Chapter 9 Virtual Memory

LI Wensheng, SCS, BUPT

Strong point:

Demand Paging

Page Replacement

Thrashing

Contents

9.1 Background

9.2 Demand Paging

9.3 Copy-on-write

9.4 Page Replacement

9.5 Allocation of Frames

9.6 Thrashing

9.7 Memory-mapped file

9.8 Allocating Kernel Memory

9.9 Other considerations

9.10 Operating-System Examples

Objectives

- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.
- To discuss the principle of the working-set model.

9.1 Background

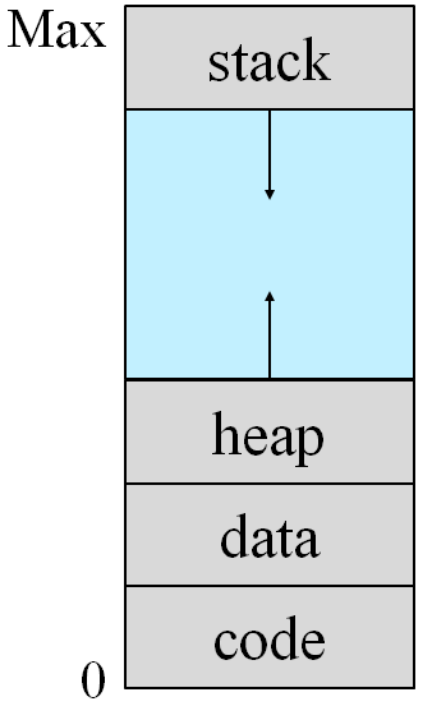
- Code needs to be in memory to execute, but entire program rarely used.
  - Error code, unusual routines, large data structures.
- Entire program code not needed at same time.
- Consider ability to execute partially-loaded program.
  - Program no longer constrained by limits of physical memory.
  - 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.

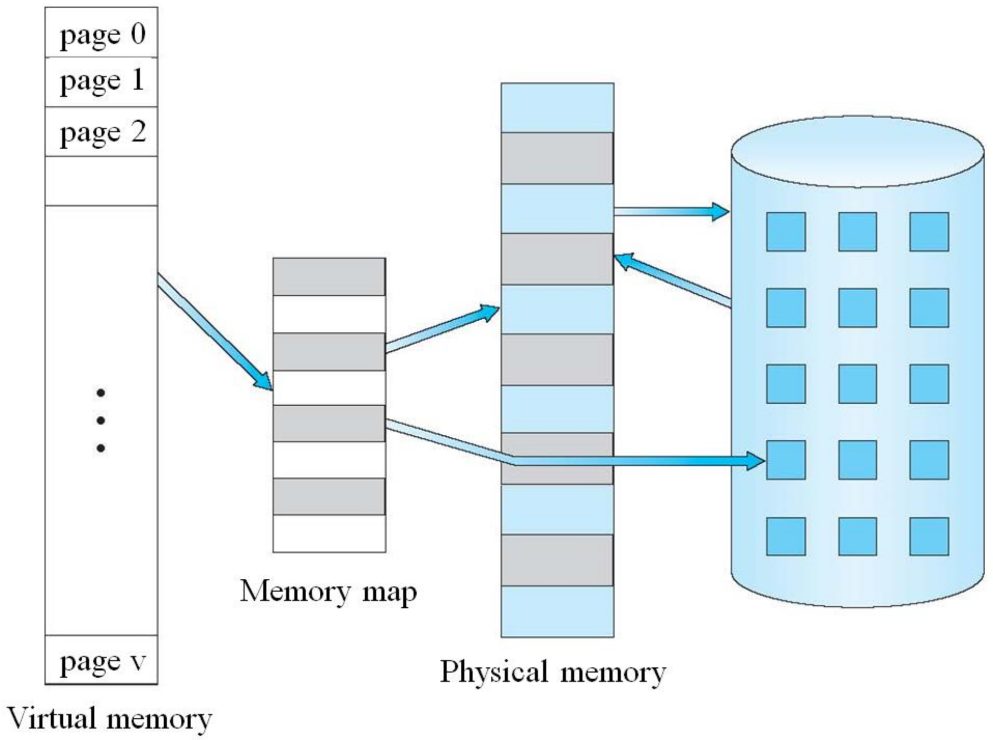
Background(Cont.)

- **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 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.**

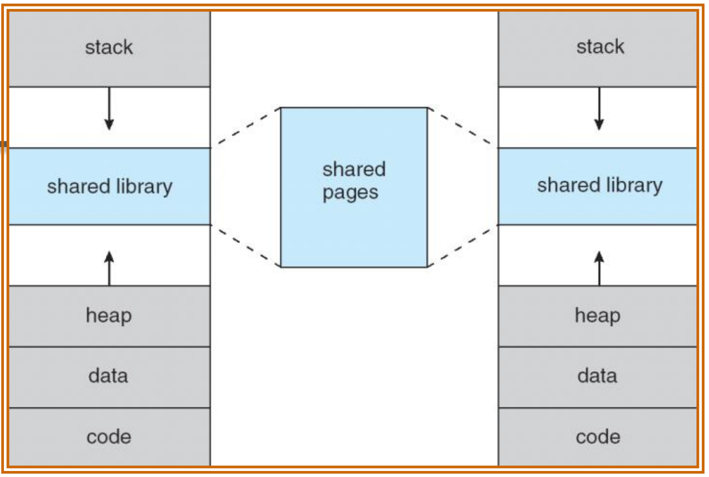
Virtual-address Space

- logical view of how process is stored in memory.
  - Start at address 0, contiguous addresses until end of space.
  - Physical memory organized in page frames.
  - MMU must map logical to physical.
- Can be larger than physical memory.





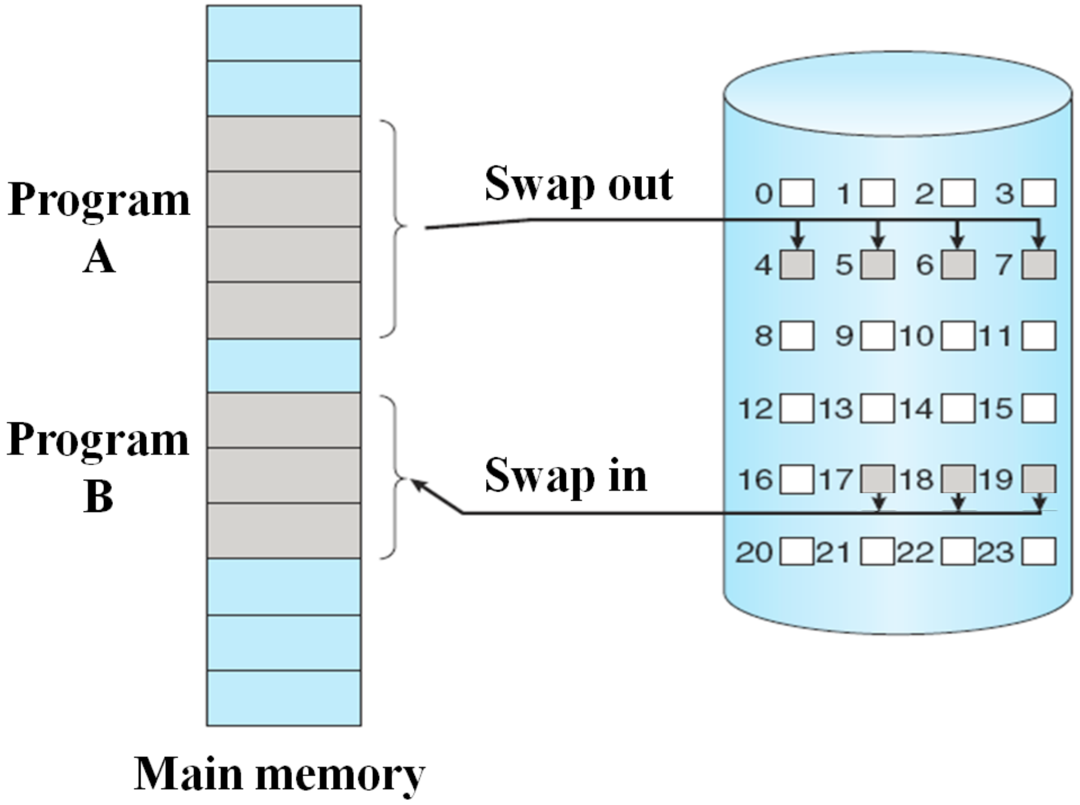
Virtual-address Space(Cont.)



- Usually, stack space starts 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 (稀疏) address spaces with holes left for growth, dynamically linked libraries, etc.
- Virtual memory allows files and memory to be shared by two or more processes through page sharing.
  - 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 up process creation

Background(Cont.)

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
- a paging system with swapping



End 1

9.2 Demand Paging

按需分配，空间共享

- Bring a page into memory only when it is needed.
- Lazy swapper: never swaps a page into memory unless that page will be needed.
- Swapper manipulates entire processes.
  Pager deals with the individual pages of a process.
- Page is needed ⇒ reference to it.
- If pages needed are already memory resident
  - No difference from non demand-paging.
- If pages needed and not memory resident
  - Need to detect
    - Invalid reference ⇒ abort.
    - Valid but not-in-memory ⇒ bring it into memory.
  - Without changing program behavior.
  - Without programmer needing to change code.

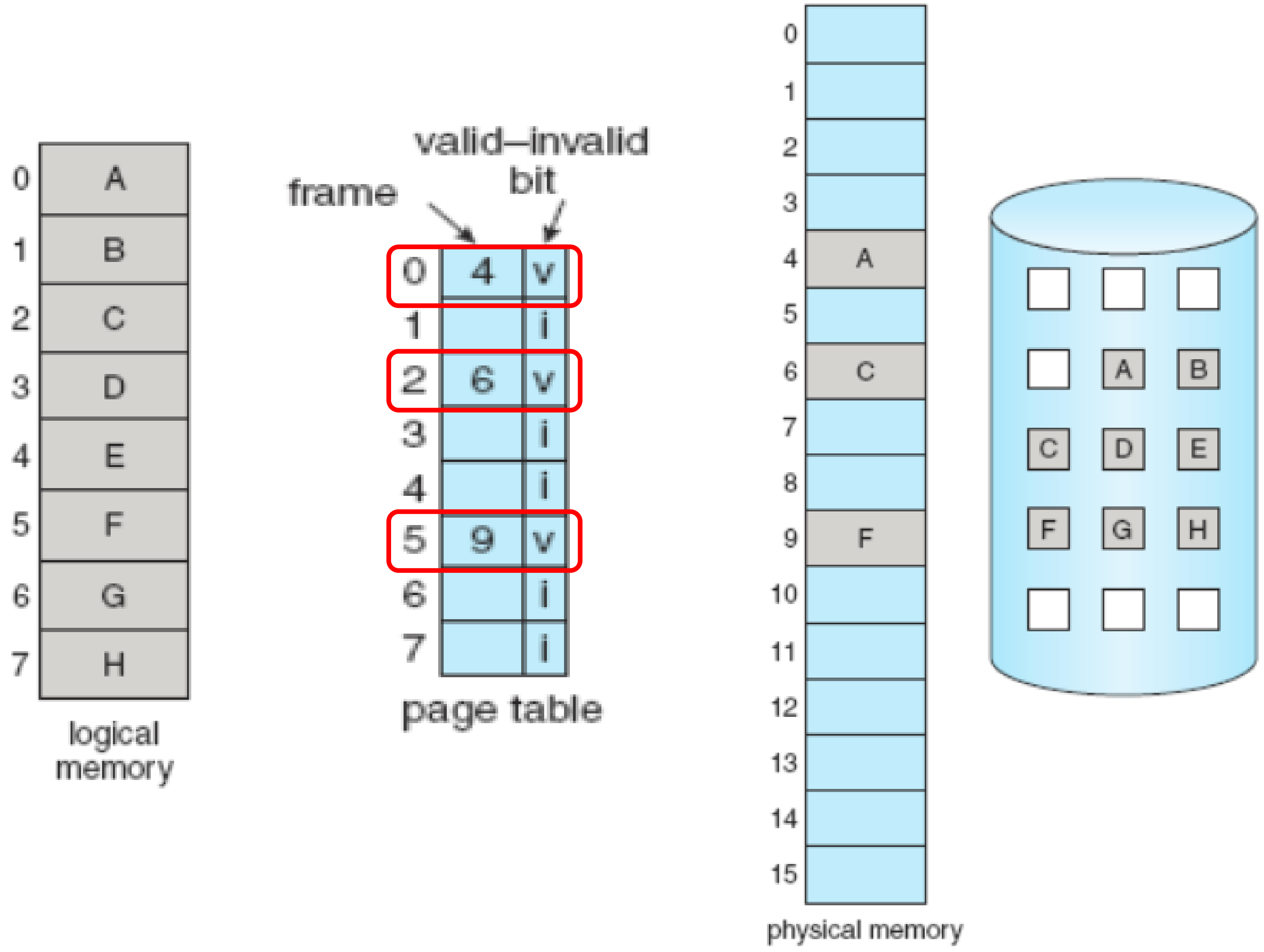
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated.
  (v ⇒ valid & in-memory,
  i ⇒ invalid or
  valid but not-in-memory)
- Initially valid–invalid bit is set to "i" on all entries.
- When a page is loaded into a frame, the frame# is written into the page-table entry, and set the Valid-Invalid bit to "v".
- During address translation, if valid–invalid bit in page table entry is i ⇒ **page fault**（缺页）.

	Frame#	valid-invalid bit
0		v
1		v
2		i
3		v
4		i
5		v
6		i
7		i

Page table

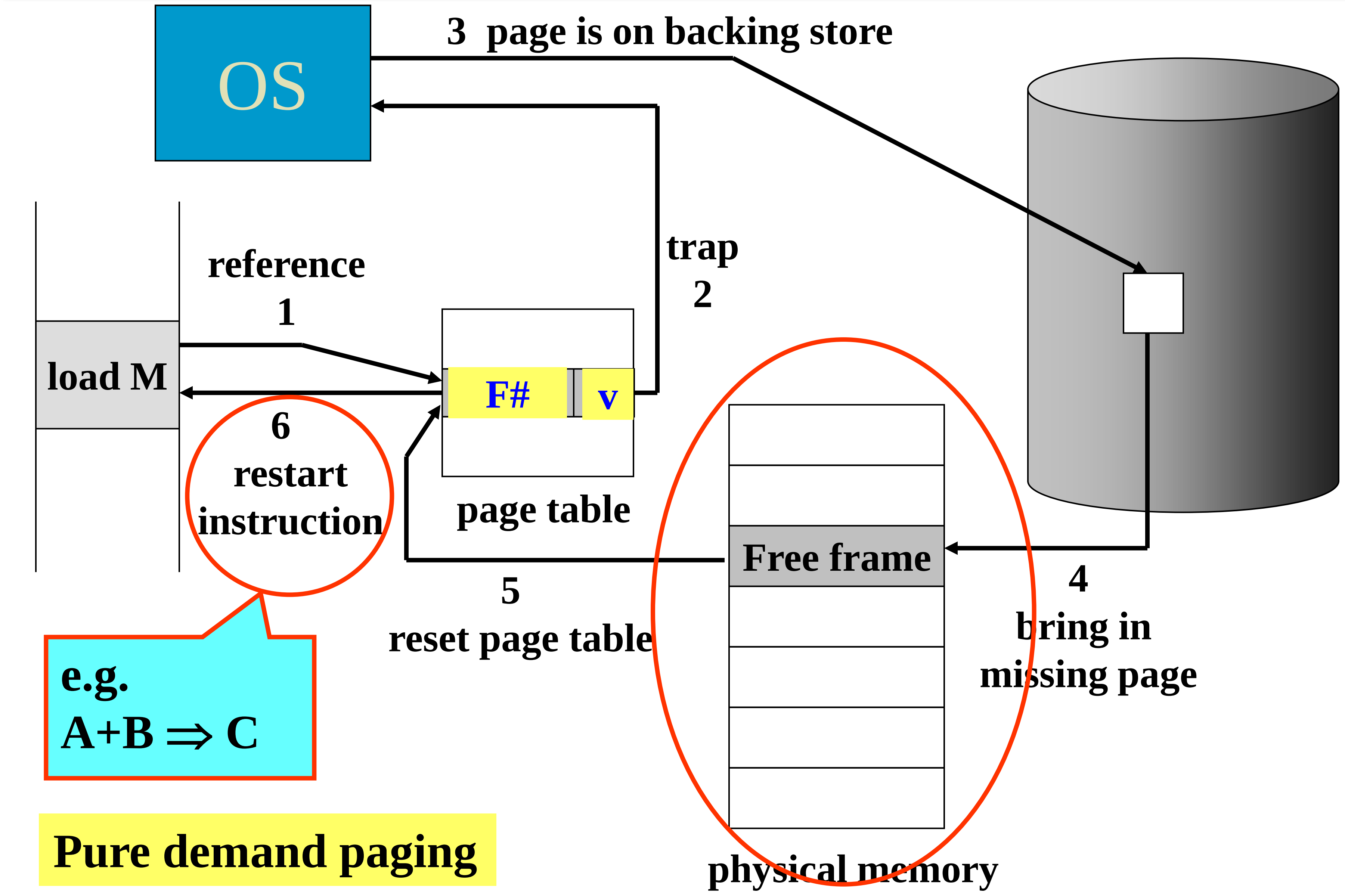
Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault

- If there is a reference to a page, first reference to that page will trap to OS ⇒ page fault.
- OS looks at another internal table (kept with PCB) to decide:
  - Invalid reference ⇒ abort.
  - Valid, just not in memory ⇒ page it in.
- Find a free frame.
- Read page into frame via scheduled disk operation.
- Reset page table, writes frame#, Set validation bit=v.
- Restart the instruction that caused the page fault.

Steps in Handling a Page Fault (Cont.)



e.g.
A+B ⇒ C

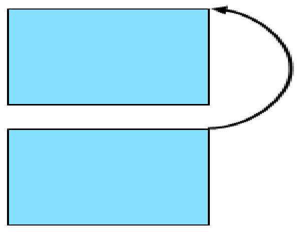
Pure demand paging

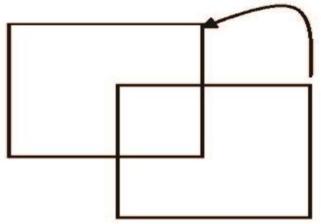
Aspects of Demand Paging

- Pure demand paging, an extreme case, start process with *no* pages in memory.
  - OS sets instruction pointer to the first instruction of process, non-memory-resident ⇒ page fault.
  - And for every other process pages on first access.
- 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

- A crucial requirement is the ability to restart any instruction after a page fault.
- Instruction A+B ==> C
  - Fetch and decode the instruction, fetch A, fetch B, Add a and B, Store the sum in C
- Instruction that could access several different locations. (e.g. IBM 360/370 MVC)
  - block move (up to 256 bytes).
  - auto increment/decrement location
  - If either block straddles a page boundary, a page fault might occur after the move is partially done.
  - Restart the whole operation?
  - What if source and destination overlap?





What happens if there is no free frame?

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

Stages in Demand Paging (worse case)

1. Trap to the OS.
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 the location of the 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. Ready, 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

- 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 = $(1 - p) \times$ memory access time
  $+ \, p \times$ page fault service time
- Three major activities
  1. Service the page fault interrupt, careful coding means just several hundred instructions needed.
  2. Read in the page, lots of time.
  3. Restart the process, again just a small amount of time.
- page fault service time = page fault overhead
  + [swap page out]
  + swap page in
  + restart overhead

1~100 μs

8ms=5+3+0.05

Performance of Demand Paging(Cont.)

- average page-fault service time ≈ 8 milliseconds
  - servicing the page-fault interrupt and restarting the process may be reduced to several hundred instructions, and each may take from 1 to 100 microseconds.
  - The page-switch time will probably be close to 8 milliseconds
    - hard disk has a seek of 5 milliseconds;
    - Hard disk has an average latency of 3 milliseconds;
    - transfer time of 0.05 milliseconds.

Demand Paging Example

- Memory access time = 200 nanoseconds
- Effective Access Time

  EAT = (1 – p) × 200 + p × (8 000 000)

  = 200 + 7 999 800 × p
- If one access out of 1,000 causes a page fault,

  P=0.001

  EAT=8199.8ns

  Slowed down by a factor of 40
- If want performance degradation < 10 percent

  200 + 7,999,800 * p < 220
  7,999,800 * p < 20

  p < .0000025

  < 1 page fault in every 400,000 memory accesses.

Exercise 1

- **On a system using paging, references to a swapped-in locations accessible through an entry in an associative table take 150ns.
If the main memory page table must be used, the reference takes 400ns.
References that result in page faults require 8ms if the page to be replaced has been modified, 3ms otherwise.
If the page fault rate is 2%, the associative table(TLB) hit rate is 70%, and 50% of replaced pages have been modified, what is the effective access time?
Assume the system is running only a single process and the CPU is idle during page swaps.**

Answer for exercise 1

1s=1000ms
1ms=1000μs
1μs=1000ns
1ns=1000ps

EAT=

Exercise 2

- Assume that we have a demand-paged memory. The page table is held in registers.
  It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified.
  Memory-access time is 100 nanoseconds.
  Assume that the page to be replaced is modified 30 percent of the time.
  What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

Answer for exercise 2

1s=1000ms

1ms=1000μs

1μs=1000ns

1ns=1000ps

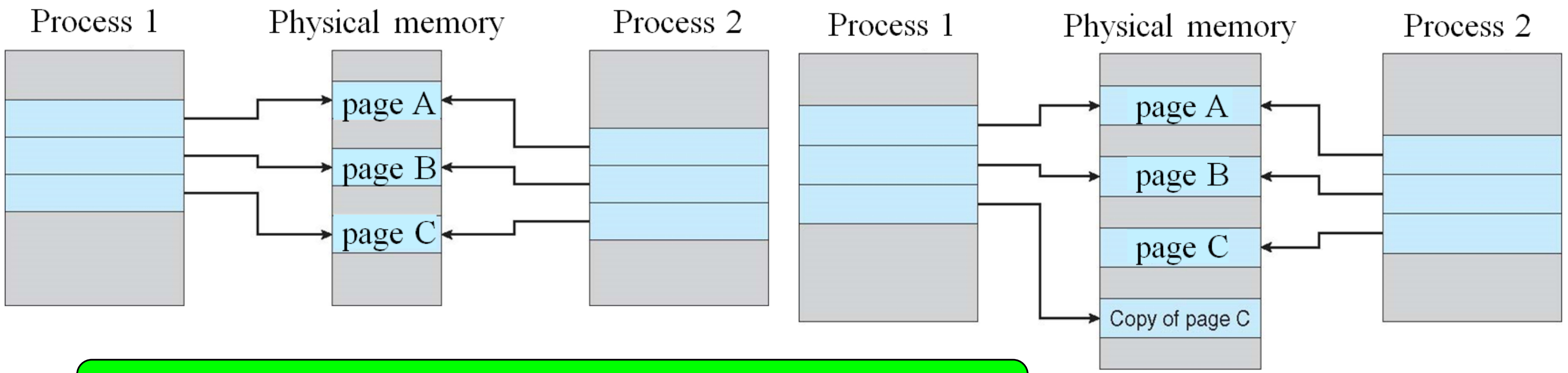
Exercise 3

- Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)
  - TLB miss with no page fault
  - TLB miss and page fault
  - TLB hit and no page fault
  - TLB hit and page fault

End 2

9.3 Copy-on-Write—process creation

- Fork()
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.
  - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- only pages that can be modified need be marked as copy-on-write.



空间共享，提高效率，提高资源利用率

COW(Cont.)

- In general, free pages are allocated from a pool of free pages.
  - Pool should always have free frames for fast demand page execution.
  - Using a technique known as zero-fill-on-demand. zero-out a page before allocating it. Erasing the previous contents.
- vfork(), does not use copy-on-write.
  has parent suspend and child using the address space of parent..
  - If the child changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.
  - Designed to have child call exec() immediately.
  - Very efficient method of process creation.
  - Sometimes used to implement UNIX command-line shell interface.

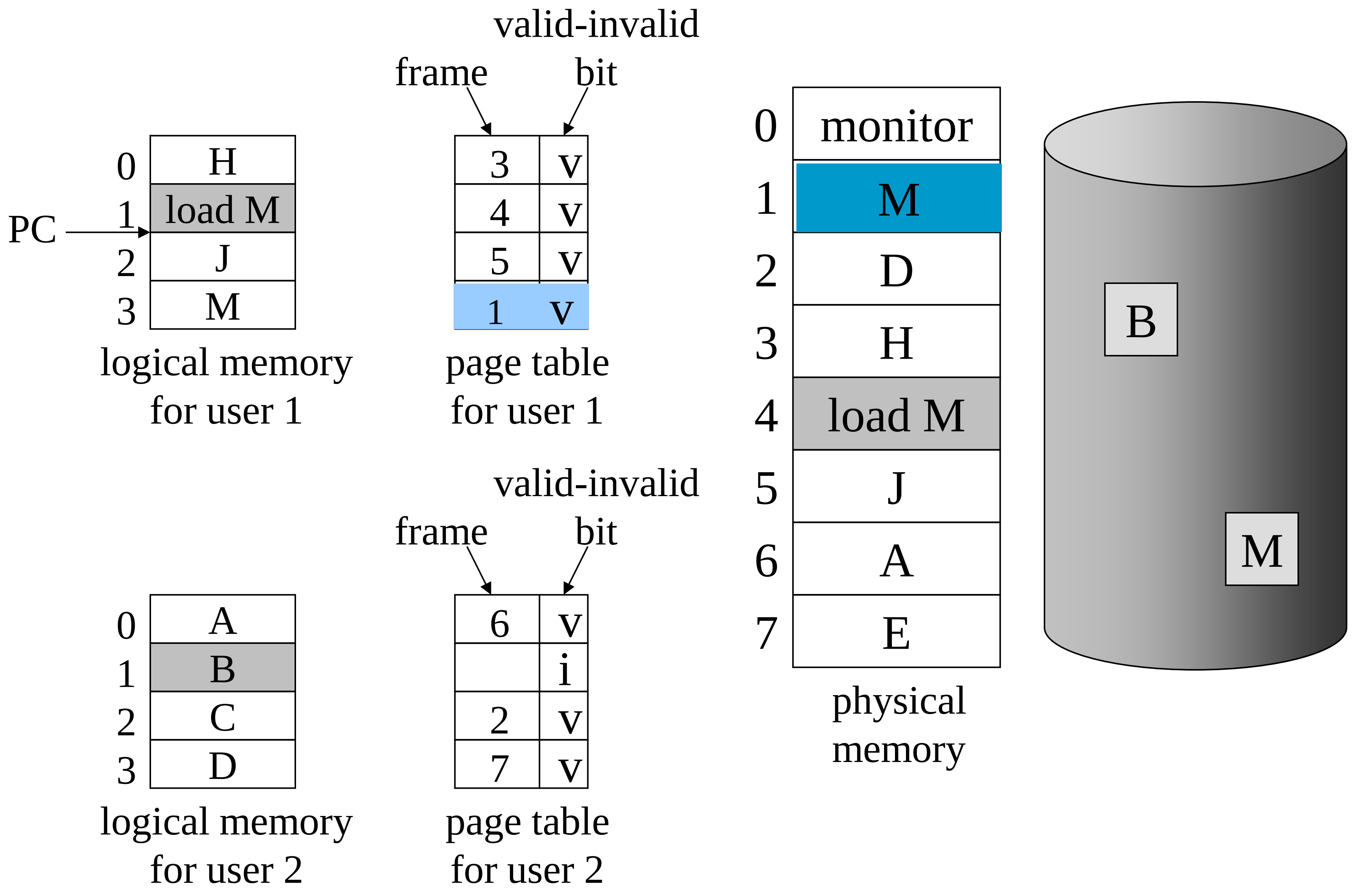
End 3

9.4 Page Replacement

帧复用共享，提高效率
提高资源利用率

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
  - Allocate a fixed percentage of memory for I/O buffers.
  - Allow both user processes and I/O subsystem to compete for all memory.
- What happens if there is no free frame?
  - Terminate the user process. (not the best choice)
  - Swap processes?
  - Page replacement, demand paging, transparent to the user.

Need For Page Replacement



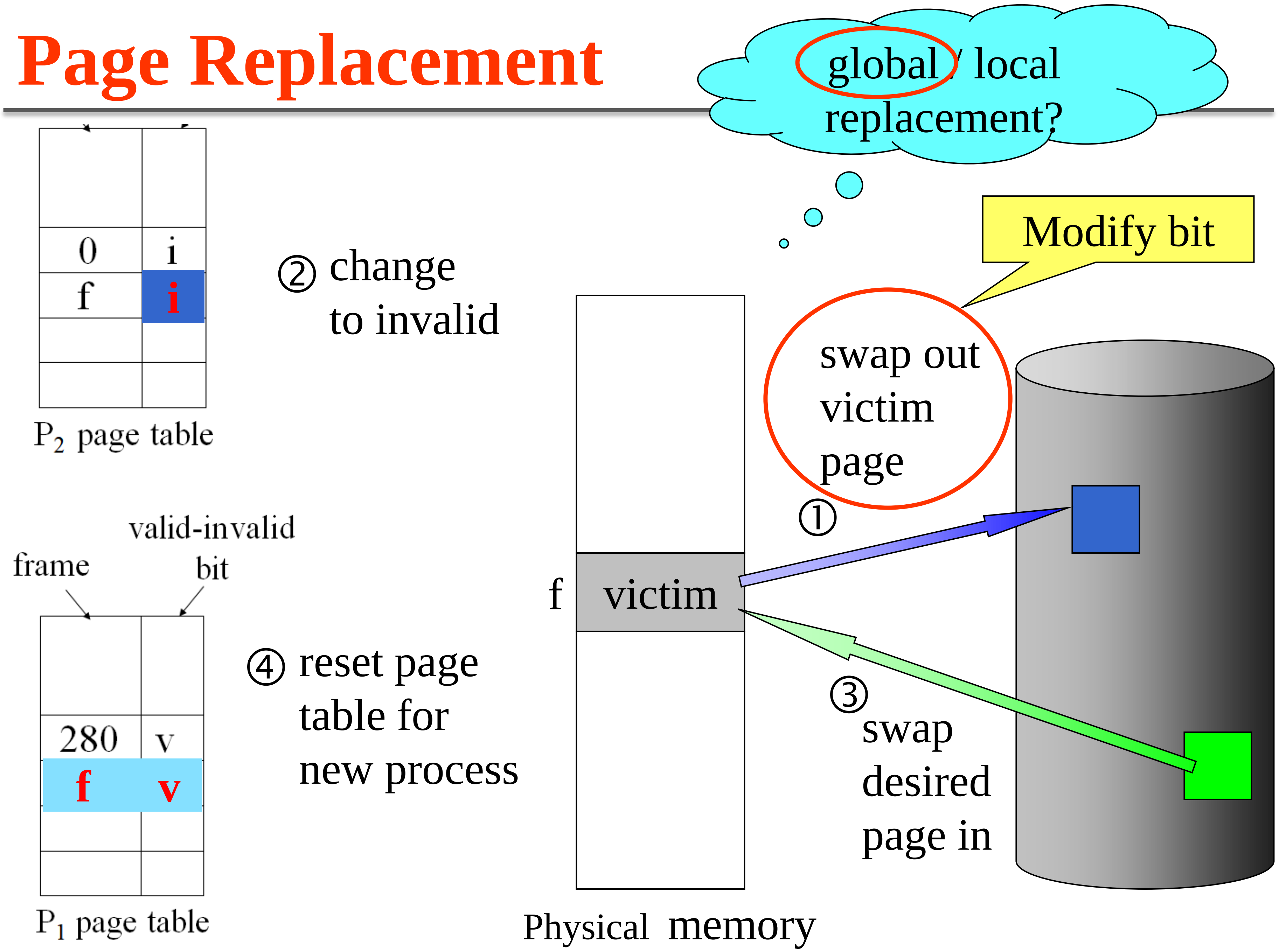
Basic Page Replacement

- Find the location of the desired page on disk.
- 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 the disk if dirty; change the page and frame tables accordingly.
- Read the desired page into the (newly) freed frame. Update the page and frame tables.
- Continue the process by restarting the instruction that caused the trap.
- Note now potentially 2 page transfers (1 out and 1 in) for page fault – increasing EAT.

Basic Page Replacement(Cont.)

- Use *modify bit (dirty bit)* to reduce overhead of page transfers.
  - modify bit is needed to indicate if the page has been altered since it was last loaded into main memory.
  - If no change has been made, the page does not have to be written to the disk when it needs to be swapped out.
  - 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.
  - With no demand paging, all the pages of a process still must be in physical memory.

Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process?
  - Which frames to replace, when page fault?
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access.
  - Page removed should be the page least likely to be referenced in the near future.
- want an algorithm which will result in minimum number of page faults.

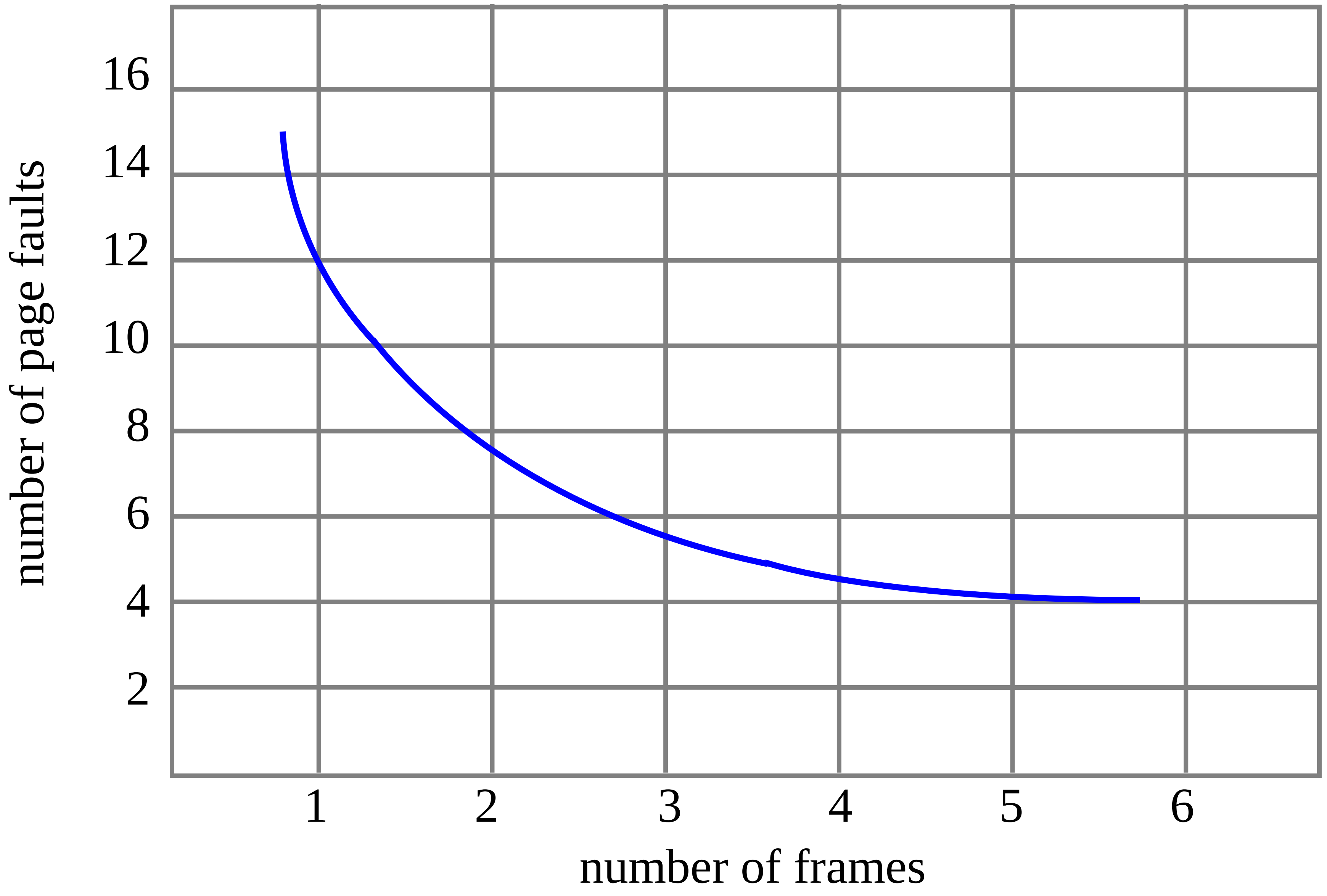
Page and Frame Replacement Algorithms(Cont.)

- **Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.**
  - **String is just page numbers, not full addresses.**
  - **Repeated access to the same page does not cause a page fault.**
  - **Results depend on the number of frames available.**

Page Replacement Algorithms

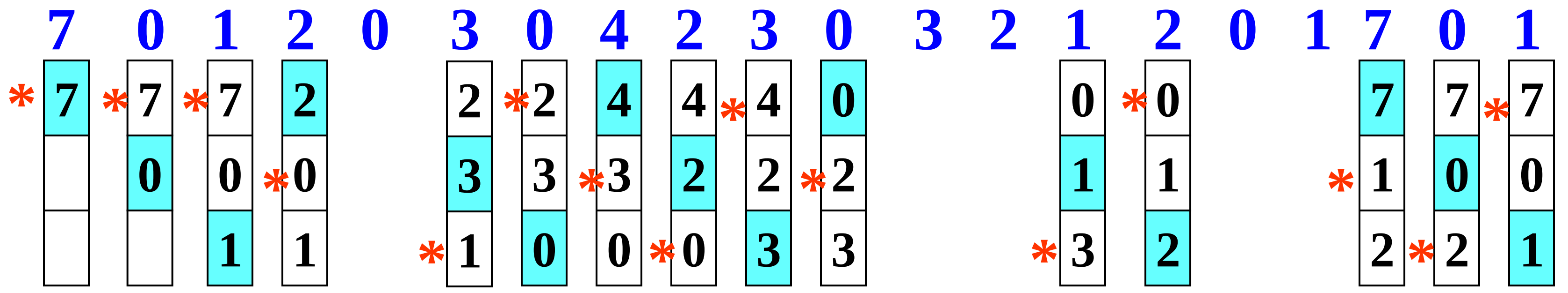
- We can generate reference strings
  - artificially by a random-number generator, or
  - by tracing a given system and record the address of each memory reference.
- E.g. record the following address sequence:
  - 0100, 0423, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
  - Page size is 100B, reduced to the following reference string:
    1, 4, 1, 6, $\boxed{1, 1, 1, 1,}$ 6, $\boxed{1, 1, 1, 1,}$ 6, $\boxed{1, 1, 1, 1,}$ 6, 1, 1
    1, 4, 1, 6, $\boxed{1,}$ 6, $\boxed{1,}$ 6, $\boxed{1,}$ 6, 1
- In all our examples, the reference strings are
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



FIFO (First-In-First-Out) Algorithm

- Associates with each page the time when it was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- Example
  - 3 frames (3 pages can be in memory at a time per process)



- How to track ages of pages?
  - Just use a FIFO queue to hold all pages in memory.
  - Treats frames allocated to a process as a circular buffer.
  - Pages are removed in round-robin style.
- The page replaced may be needed again very soon.

Belady's Anomaly

more frames ⇒ more page faults

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

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

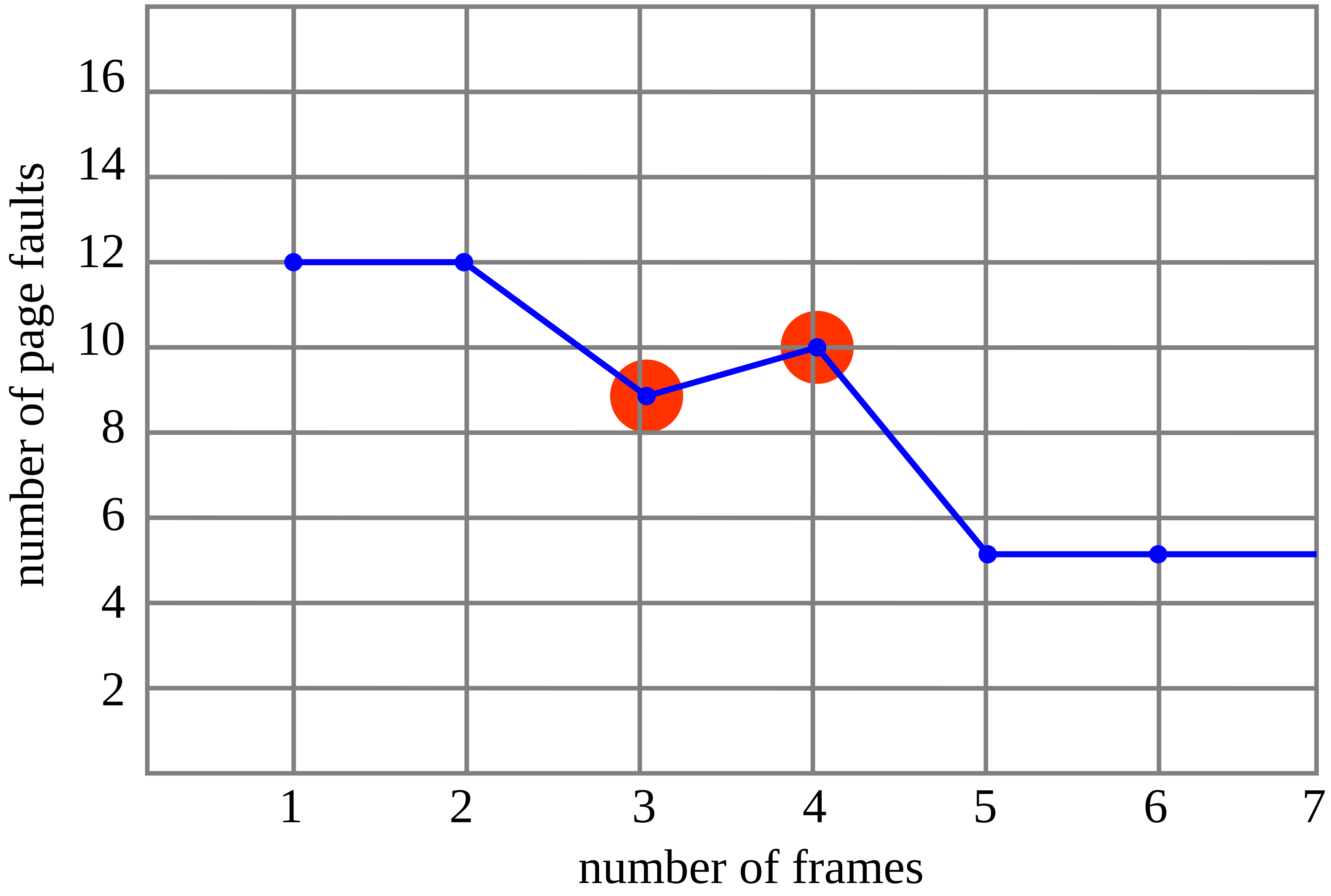
9 times page fault
Page fault rate:
9/12=75%

- 4 frames

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

10 tmes page fault
Page fault rate:
10/12=83.3%

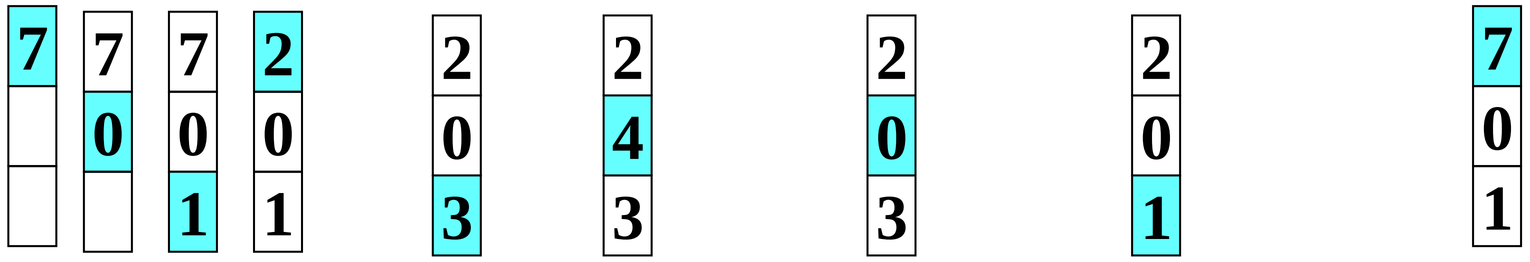
FIFO Illustrating Belady's Anamoly



Optimal Algorithm

- Replace the page that will not be used for the **longest period** of time.
- 3 frames example

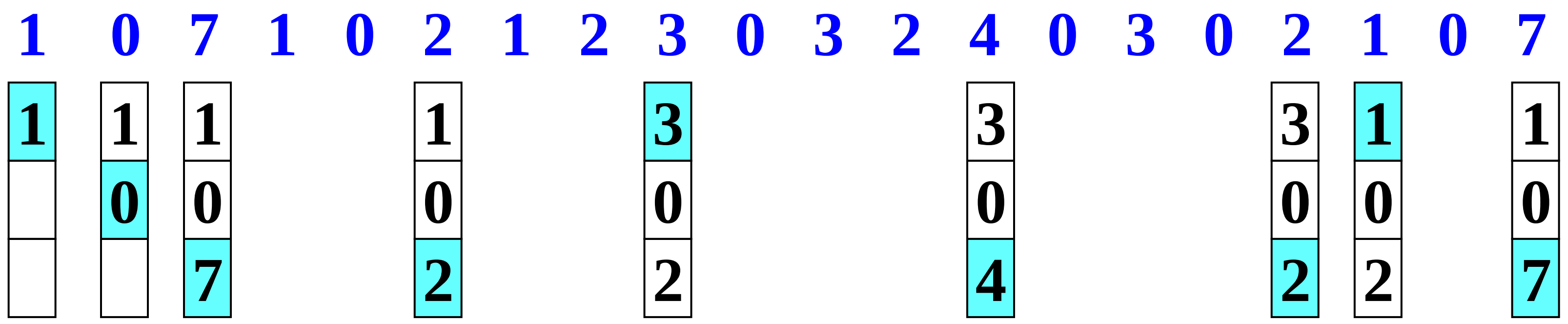
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



- Only 9 page faults
- **OPT** (or MIN) has the **lowest** page-fault rate of all algorithms, and **never** suffer from Belady's anomaly.

OPT Algorithm(Cont.)

- for the OPT algorithm， the page-fault rate on *S* is the same as the page-fault rate on S^R.



- Requires future knowledge of the reference string.
- Thinking about:
  - How do you know the reference string?
- Impossible to have perfect knowledge of future events.
- Used mainly for comparison studies
  - measures how well your algorithm performs.

OPT Algorithm(Cont.)

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

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

7 times page fault
Page fault rate:
7/12=58.3%

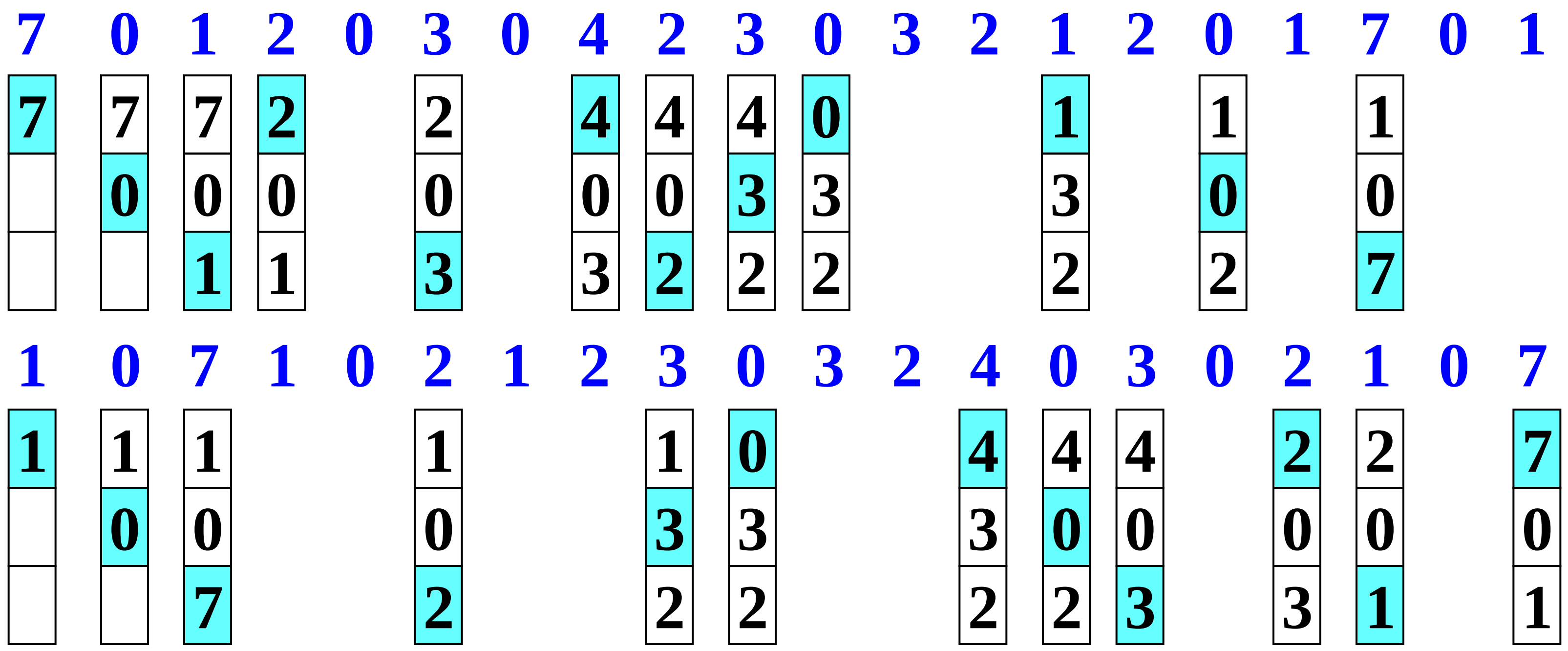
- **4 frames**

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

6 times page fault
Page fault rate:
6/12=50%

Least Recently Used (LRU) Algorithm

- replaces the page that has not been used for the longest period of time.
- By the principle of locality, this should be the page least likely to be referenced in the near future.
- Associates with each page the time of that page's last use. This would require a great deal of overhead.
- Example



LRU Algorithm example

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

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

10 times page fault
Page fault rate:
10/12=83.3%

- **4 frames**

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

8 times page fault
Page fault rate :
8/12=66.7%

Replacement Algorithms Summary

- The FIFO algorithm uses the time when a page was brought into memory.
  - The oldest page is replaced.
- The OPT algorithm uses the time when a page is to be used.
  - The page that will not be used for the longest period of time is replaced.
- the LRU algorithm uses the time when a page was referenced last.
  - The page that has not been used for the longest period of time is replaced.

Exercise 4

Consider the following page reference string:

1, 2, 4, 5, 3, 4, 1, 6, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

How many page faults would occur for the following replacement algorithms.

Assume there are three frames available, and all frames are initially empty.

(1) Optimal replacement

(2) LRU replacement

(3) FIFO replacement

Answer for exercise 4

OPT: **Page fault:** **times,** **page fault rate:**

LRU: **Page fault:** **times,** **page fault rate:**

Answer for exercise 4 (Cont.)

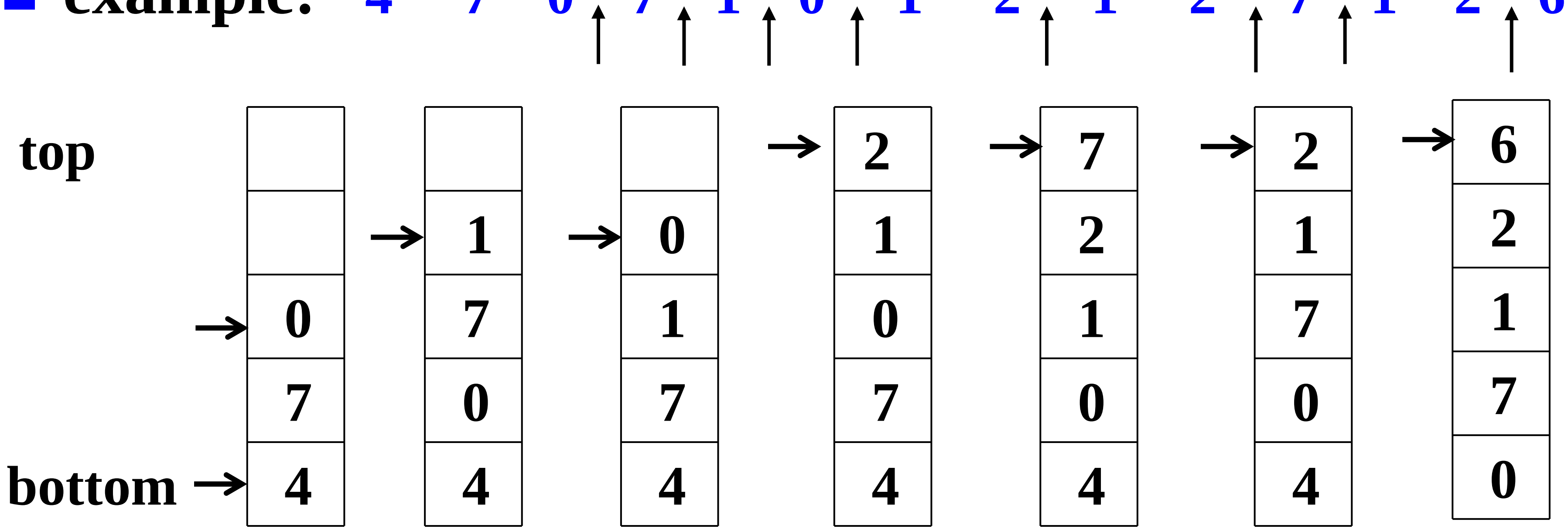
FIFO: Page fault: times, page fault rate:

LRU implementation

- may require substantial hardware assistance.
- Problem -- determine an order for the frames defined by the time of last use.
- Counters
  - Every page entry has a counter, a time-of-use field; add to the CPU a logical clock or counter. The clock is incremented for every memory reference.
  - Every time a page is referenced through this entry, copy the contents of the clock into the time-of-use field in the page-table for that page.
  - When a page needs to be replaced, look at the counters to determine which are to be replaced.
    - Replace the page with the smallest time value.

LRU implementation (Cont.)

- Stack
  - keep a stack of page numbers in a double link form.
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed, at worst.
  - No search for replacement, LRU page is at the bottom.
- example: 4 7 0 7 1 0 1 2 1 2 7 1 2 6



Exercise 5

Page	Frame number	Valid/ invalid	Page in time	Reference time	Reference bit	Modify bit
0	60	1	225	326	1	1
1	35	0	100	105	1	
2	50	0	138	245	1	1
3	35	1	289	321	1	1
4		0				
5	50	1	312	387	1	0
6		0				
7	35	0	268	280	1	0

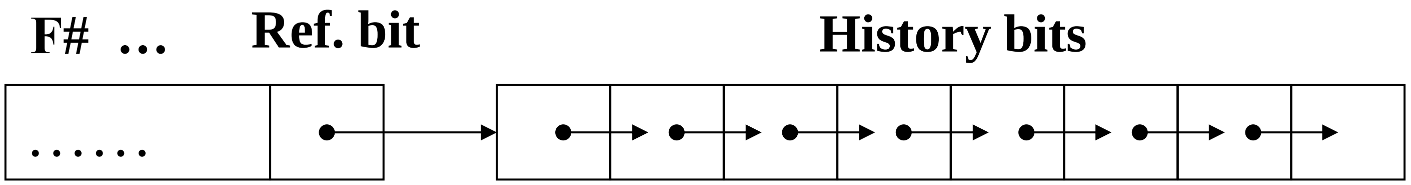
- **A process has 8 pages, its page table is shown as above, assume three frames are allocated to this process.**
- **Now page 6 is needed.**
- **FIFO replacement algorithm is used, which frame will the page be paged into?**
- **LRU replacement algorithm is used, which frame will the page be paged into?**

LRU Approximation Algorithms

- **LRU needs special hardware and still slow.**
- **Reference bit algorithm**
  - **With each page entry associate a bit, initially set to 0.**
  - **When page is referenced, the bit is set to 1 by the hardware.**
  - **Replace the one which is 0 (if one exists).**
  - **Problem： We do not know the order.**

LRU Approximation Algorithms(Cont.)

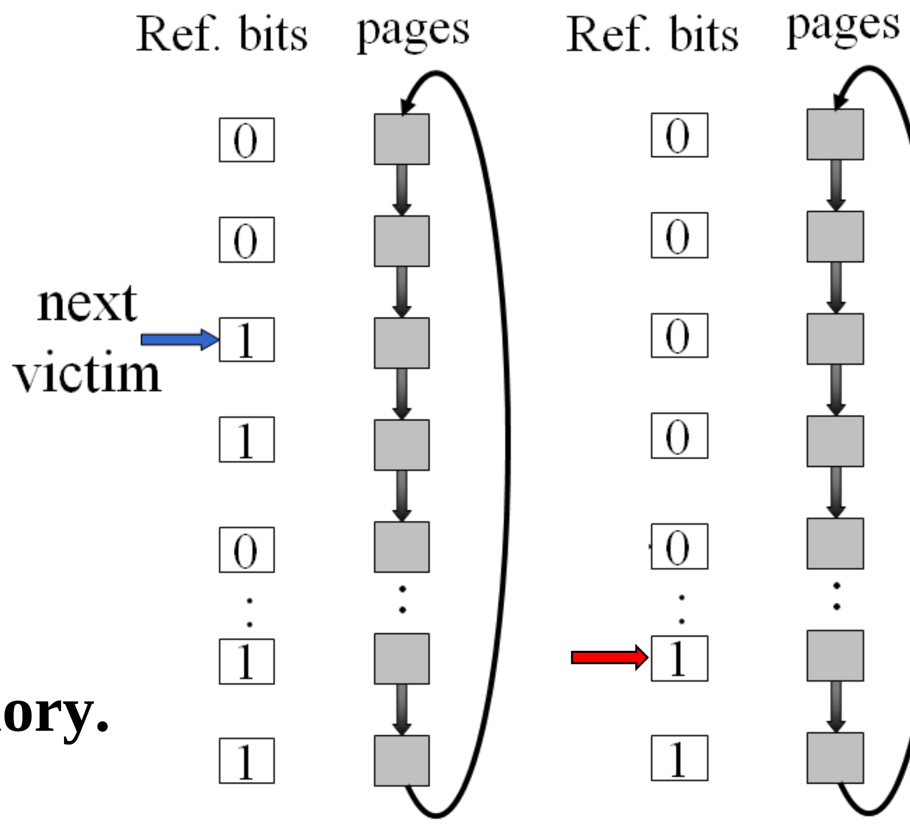
- **Additional-reference-bits algorithm**
  - Keep an 8-bit byte for each page in a table in memory.
  - At regular intervals, a timer interrupts, OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.



  - These 8-bit byte contains the history of page use for the last 8 time periods.
  - The page with the lowest number is the LRU page, can be replaced.
  - 1 00000000 / 0 11111111 ?

LRU Approximation Algorithms(Cont.)

- **Second chance**
  - **Basic algorithm: FIFO replacement algorithm.**
  - **Need reference bit**
  - **Clock replacement**
  - **If page to be replaced (in clock order) has reference bit = 1. then:**
    - **set its reference bit 0, its arrival time to the current time.**
    - **leave the page in memory.**
    - **replace next page (in clock order), subject to same rules.**



LRU Approximation Algorithms(Cont.)

- **Enhanced Second-Chance Algorithm**
  - **Considering both the reference bit and the modify bit as an ordered pair (reference bit, modify bit).**
  - **Each page is in one of four classes:**
    - **(0,0) neither recently used nor modified -- best page to replace**
    - **(0,1) not recently used but modified -- the page will need to be written out before replacement.**
    - **(1,0) recently used but clean -- probably will be used again soon.**
    - **(1,1) recently used and modified -- probably will be used again soon, and the page will be need to be written out to disk before replacement.**
  - **Examine the class to which that page belongs.**
  - **Replace the first page encountered in the lowest non-empty class.**
  - **Might need to search circular queue several times.**

LRU Approximation Algorithms(Cont.)

- Counting-based page replacement
  - Keep a counter of the number of references that have been made to each page.
  - LFU Algorithm: Least Frequently Used
    - replaces page with the smallest count.
    - shift the counts right by 1 bit at regular intervals.
  - MFU Algorithm: Most Frequently Used
    - based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
  - neither MPU nor LFU replacement is common.

Page-Buffering Algorithm

- Keep a pool of free frames.
  - When a page fault occurs,
    - A victim frame is chosen as before.
    - The desired page is read into a free frame from the pool before the victim is written out.
  - When the victim is later written out, its frame is added to the free-frame pool.
- Possibly, maintain a list of modified pages.
  - Whenever the paging device is idle, a modified page is selected and is written to the disk.
  - Its modify bit is then reset.
- Possibly, keep a pool of free frames, but to remember which page was in each frame.
  - The old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

End 4

9.5 Allocation of Frames

- Each process needs **minimum** number of pages.
- Example:
  - All memory reference instructions have only one memory address, needs at least 2 frames per process.
  - If one-level indirect addressing is allowed, requires at least 3 frames per process.
  - Move instruction, PDP-11, needs 6 pages
    - instruction itself may straddle 2 pages.
    - each of its two operands may be indirect references.
  - MVC instruction, IBM 370, needs 6 pages:
    - instruction is 6 bytes, might straddle 2 pages.
    - 2 pages to handle *from* block.
    - 2 pages to handle *to* block.
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 93 frames and 5 processes, give each 18 frames, 3 leftover frames used as a free-frame buffer pool.
- Proportional allocation – Allocate available memory to each process according to its size.
  - s_i: size of virtual memory for process p_i
  - $S=\sum s_i$
  - m: total number of frames
  - a_i: allocation for process p_i

$$a_i = s_i/S \times m$$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

- Adjust each a_i to be a integer greater than its minimum number of frames, with the sum not exceeding *m*.
- Allocation may vary according to the multiprogramming level.
- when a page fault occurs, one of the pages of that process must be replaced. -- Local replacement

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size, or on a combination of size and priority.
  - Dynamic as degree of multiprogramming, process sizes change.
- If process P_i generates a page fault,
  - select for replacement one of its frames.
    Local replacement
  - select for replacement a frame from a process with lower priority number.
    Global replacement

Global vs. Local replacement

- Global replacement
  - process selects a replacement frame from the set of all frames; one process can take a frame from another.
  - Easiest to implement, commonly used method.
  - Operating system keeps list of free frames.
  - Free frame is added to resident set of process when a page fault occurs.
  - One problem: a process cannot control its own page-fault rate.
- Local replacement
  - When page fault occurs, select page from among the resident set of the process that suffers the fault.
  - The set of pages in memory for a process is affected by the paging behavior of only that process.
- Global replacement results in greater system throughput.

Exercise 6

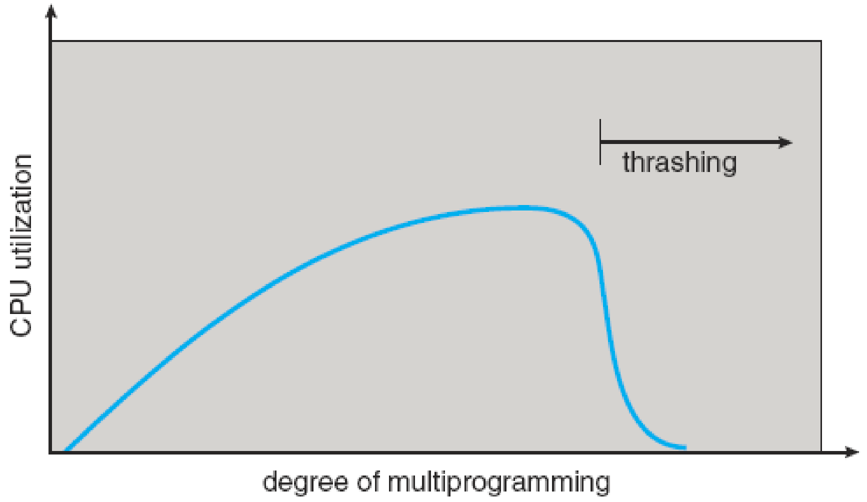
- Consider a machine in which all memory-reference instructions have two memory addresses, and one-level indirect addressing is allowed; if an instruction is assumed to be stored in only one frame, then the minimum number of frames per process is _____. And briefly why?

Answer to exercise 6

-

End 5

9.6 Thrashing



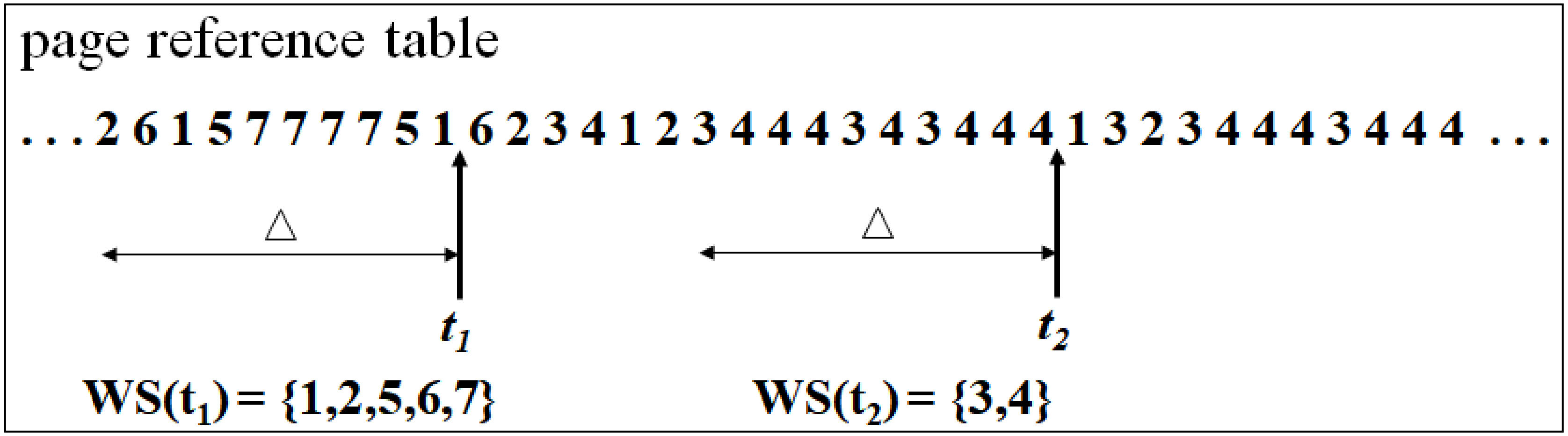
- If a process does not have "enough" pages, the page fault rate is very high.
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization.
    - OS thinks that it needs to increase the degree of multiprogramming.
    - Another process added to the system.
- This high paging activity is called thrashing.
- A process is thrashing if it is spending most time paging than executing.
- must decrease the degree of multiprogramming.

Thrashing(Cont.)

- **Why does thrashing occur?**
  - **Σ locality size > total memory size**
  - **using a local (or priority) replacement algorithm, Can limit the effect of thrashing.**
- **To prevent thrashing, must provide a process with as many frames as it needs.**
  - **How to know how many frames it "needs"?**
  - **looking at how many frames a process is actually using.**
- **Locality model of process execution**
  - **Locality: a set of pages that are actively used together, Localities are defined by the program structure and data structure.**
  - **Process migrates from one locality to another. e.g. function call.**
  - **Localities may overlap.**

Working-Set Model

- Δ ≡ working-set window ≡ a fixed number of page references.
  - Example: 10,000 instruction
- *Working Set*: the set of pages in the most recent Δ page references.



- *WSS$_i$* (working set size of Process *P$_i$*) : (varies in time) the number of pages in *Working Set* of process P$_i$.

Working-Set Model (Cont.)

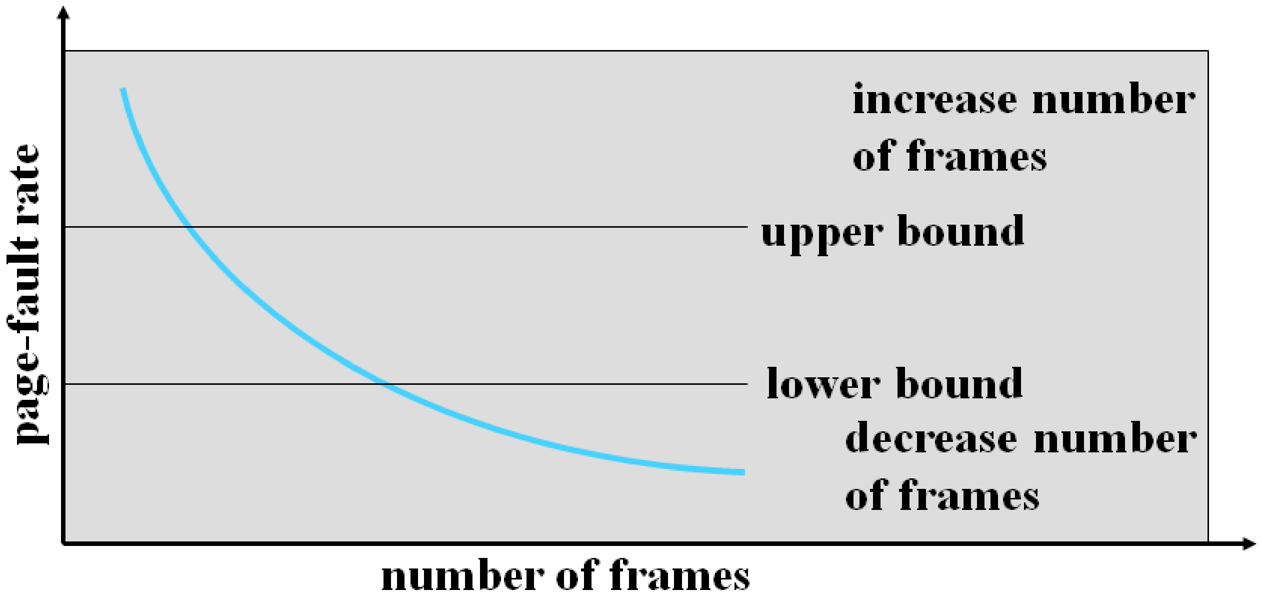
- The accuracy of the working set depends on the selection of Δ.
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.
- WSS_i -- process i needs WSS_i frames.
- $D = \Sigma WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m$ (available frames) $\Rightarrow$ Thrashing
- OS monitors the working set of each process.
  - if $D > m$, then suspend one of the processes.
- difficulty: keeping track of the working set.

Working-Set Model (Cont.)

- keeping track of the working set:
  - Approximate with interval timer + a reference bit
  - E.g : Δ = 10,000 references
    - Timer interrupts after every 5000 time units.
    - Keep 2 in-memory bits for each page.
    - Whenever a timer interrupts, copy and set the values of all reference bits to 0.
    - If one of the bits in-memory = 1 ⇒ page in working set.
- Why is this not completely accurate?
- Improvement: increasing the number of history bits and the frequency of interrupts.
  - E.g: 10 history in-memory bits and interrupt every 1000 time units.

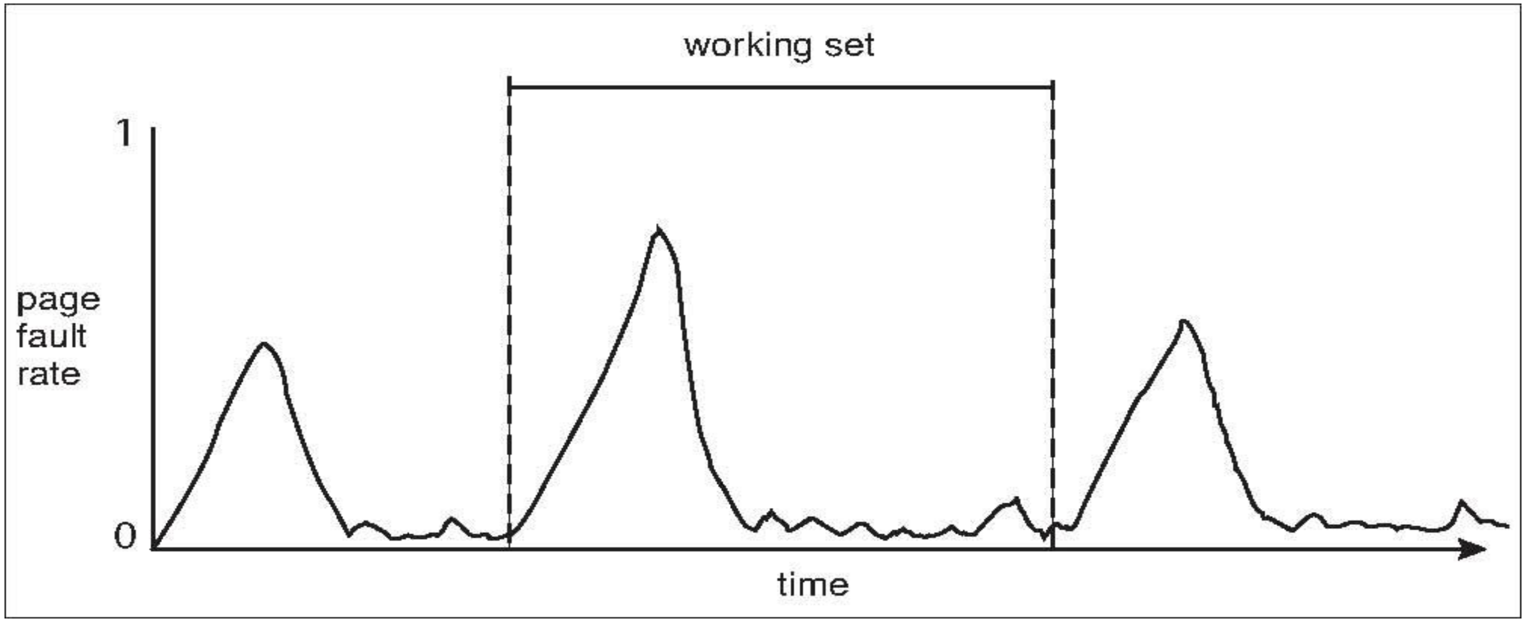
Page-Fault Frequency Scheme

- More direct approach than ***Working-Set Model***.
- **page-fault frequency (PFF) : establish "acceptable" page-fault rate and use local replacement policy.**
  - **If actual rate is too low, process loses frame.**
  - **If actual rate is too high, process gains frame.**
    Select some process and swap it out to backing store.



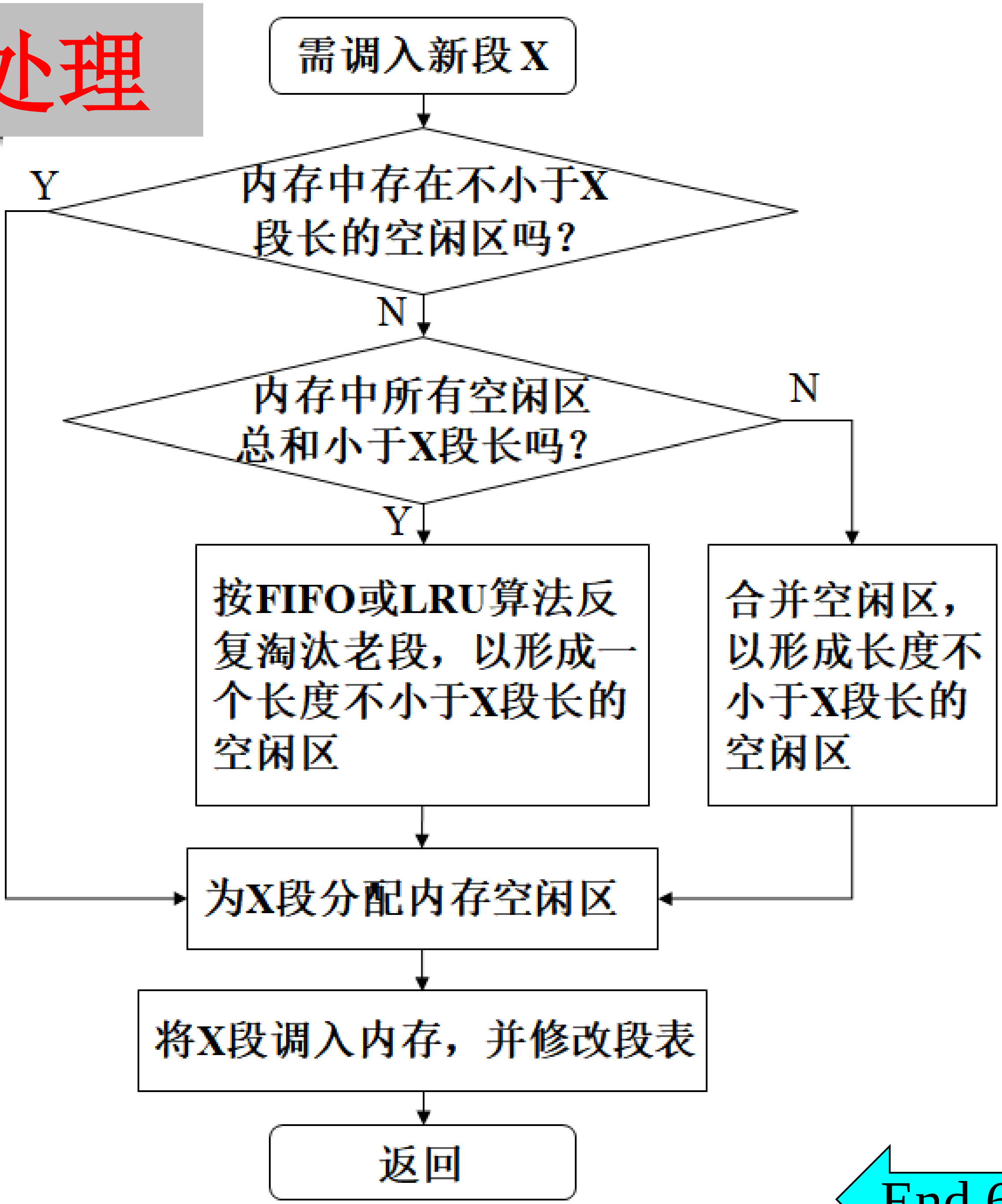
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate.
- Working set changes over time.
- Peaks and valleys over time.



* 缺段中断及处理

- 当CPU要访问的指令/数据不在内存中时，产生“缺段中断”。
- OS捕获中断，并调用相应的中断处理程序，进行处理。
- 缺段处理过程：



需调入新段X
内存中存在不小于X段长的空闲区吗？
Y
N
内存中所有空闲区总和小于X段长吗？
N
Y
按FIFO或LRU算法反复淘汰老段，以形成一个长度不小于X段长的空闲区
合并空闲区，以形成长度不小于X段长的空闲区
为X段分配内存空闲区
将X段调入内存，并修改段表
返回

End 6

9.7 Memory-Mapped Files

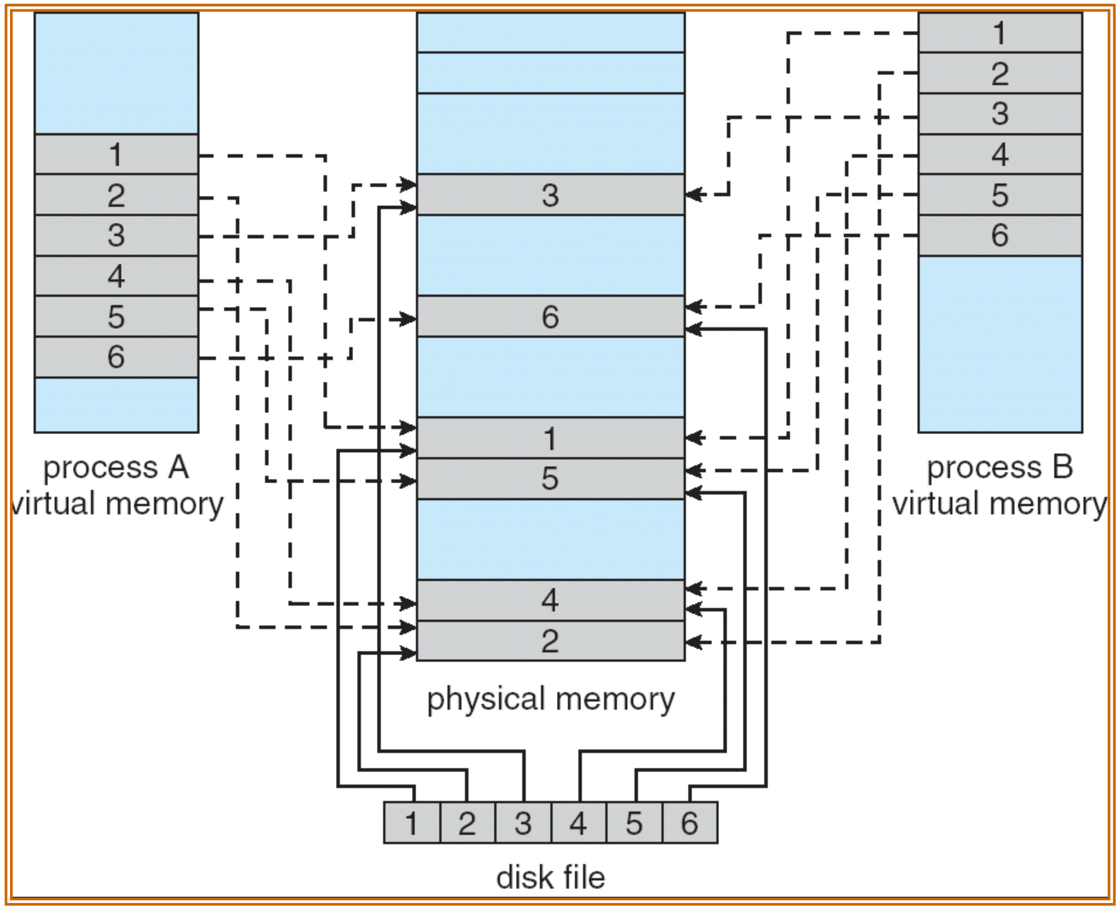
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.
- A file is initially read using demand paging.
    - A page-sized portion of the file is read from the file system into a physical page.
    - Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies and speeds up file access by treating file I/O through memory rather than read() and write() system calls.
- When does written data make it to disk?
    - Periodically, e.g, when the pager scans for dirty pages.
    - at file close() time.

Memory-Mapped File (Cont.)

- **Some Oses, memory mapping only through a specific system call.**
- **Some Oses, choose to memory-map a file. (Solaris)**
  - **Process explicitly request memory mapping a file via mmap() system call.**
    - **File is mapped into process address space.**
  - **For standard I/O (open(), read(), write(), close())**
    - **File is mapped to kernel address space.**
    - **Process still does read() and write()**
      - **Copies data to and from kernel space and user space**
  - **Uses efficient memory management subsystem.**

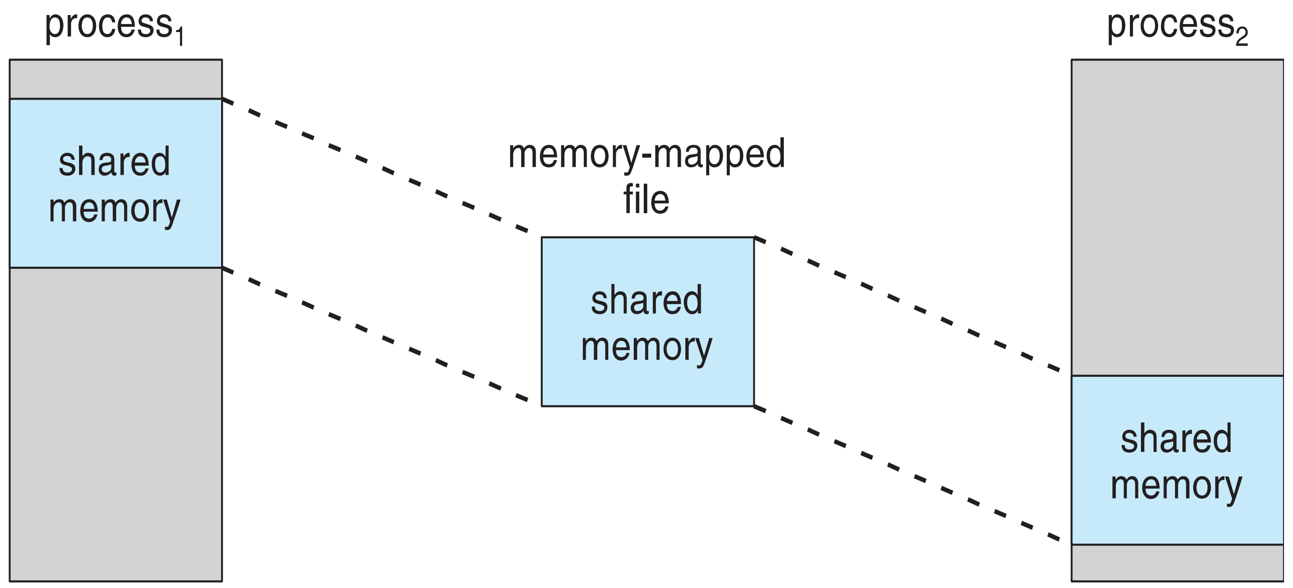
Memory-Mapped Files (Cont.)

- **Allows several processes to map the same file concurrently, allowing the pages in memory to be shared.**



Memory-Mapped Files (Cont.)

- **COW can be used for read/write non-shared pages.**
  - **allowing processes to share a file in read-only mode but to have their own copies of any data they modify.**
- **Memory mapped files can be used for shared memory.**



Shared Memory in Windows API

- First create a file mapping for the file to be mapped.
- Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
  - Producer create shared-memory object using memory mapping features.
  - Open file via CreateFile(), returning a HANDLE.
  - Create mapping via CreateFileMapping() creating a named shared-memory object.
  - Create view via MapViewOfFile().
- Sample code in Textbook (P.351-352)

Memory-mapped I/O

- ranges of memory addresses are set aside and are mapped to the device registers.
  - Reads and writes to these memory addresses cause the data to be transferred to and from the device registers.
  - appropriate for devices that have fast response times, such as video controllers.
  - e.g. IBM PC, each location on the screen is mapped to a memory location.
- also convenient for other devices, such as the serial and parallel ports used to connect modems and printers to a computer.
  - The CPU transfers data through these kinds of devices by reading and writing a few device registers, called an I/O port.

End 7

9.8 Allocating Kernel Memory

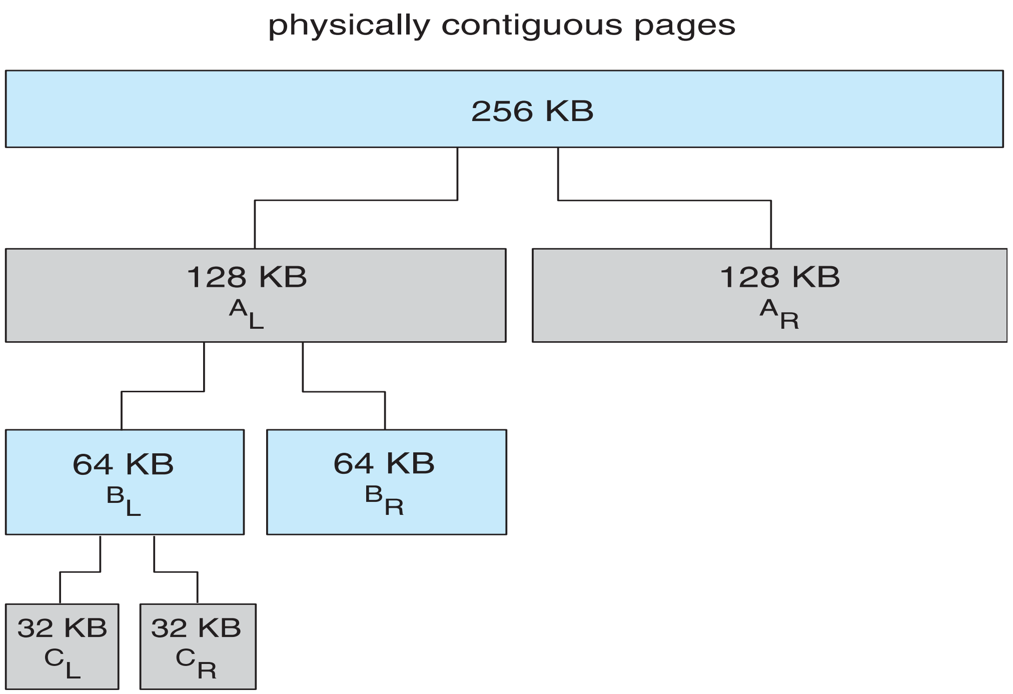
- Treated differently from user memory .
- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes.
  Reasons are:
  - Kernel requests memory for data structures of varying sizes, some of which are less than a page in size.
  - Some kernel memory needs to be contiguous.
    i.e. for device I/O
    Certain hardware devices interact directly with physical memory, and may require memory residing in physically contiguous pages.
- Buddy system
- Slab allocation

Buddy System (伙伴系统）

- Allocates memory from a fixed-size segment consisting of physically contiguous pages.
- Memory is allocated by using a power-of-2 allocator.
  - Satisfies requests in units sized as power of 2.
  - Request rounded up to next highest power of 2.
  - Entire space available is treated as a single block of 2^U
    - If a request of size s such that $2^{U-1} < s <= 2^U$, entire block is allocated,
      Otherwise block is split into two equal buddies.
    - Process continues until smallest block greater than or equal to s is generated.

Buddy System (Cont.)

- E.g. 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each.
  - One further divided into B_L and B_R of 64KB each
  - One further divided into C_L and C_R of 32KB each
  - One used to satisfy request.
- Advantage – quickly coalesce unused (buddy) chunks into larger chunk.
- Disadvantage – internal fragmentation



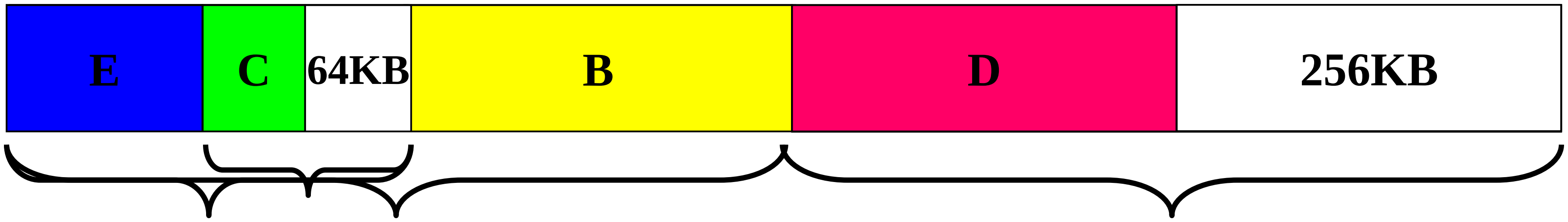
Buddy System， example

1MB block

A: Request 100KB **B: Request 240KB** **C: Request 64KB**

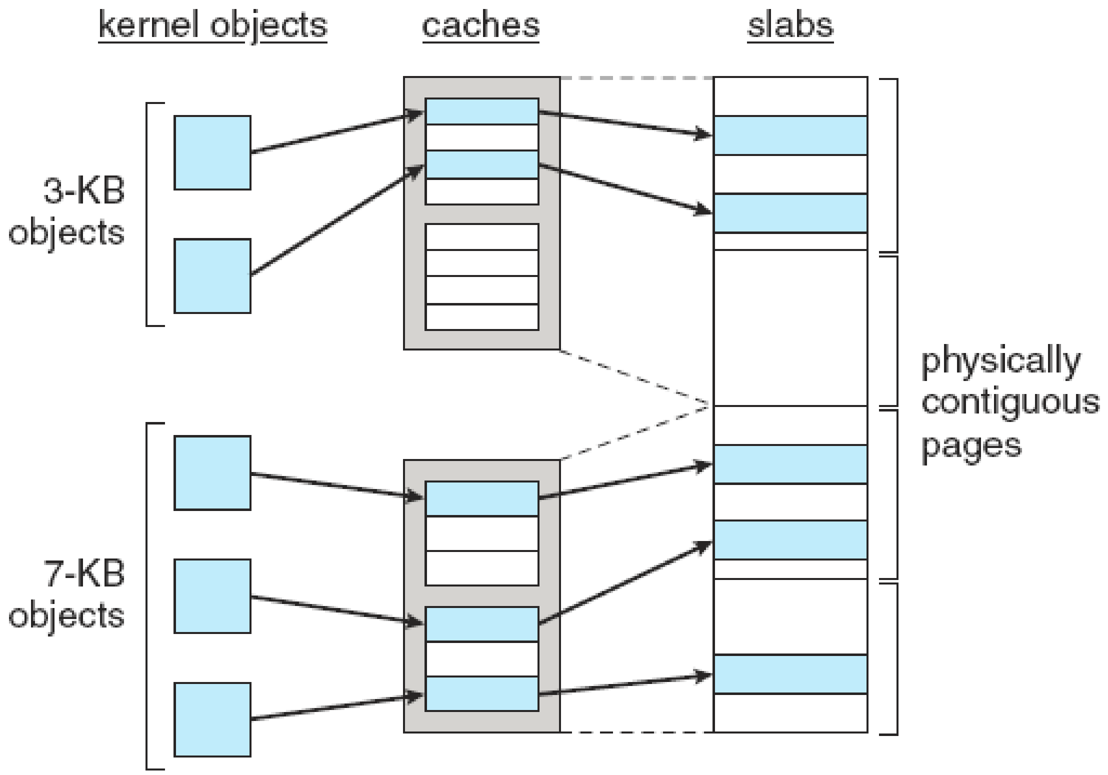
D: Request 256KB **Release B** **Release A**

E: Request 75KB **Release C** **Release E** **Release D**



Slab allocation

- A slab is made up of one or more physically contiguous pages.
- A cache consists of one or more slabs.
- There is a single cache for each unique kernel data structure, e.g.
  - PCBs
  - semaphores
- Each cache is filled with objects,
  - instantiations of the kernel data structure.



Slab allocation (Cont.)

- When cache created, filled with objects marked as free
  - The number of objects in the cache depends on the size of the associated slab.
- When structures stored, objects marked as used.
- If slab is full of used objects, next object allocated from empty slab.
  - If no empty slabs, new slab allocated.
- Benefits
  - no fragmentation
  - fast memory request satisfaction

Slab Allocator in Linux

- For example process descriptor is of type struct task_struct, approx 1.7KB of memory.
- New task → allocate new struct from cache
  - Will use existing free struct task_struct
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty slab and assigned to a cache, memory is allocated from this slab.

* Slab Allocator in Linux (Cont.)

- Slab started in Solaris 2.4, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited amount of memory.
    - Simple List of Blocks, maintains 3 lists of objects for
      - Small objects, objects<256B
      - medium objects, objects<1024B
      - large objects, objects<page size
    - Memory requests are allocated from an object on an appropriately size list using first-fit policy.
  - SLUB is now the default allocator, performance-optimized SLAB
    - Moves the metadata stored with each slab to the page structure for each page.
    - Removes per-CPU queues for objects in each cache.

end 8

9.9 Other Considerations

- Prepaging
- Page Size
- TLB Reach
- Inverted Page Tables
- Program Structure
- I/O interlock

Prepaging

- To reduce the large number of page faults that occurs at process startup or a swapped-out process resatart.
- Prepage all or some of the pages a process will need, before they are referenced.
- if prepaged pages are unused, I/O and memory was wasted.
- Prepaging may be an advantage in some cases.
  - whether the cost of using prepaging is less than the cost of servicing the corresponding page faults.
- Example
  - S pages are prepaged
  - a fraction α of these S pages is actually used ($0<\alpha<1$)
  - the cost of the $S\times\alpha$ saved page faults is greater or less than the cost of prepaging $S\times(1-\alpha)$ unnecessary pages.
  - if α is close to 0, prepaging loses; if α is close to 1, prepaging wins.

Page size selection

- Page size ranging from 4KB to 4MB.
- size of the page table
  - each active process must have its own copy of the page table, a large page size is desirable.
- fragmentation
  - to minimize internal fragmentation, small page size is needed.
- I/O overhead
  - I/O time = seek time + latency time + transfer time
  - seek and latency time normally dwarf transfer time
  - to minimize I/O time, large page size is desired
- locality
  - with a small page size, locality will be improved, total I/O should be reduced.
  - to minimize the number of page faults, a large page size is need.

TLB Reach

- The amount of memory accessible from the TLB.
- TLB Reach = (TLB Size) × (Page Size)
- Ideally, the working set of each process is stored in the TLB.
  - Otherwise, there is a high degree of page faults.
  - process will spend a considerable amount of time resolving memory references in the page table rather than TLB.
- Increasing the Size of the TLB reach by
  - Increase the Page Size. This may lead to an increase in fragmentation, as not all applications require a large page size.
  - Provide Multiple Page Sizes. This allows applications that require larger page sizes to have the opportunity to use them without an increase in fragmentation.

* Example Page Sizes

Computer	page size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes to 4 Mbytes
PowerPC	4 Kbytes

Inverted Page Tables

- Purpose: to reduce the amount of physical memory needed to track virtual-to-physical address translations.
  - has one entry per frame of physical memory.
  - indexed by the pair <process-id. page-number>.
  - keep information about which virtual memory page is stored in each physical frame.
- When page faults, other information required.
  - e.g on where each virtual page is located, the number of pages.
- An external page table (one per process) must be kept.
  - Be paged in and out of memory as necessary.
- A page fault may now cause the virtual memory manager to generate another page fault to page in the external page table it needs.

Program structure

- Page size = 1024 words
- int A[ ][ ] = new int[1024][1024];
- Each row is stored in one page
- The OS allocates fewer than 1024 frames to the entire program
- Program 1:

```
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        A[i][j] = 0;
```

  1024 × 1024 page faults
- Program 2:

```
for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
        A[i][j] = 0;
```

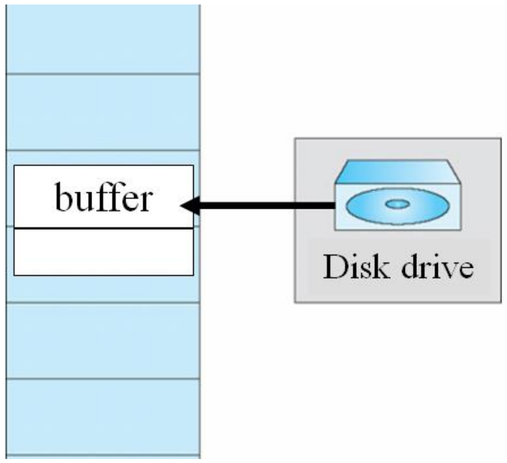
  1024 page faults

Program structure (Cont.)

- Careful selection of data structures and programming structures can increase locality, and hence lower the page-fault rate and the number of pages in the working set. e.g.
  - A stack has good locality, since access is always made to the top.
  - A hash table is designed to scatter references, producing bad locality.
- compiler and loader have a significant effect on paging.
  - Separating code and data and generating reentrant code means that code pages can be read-only, and will never be modified.
    - Clean pages do not have to be paged out to be replaced.
  - loader can avoid placing routines across page boundaries
    - keeping each routine completely in one page.
    - Routines that call each other many times can be packed into the same page.

I/O Interlock and page locking

- Sometimes pages must be locked into memory.
- Consider pending I/O.
  - Pages that are used for I/O from a device must be locked from being selected for eviction by a page replacement algorithm.
- Never to execute I/O to user memory.
  - Data are copied between system memory and user memory.
  - I/O takes place only between system memory and I/O device.
- Allow pages to be locked into memory.
  - A lock bit is associated with every frames.
  - Locked page cannot be replaced
  - When I/O complete, unlocked.
- Pinning of pages to lock into memory.



I/O Interlock and page locking (Cont.)

- Some or all of the OS kernel is locked into memory.
- User processes may need to lock pages into memory.
- Consider: system with demanding paging, priority scheduling, global replacement.
  - A low-priority process L faults.
    - OS selecting a replacement frame, and paging the necessary page in.
    - Then, L enters the ready queue. It may not be selected for a long time.
  - While L waits, a high-priority process H faults.
    - Looking for a victim, it is the page that L just brought in -- clean and has not been used for a long time.

I/O Interlock and page locking (Cont.)

- Use the lock bit, to prevent replacement of a newly brought-in page until it can be used at least once.
  - When a page is selected for replacement, its lock bit is turned on.
  - Lock remains on until the faulting process is again dispatched.
- Most OSes have a system call allowing an application to request a region of its logical address space be pinned.
- Dangerous: lock bit may get turned on but never turned off.
  - Locked frame becomes unusable.

End 9

9.10 Operating System Examples(*)

- Windows
  - demand paging with clustering
  - maintains a list of free frames and a threshold value
  - Processes: working set minimum and maximum.
    - At working set maximum, using local LRU policy
  - *clock* / FIFO algorithm, up to the type of processor
- Solaris
  - demand paging
  - Maintains a list of free pages
  - Maintains a cache list of pages that have been "freed" but have not yet been overwritten
  - second chance algorithm, two-hand clock

Homework (page 366)

9.4 9.5 9.11

- experiment 4: 9.21

supplement

- Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.
- Consider the following page reference string:
  1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
  How many page faults would occur for the following replacement algorithms, assuming three, or four frames?
  Remember all frames are initially empty, so your first unique pages will all cost one fault each.

1. LRU replacement
2. FIFO replacement
3. Optimal replacement

You can do it!