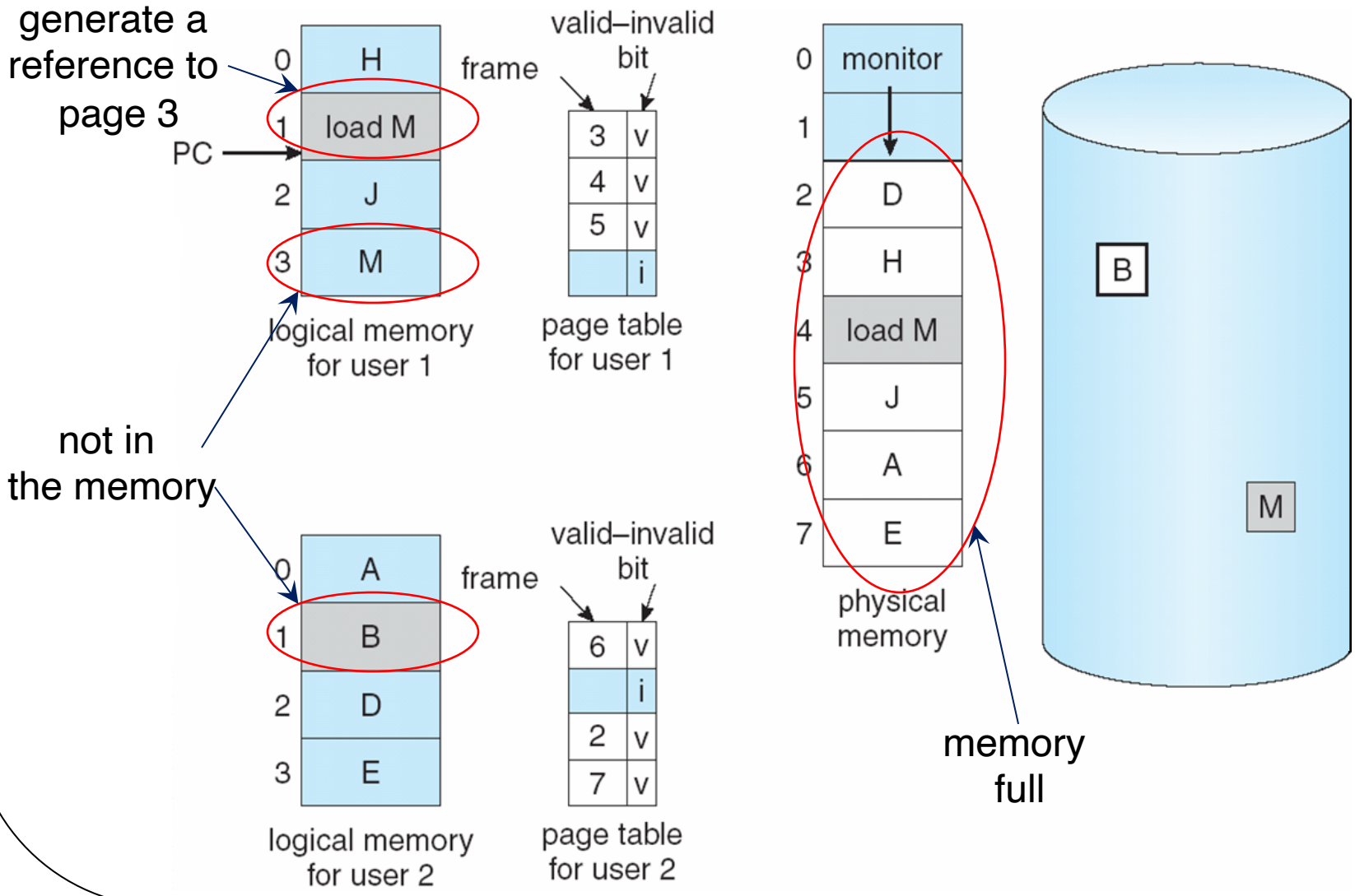
$$EAT = (1 - p) \times \text{memory access time} + p \times \text{page fault time}$$

- In practice, memory access time << page fault time.

Performance of Demand Paging (Cont.)



Page Replacement



Page Replacement (Cont.)

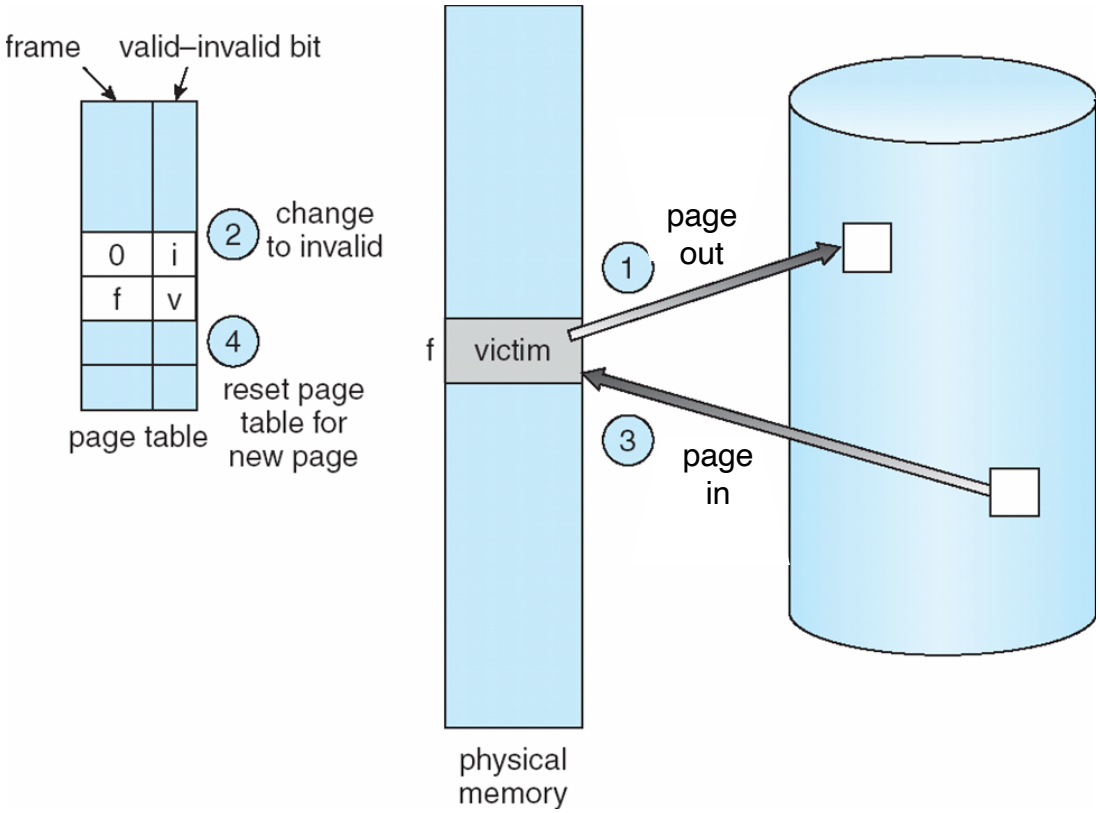
- When page fault occurs, what happens if there is no free frame?

  Solution: Find some page in memory, not really in use, and page it out (i.e., replace page)

- If no free frame is available, then two page transfers may be necessary, one out and one in. This increases further the page fault time!

- Use a *page replacement algorithm* to locate the page to be replaced. Desirably, the algorithm should result in minimum number of page faults

Page Replacement (Cont.)



1. Find a *victim* page using place replacement algorithm and write the victim page to the backing store (*page-out*)
2. Change the page table entry of the victim page to invalid
3. Bring the desired page into the newly freed frame (*page-in*)
4. Update the page table entry for the new page

When replacing pages, **only** *dirty pages* are actually written to backing store

Page Replacement (Cont.)

- This overhead can be minimized by not writing pages that have not been modified since they were brought into memory. Such pages are identified by using a *modify* (or *dirty*) bit
- When replacing pages, **only** *dirty pages* are actually written to backing store
- Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory

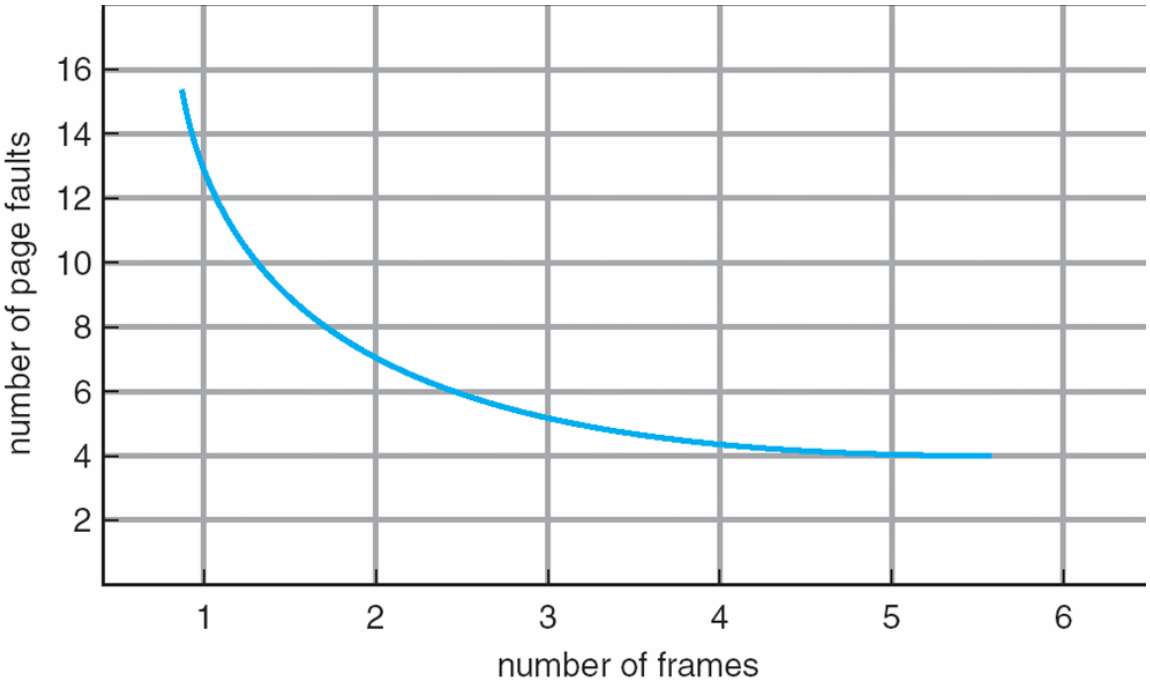
Page-Replacement Algorithms

- Objective is to have an algorithm with lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string), counting the number of page faults
- In all our examples, the ref. string used is

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

Page-Replacement Algorithms (Cont.)

- Obviously, as the # of available frames increases, the # of page faults will decrease:



Belady's Anomaly: more frames $\not\Rightarrow$ fewer page faults

Outline

- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- Second-Chance Algorithm (or Clock Algorithm)

First-In-First-Out (FIFO) Algorithm

- The easiest and simplest algorithm.

  Replace the page that has been *loaded* in the memory for the longest period of time

- Example using 3 free frames:

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	4	4	4	5	5	5	5	5	5
F2		2	2	2	1	1	1	1	1	3	3	3
F3			3	3	3	2	2	2	2	2	4	4
	P	P	P	P	P	P	P			P	P	

9 page faults

FIFO Algorithm (Cont.)

– Example using 4 free frames:

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	5	5	5	5	4	4
F2		2	2	2	2	2	2	1	1	1	1	5
F3			3	3	3	3	3	3	2	2	2	2
F4				4	4	4	4	4	4	3	3	3
	P	P	P	P			P	P	P	P	P	P

10 page faults !

- Belady's Anomaly
  - more frames $\not\Rightarrow$ fewer page faults

Optimal Algorithm

- Replace the page that will not be used for the longest period of time.

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	1	1	1	1	4	4
F2		2	2	2	2	2	2	2	2	2	2	2
F3			3	3	3	3	3	3	3	3	3	3
F4				4	4	4	5	5	5	5	5	5
	P	P	P	P			P				P	

6 page faults

- *Problem: How will the OS know the future references?*

Optimal Algorithm (Cont.)

- Theoretically best but practically infeasible
- Used as a benchmark to measure how well other algorithms perform
- Does not suffer from Belady's anomaly:

  Add a frame $\Rightarrow$ more look-ahead $\Rightarrow$

  Inclusion property : Pages loaded in *n* frames is always a subset of pages in *n+1* frames

Least Recently Used (LRU) Algorithm

- Replace the page that has not been used for the longest period of time

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	1	1	1	1	1	5
F2		2	2	2	2	2	2	2	2	2	2	2
F3			3	3	3	3	5	5	5	5	4	4
F4				4	4	4	4	4	4	3	3	3
	P	P	P	P			P			P	P	P

8 page faults

- Similar to Optimal Algorithm, but looking backward in time (and not susceptible to Belady's anomaly)

Implementation of LRU Algorithm

- Counter Implementation
  - Each page table entry has a time-of-use field
  - The CPU increments a logical clock for every memory reference
  - Whenever a page is referenced, the value of the clock is copied to the time-of-use field of the page
  - When a page fault occurs, the page with the smallest time-of-use value is kicked out

Implementation of LRU Algorithm (Cont.)

- Stack implementation
  - Keep a stack of page numbers in a doubly linked list
  - When a page is referenced:
    - move it to the top
    - the bottom of the stack is the LRU page
  - List update is expensive, but no search is needed for page replacement

Implementation of LRU Algorithm (Cont.)

- Stack Implementation Example

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

8 Page Faults

Stack of Page Numbers

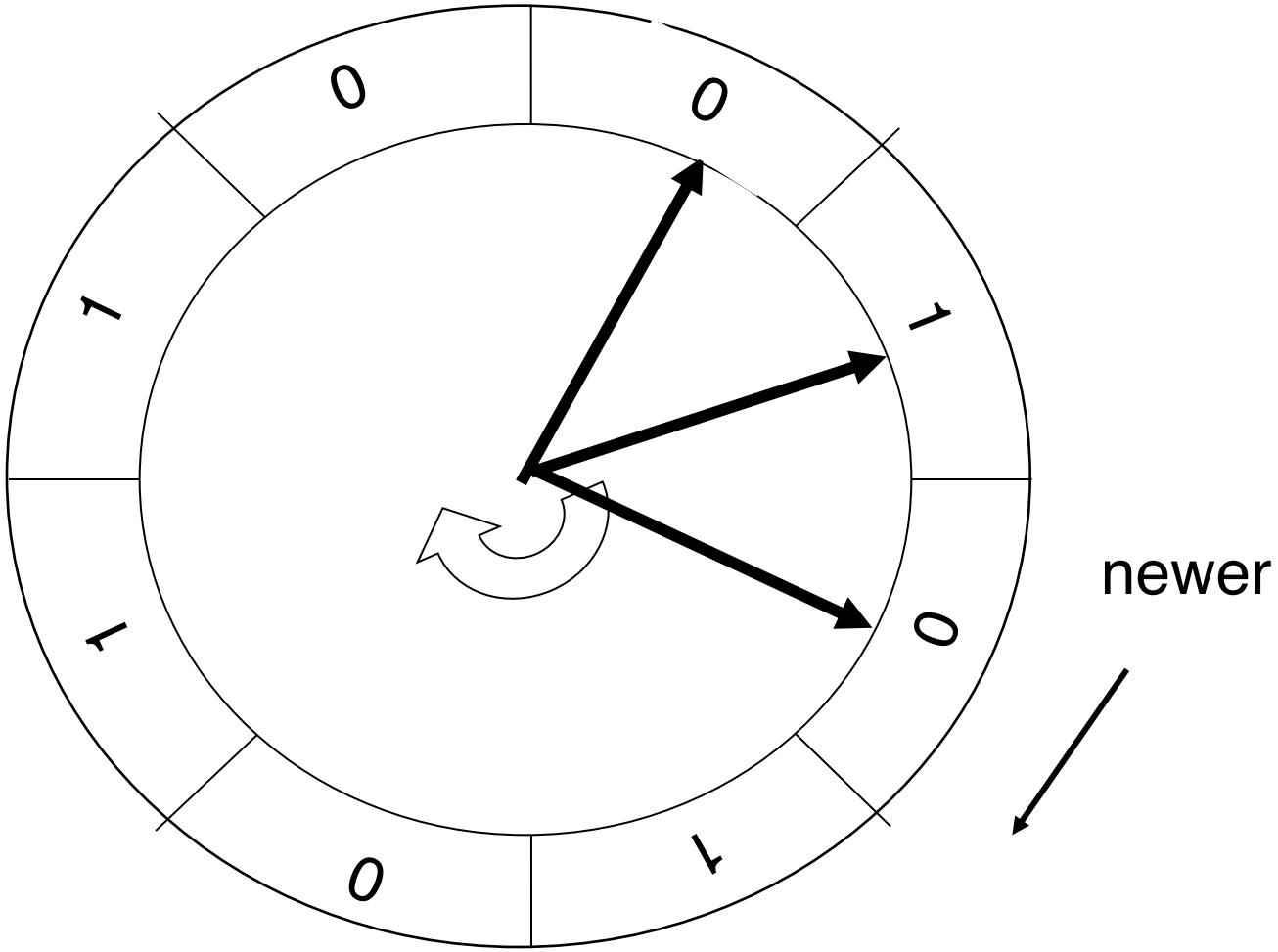
LRU Approximation Algorithms

- Hardware support necessary for exact algorithms are expensive and generally not available
- However, many systems support a hardware settable *reference bit*
  - With each page associate a bit *R*, initially = 0
  - When page is referenced, hardware sets its *R* to 1
  - Replace page with *R* = 0 (if one exists). Note that the exact order of use cannot be determined by *R*
    - ✳ There may be many pages with R = 0
- Approximation Algorithm:
  - Second-Chance (Clock) algorithm

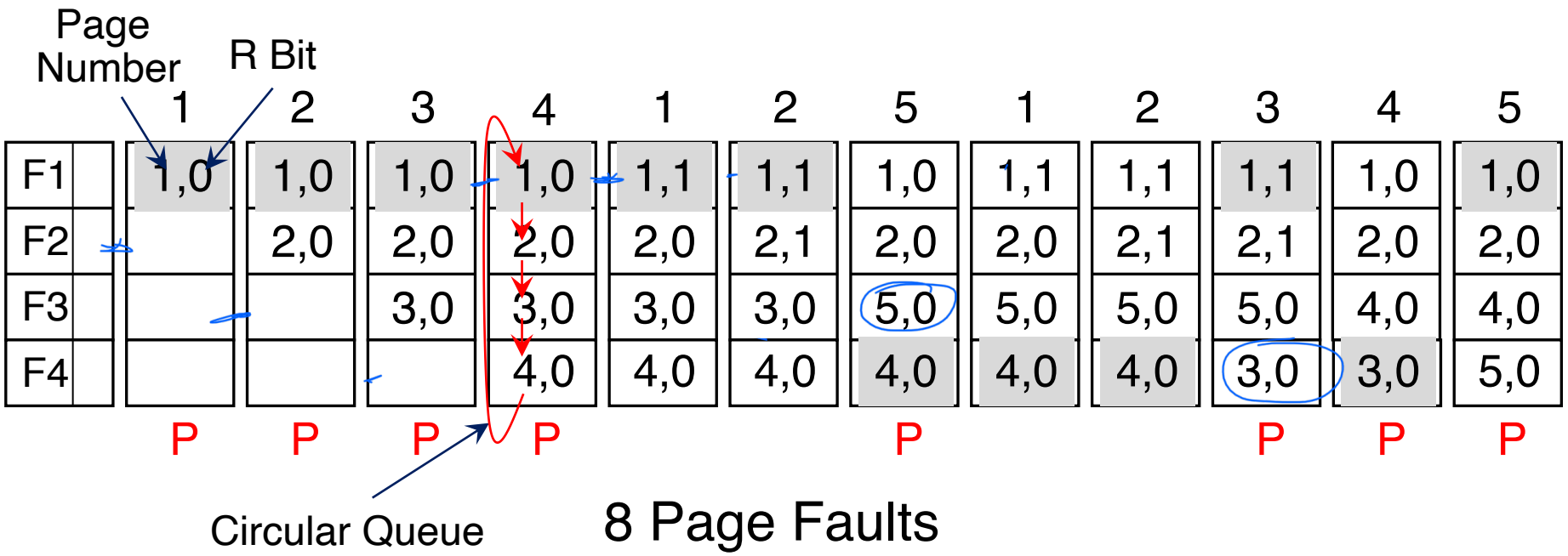
Second-Chance Algorithm

- A variation of FIFO algorithm
- Choose oldest page as candidate to be replaced
- If the reference bit *R* of the candidate is set (*R*=1), the page is given a second chance:
  - the *R* bit is cleared and the page is treated as if it has just been read in
- Otherwise, the page is kicked out

Clock Algorithm (Cont.)



Example: Clock Algorithm



The second digit of each entry is the *R* bit. The shaded entry is where the clock hand points to. It moves upwards. When reaching the top, it will go back to the bottom

Allocation of Frames

- The OS must decide how many pages to bring into physical memory
  - the smaller the amount of memory allocated to each process, the more processes can reside in memory
  - small number of pages loaded increases page faults
  - beyond a certain size, further allocations of pages will not effect the page fault rate

Fixed vs. Variable Allocation

- *Fixed Allocation*
  - gives a process a fixed number of frames in physical memory within which to execute
    - e.g., equal allocation or proportional allocation
  - when a page fault occurs, one of the pages of that process must be replaced
- *Variable Allocation*
  - allows the number of page frames allocated to a process to be varied over the lifetime of the process

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 process

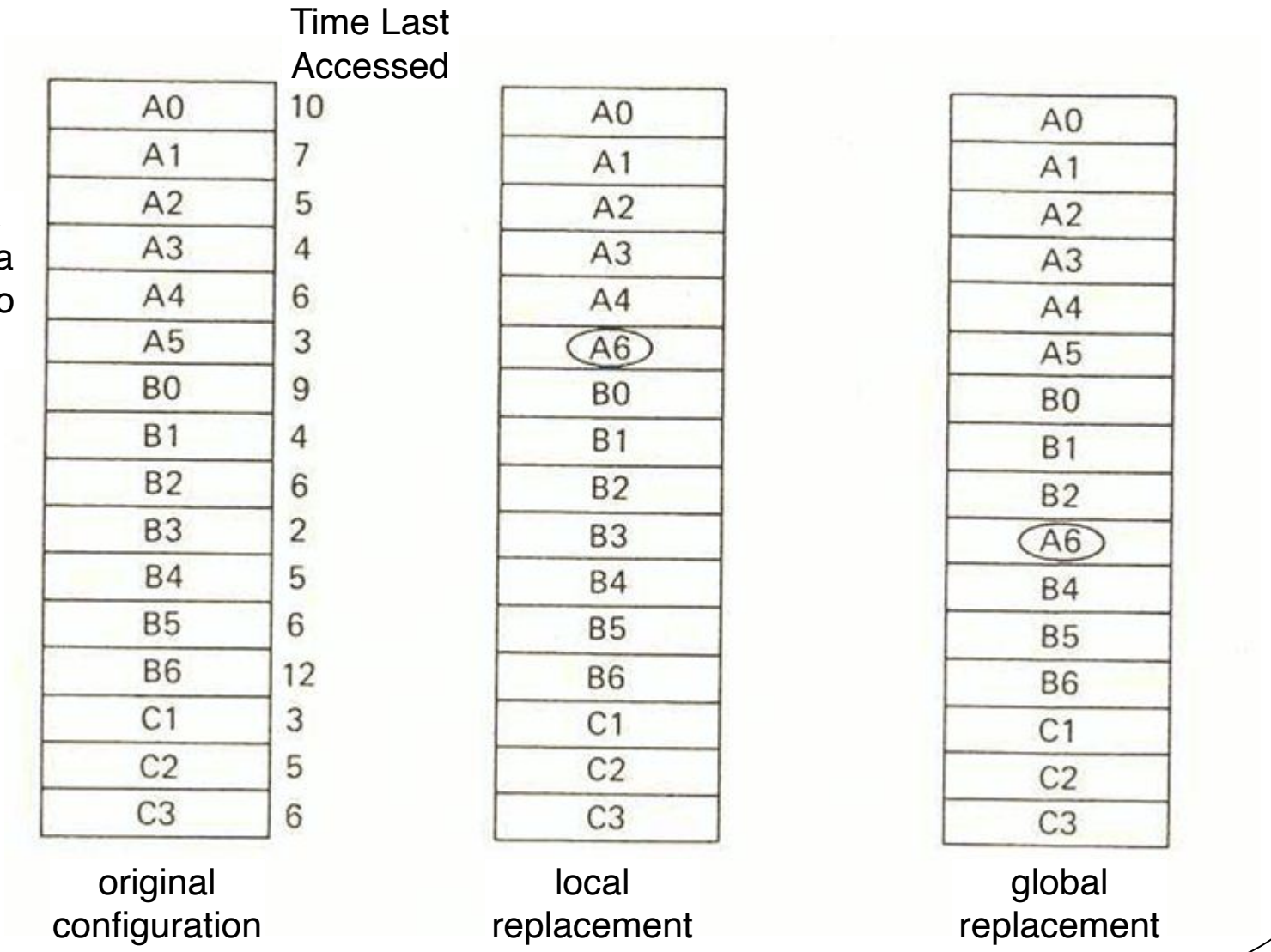
  $\Rightarrow$ Performance depends on external circumstances (other processes)

- *Local replacement* – each process selects only from its own set of allocated frame

  $\Rightarrow$ may hinder other processes by not making available its less used pages/frames

Global vs. Local Allocation (Cont.)

Process A generates a reference to page A6



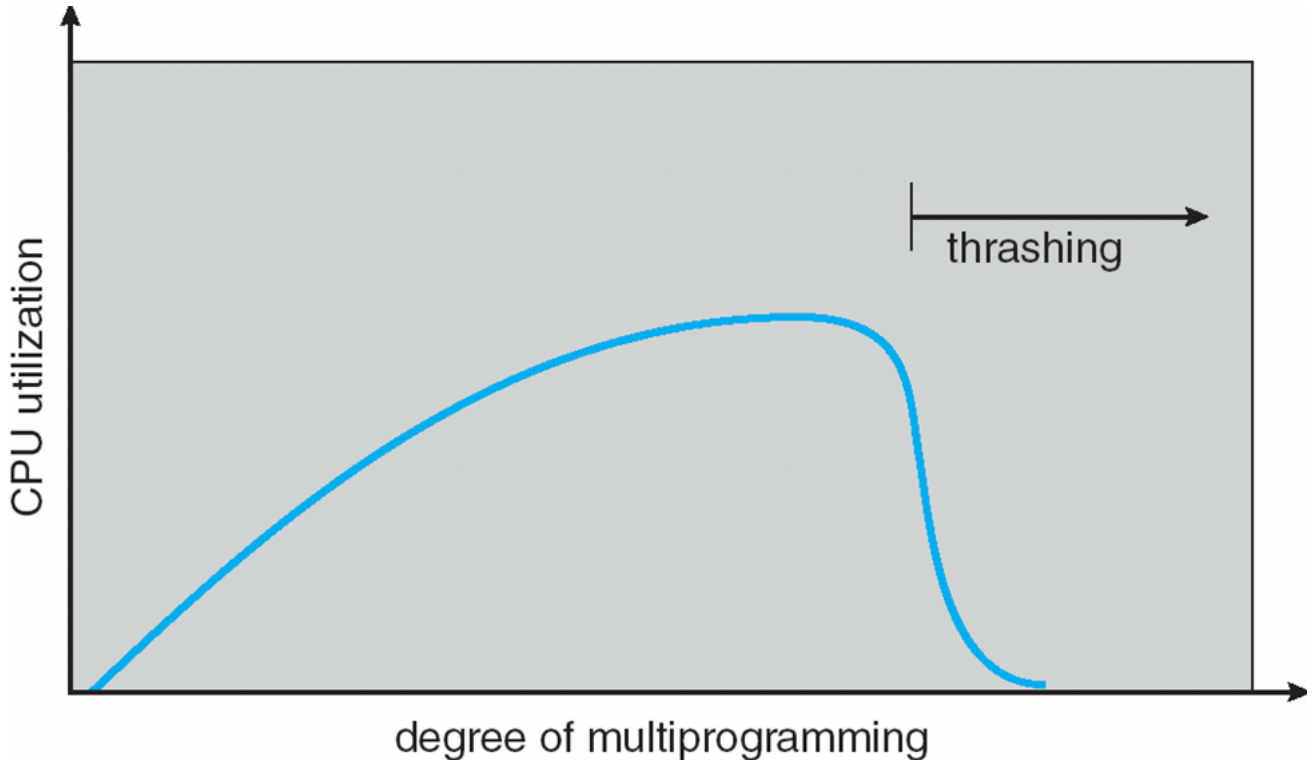
Allocation of Frames – Summary

	Local Replacement	Global Replacement
Fixed Allocation	•Number of frames allocated to a process is fixed. •Page to be replaced is chosen from among the frames allocated to that process.	•Not possible.
Variable Allocation	•The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. •Page to be replaced is chosen from among the frames allocated to that process.	•Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high.
  - This leads to low CPU utilization
- A common wisdom to improve CPU utilization is to increase the degree of multiprogramming
  - Another process is added to the system
  - CPU utilization drops further

Thrashing (Cont.)



- *Thrashing* ≡ a process is busy bringing pages in and out (no work is being done)

Strategies to combat Thrashing

- Two approaches are used:
  - Working-Set Model
  - Page-Fault Frequency

Working-Set Model

- *Locality of Reference*: Processes tend to refer to pages in a localised manner
  - *Temporal Locality*: locations referred to just before are likely to be referred to again (i.e., location references are clustered over time)
  - *Spatial Locality*: code & data are usually clustered physically
- This observation led to the Working-Set Model

Working-Set Model

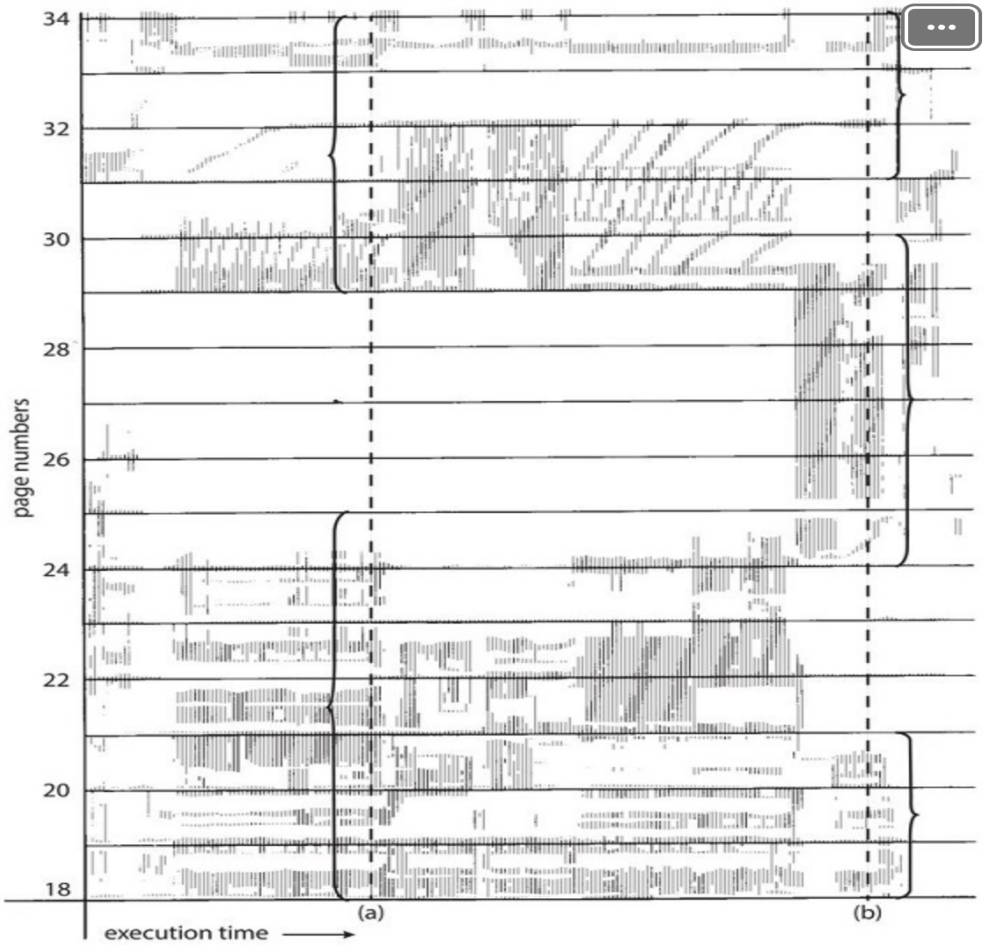
Locality Example

```
sum = 0;
for (I =0; I < n; i++)
  sum += a[i];
return sum;
```

- Data
  - Temporal: `sum` referenced in each iteration
  - Spatial: elements of array `a[]` accessed in consecutive manner
- Instructions
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

Working-Set Model (Cont.)

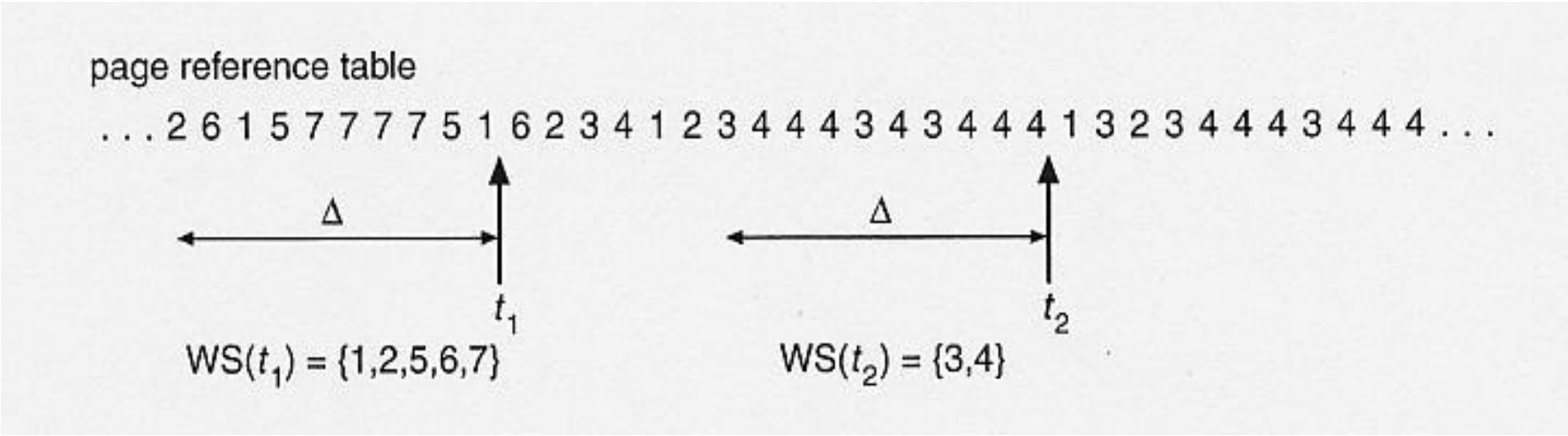
- During the lifetime of the process, references are confined to a subset of pages



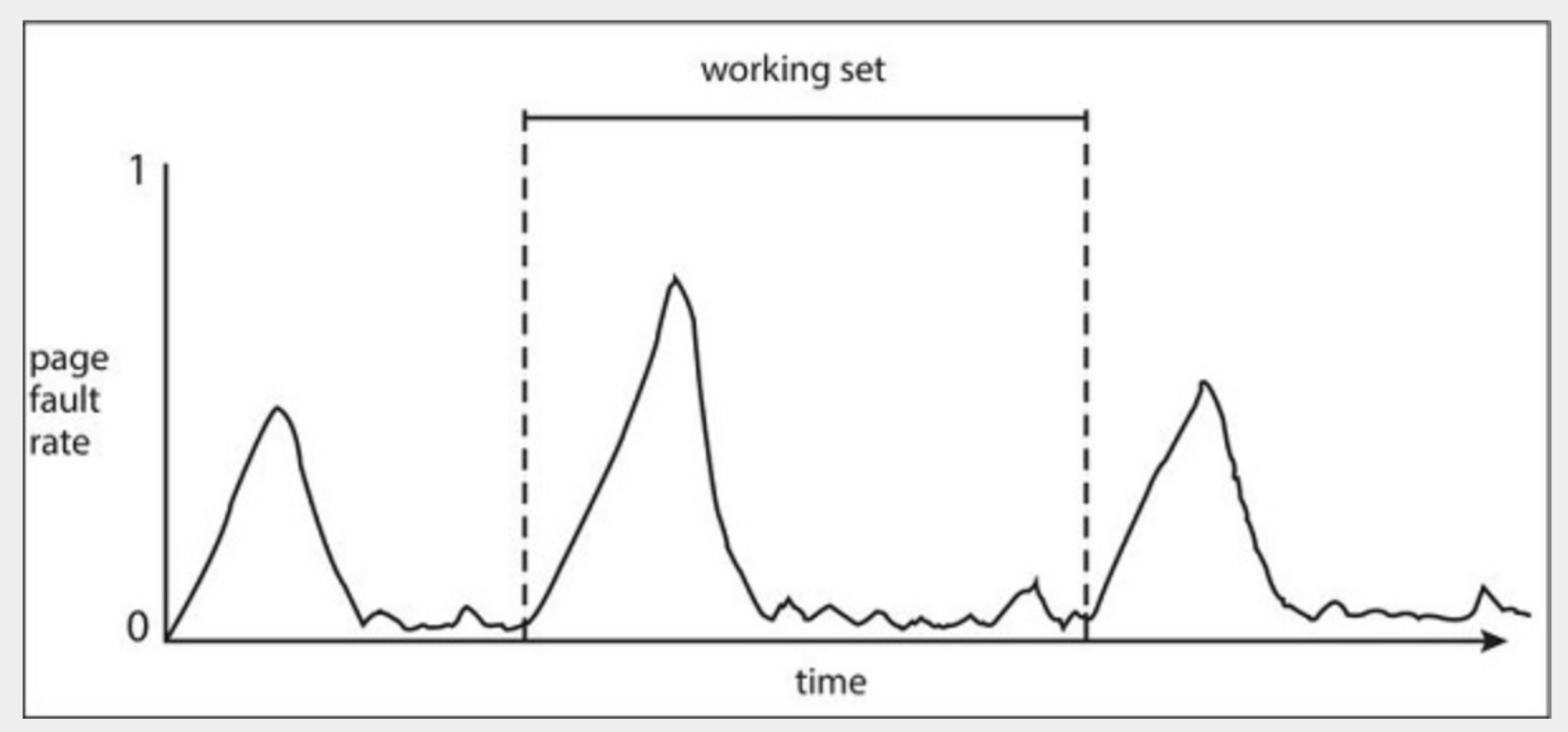
Working-Set Model (Cont.)

- $\Delta \equiv$ *working-set window* $\equiv$ a fixed number of page references

  For example: if Δ= 10 memory references, then the working set at t_1 is {1,2,5,6,7}, at t_2 is {3,4}



Working-Set Model (Cont.)

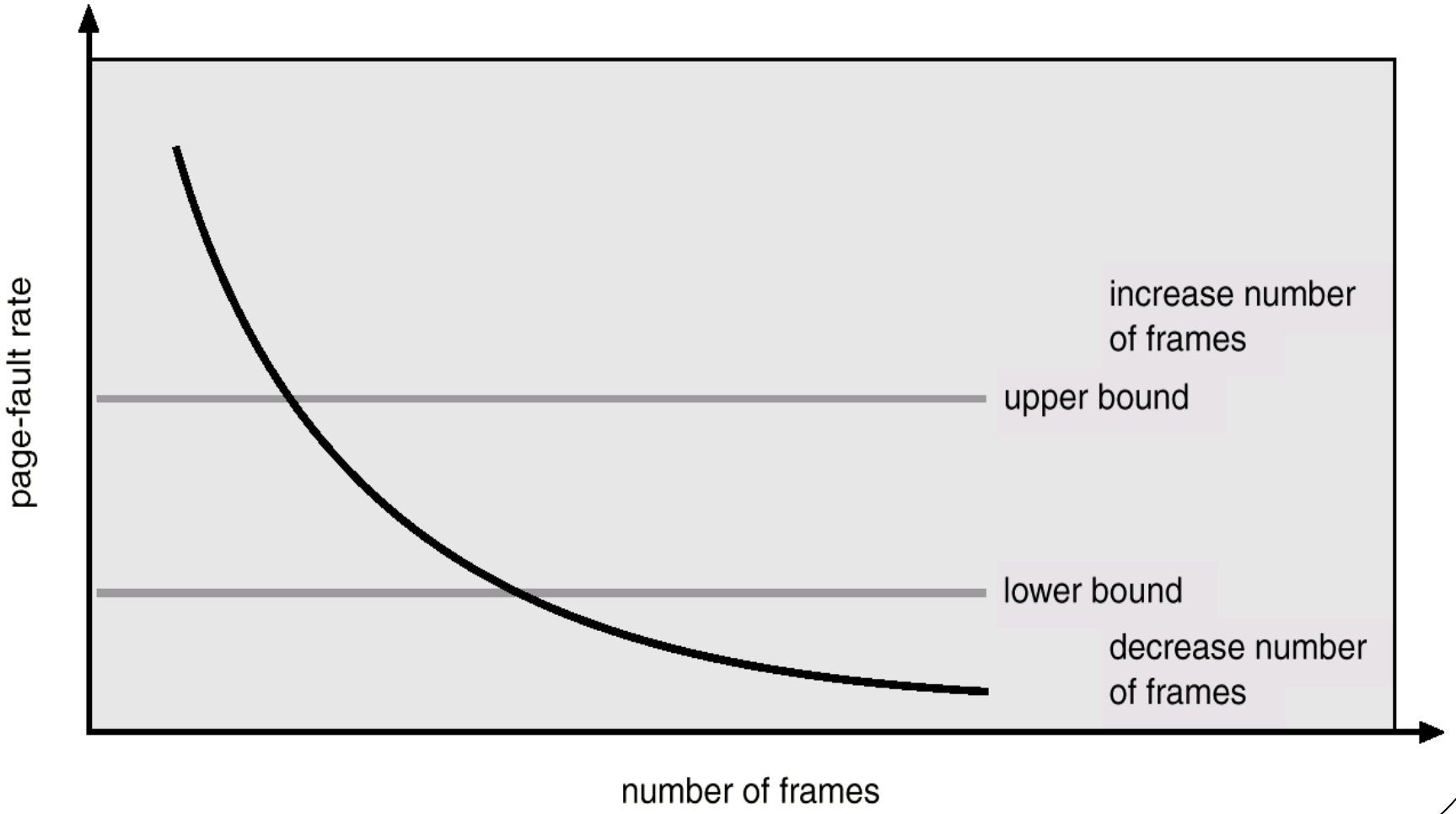


Working-Set Model

- WSS_i (working set size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- $D = \Sigma\ WSS_i \equiv$ total demand for frames

  m = total # of frames
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D > m$, then suspend (i.e., swap out) one of the processes

Page-Fault Frequency Scheme



Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate of a process is too low, remove a frame from that process
  - If actual rate too high, give that process a frame
- May have to suspend (i.e., swap out) a process if the page fault rate increases and no free frames are available

Summary

- Virtual Memory
  - To allow a large logical address space to be mapped onto a smaller physical memory
  - Demand paging is a way to implement virtual memory
- Page Replacement Algorithms
  - FIFO, Optimal, LRU, LRU Approximation
- Frame Allocation
  - Fixed vs. Dynamic Allocation
  - Local vs. Global Allocation

Summary (Cont.)

- Thrashing: process is busy bringing pages in and out (no work is being done)
  - Working-set Model
  - Page-Fault Frequency Scheme