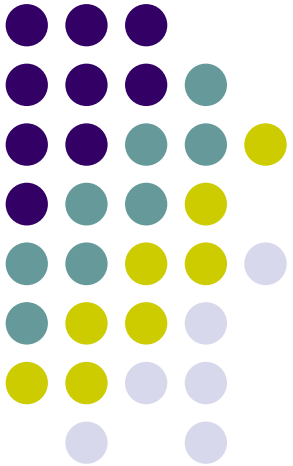
Chapter 9: Virtual Memory Management

WHAT'S AHEAD:

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- OS Examples

WE AIM:

- 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

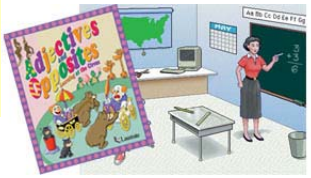


Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심•요점 가상기억(virtual memory)은 실제 있다

"virtual"의 정의: 겉모습이나 명목상으로는 그렇지 않으나 실제 존재하는, 즉 실질적인, 사실상의, 실제적인

우리말 "가상"의 정의: 1. [가상(假想)] 사실이 아니거나 사실 여부가 분명하지 않은 것을 사실이라고 가정하여 생각함. 2. [가상(假像)] 실물처럼 보이는 거짓 형상. {네이버 국어사전} 즉, 우리말 "가상 " 의 뜻은 실재하지 않는 것을 의미함. 그러나 가상 기억은 실제 존재하는 기억공간을 의미함. 따라서 영어 용어인 "virtual memory"에 대한 우리말 번역인 "가상기억 " 은 원래의 뜻을 잘못 전달할 우려가 있음.



그렇다면 가상기억은 어디에 존재하는가?

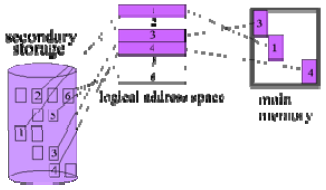
소프트웨어 프로그램 – 논리적 주소공간

그리고 어디에 존재하지 않는가?

주기억장치에 없음 – 물리적 주소공간

가상 주소는 무엇으로 귀결되는가?

"변환"을 통하여 주기억장치의 실제 주소

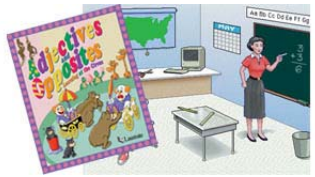


Core Ideas

Virtual Memory Is Not Unreal

"**Virtual**" is defined as *existing* or *resulting* in essence or effect though *not in actual fact*, form, or name.

http://www.thefreedictionary.com/virtual



Then, where does the virtual memory exist?

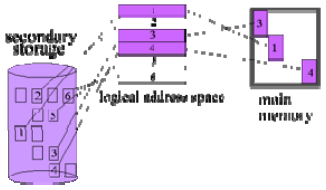
In software programs – Logical address space

And, where does it NOT exist?

Not in the main memory – Physical memory space

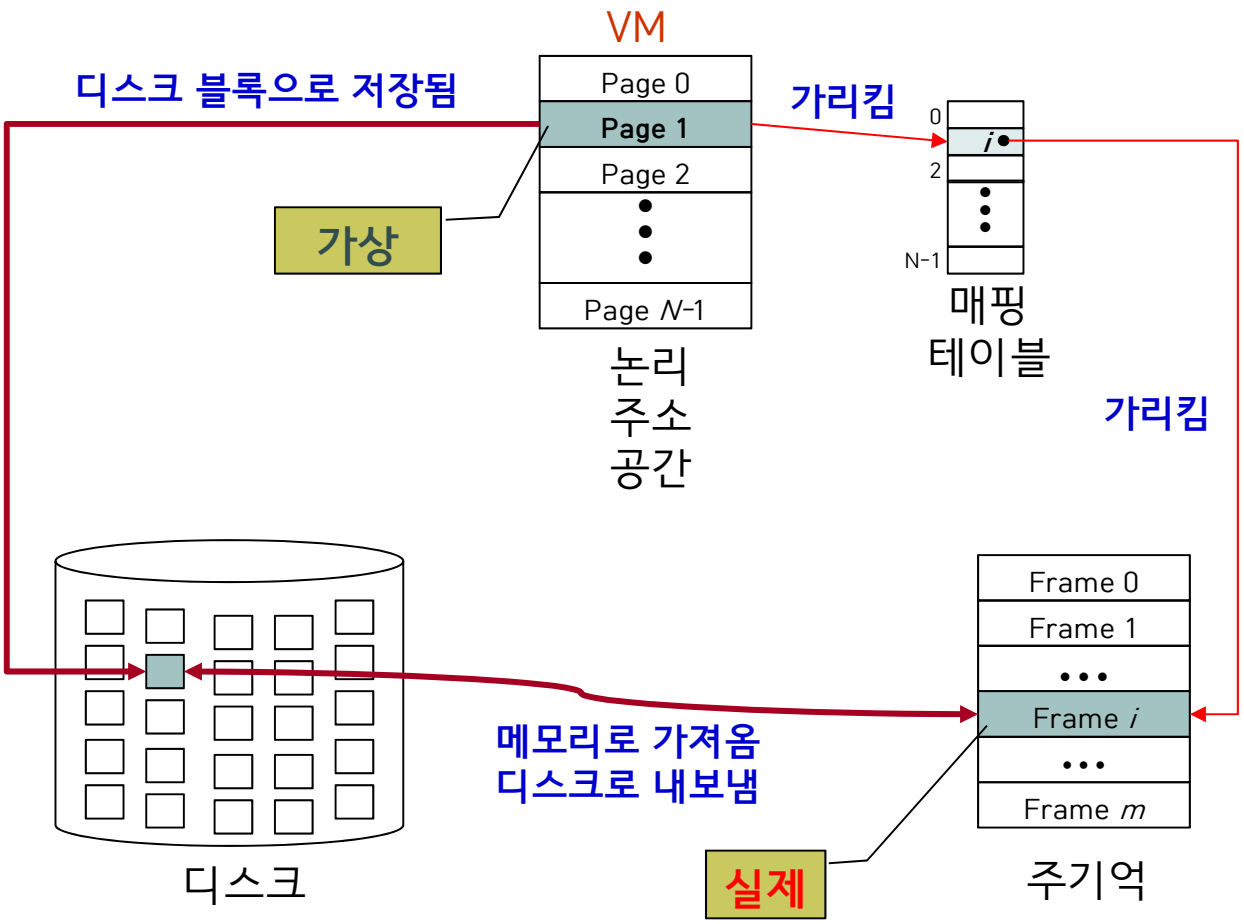
What does a virtual address result in?

Real address in main memory through "translation"



핵심•요점 가상기억은 어떻게 구현되어 기능하는가?

"페이지 1은 실제로 어떻게 구현되는가?



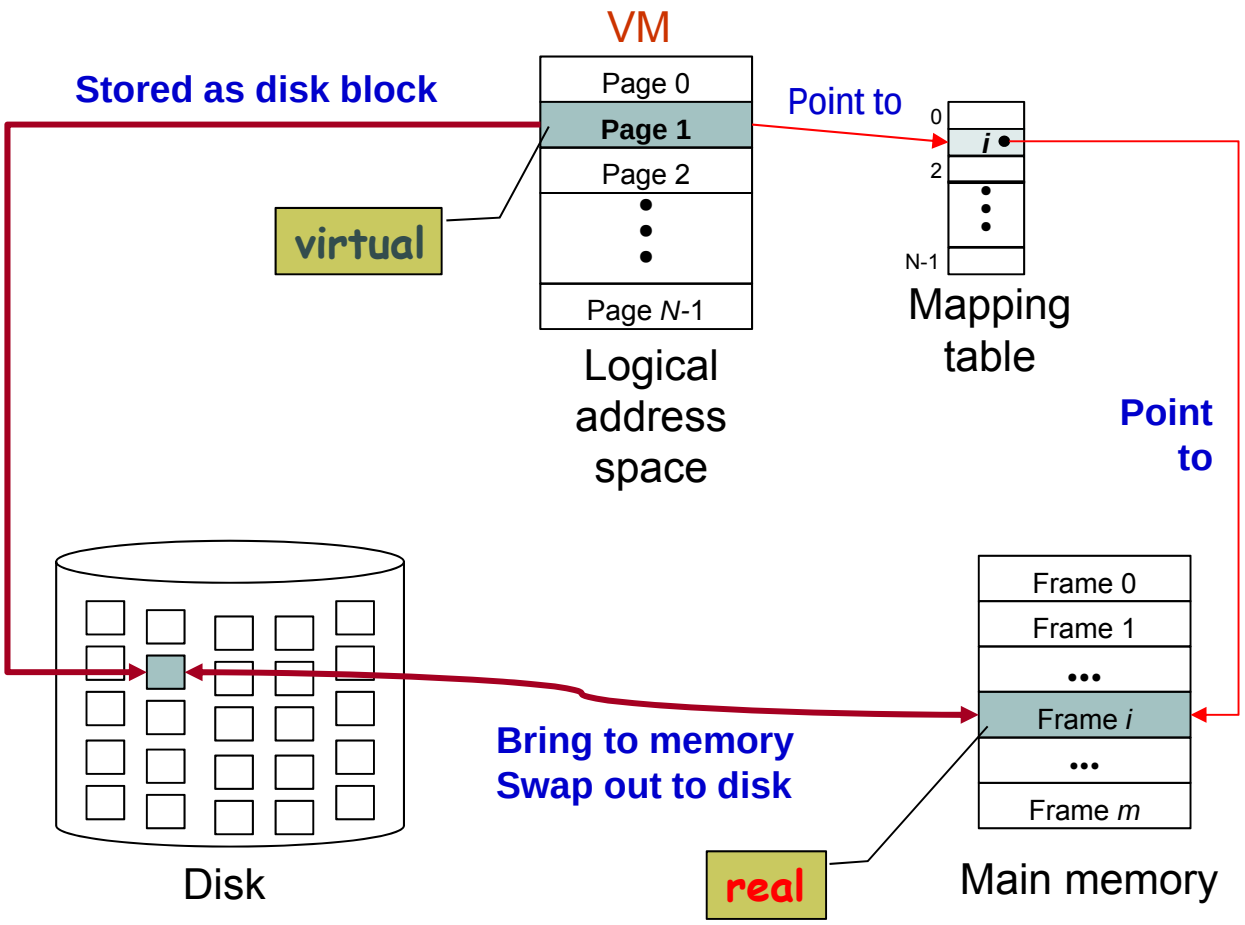
Remember !

- ✓ Virtual address space 가상주소공간
- ✓ Page table 페이지 테이블
- ✓ "Paging" in/out 페이지의 입출력 수행
- ✓ Demand paging 요구 페이징
- ✓ Page fault 페이지 폴트
- ✓ Copy-on-write 쓸 때 복사
- ✓ Page replacement 페이지 재배치
- ✓ Least recently used (LRU) algorithm 가장 오래전 사용 알고리즘
- ✓ Clock algorithm 클록 알고리즘
- ✓ Thrashing 쓰래싱
- ✓ Working set 워킹 셋
- ✓ Memory-mapped file 메모리 사상 파일
- ✓ Buddy system allocator 버디 시스템 할당기
- ✓ Slab allocator 슬랩 할당기

Core Ideas

How Is VM Made Real & Working?

How is "Page 1" made real?



Remember !

- ✓ Virtual address space
- ✓ Page table
- ✓ "Paging" in/out
- ✓ Demand paging
- ✓ Page fault
- ✓ Copy-on-write
- ✓ Page replacement
- ✓ Least recently used (LRU) algorithm
- ✓ Clock algorithm
- ✓ Thrashing
- ✓ Working set
- ✓ Memory-mapped file
- ✓ Buddy system allocator
- ✓ Slab allocator

Background

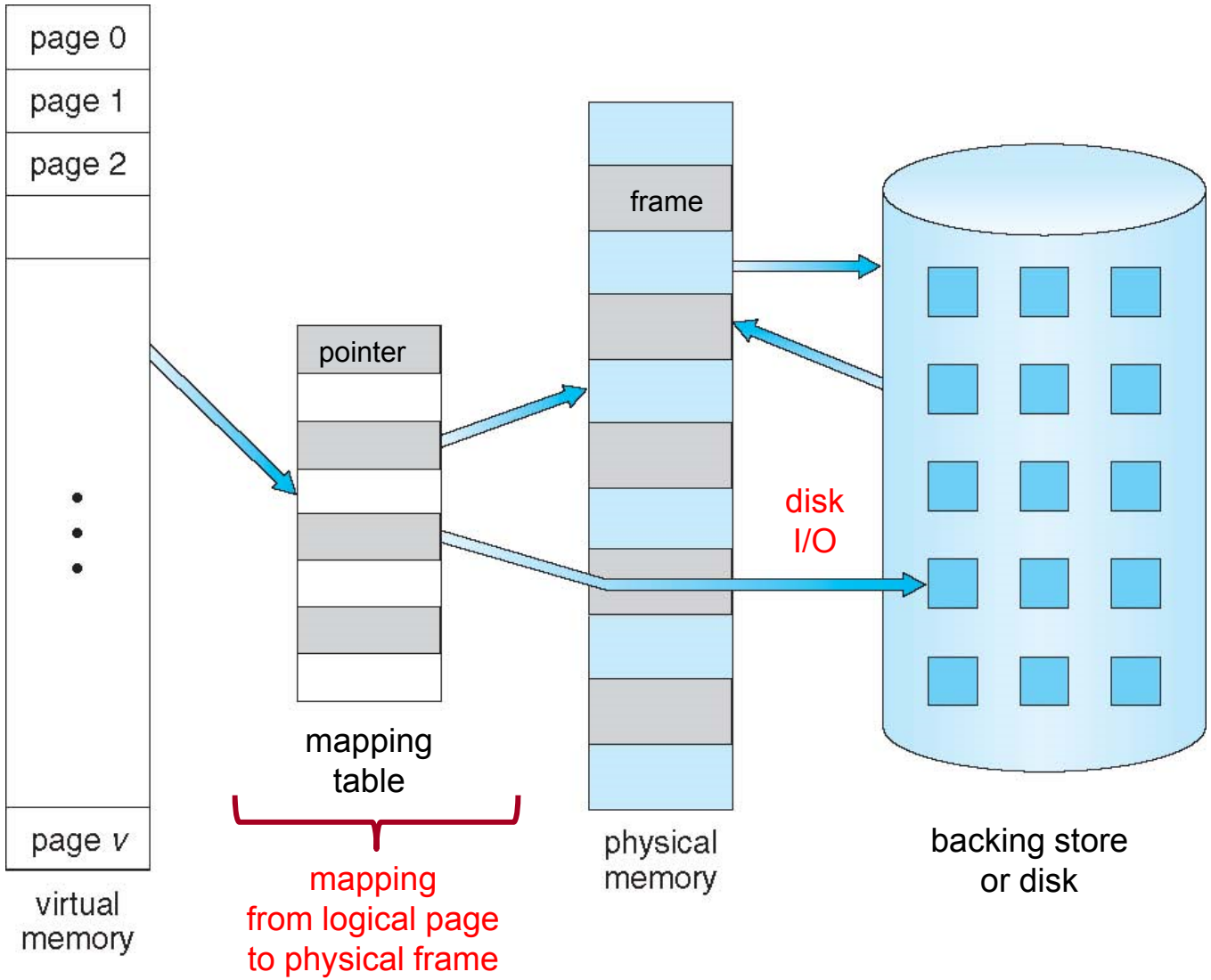
- Von Neumann architecture
  - Code needs to be in memory to execute
  - Memory is not usually large enough for the entire program code (*cf.* memory hierarchy)
- On the other hand, entire program code is rarely needed at same time
  - We can place part of a program in memory at a time
- Consider ability to execute partially-loaded program
  - Programs no longer constrained by limits of physical memory
  - Programs could be larger than physical memory

Background (Cont.)

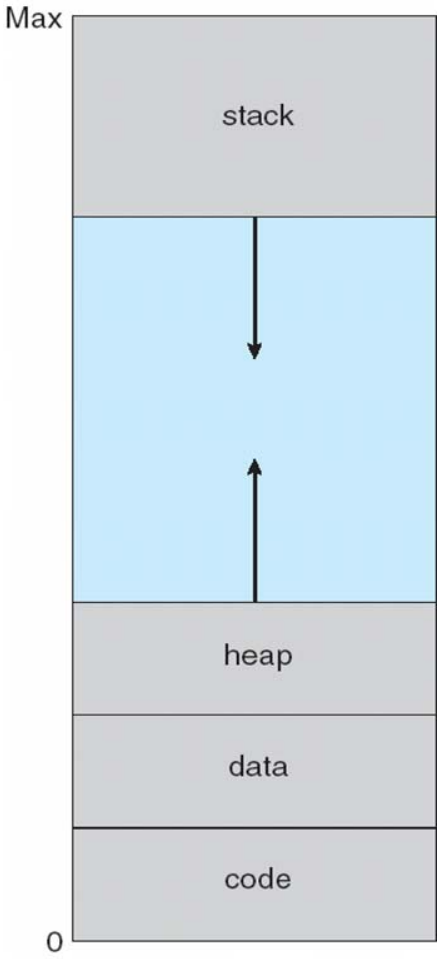
- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more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

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

Virtual Memory That is Larger Than Physical Memory



Virtual-address Space

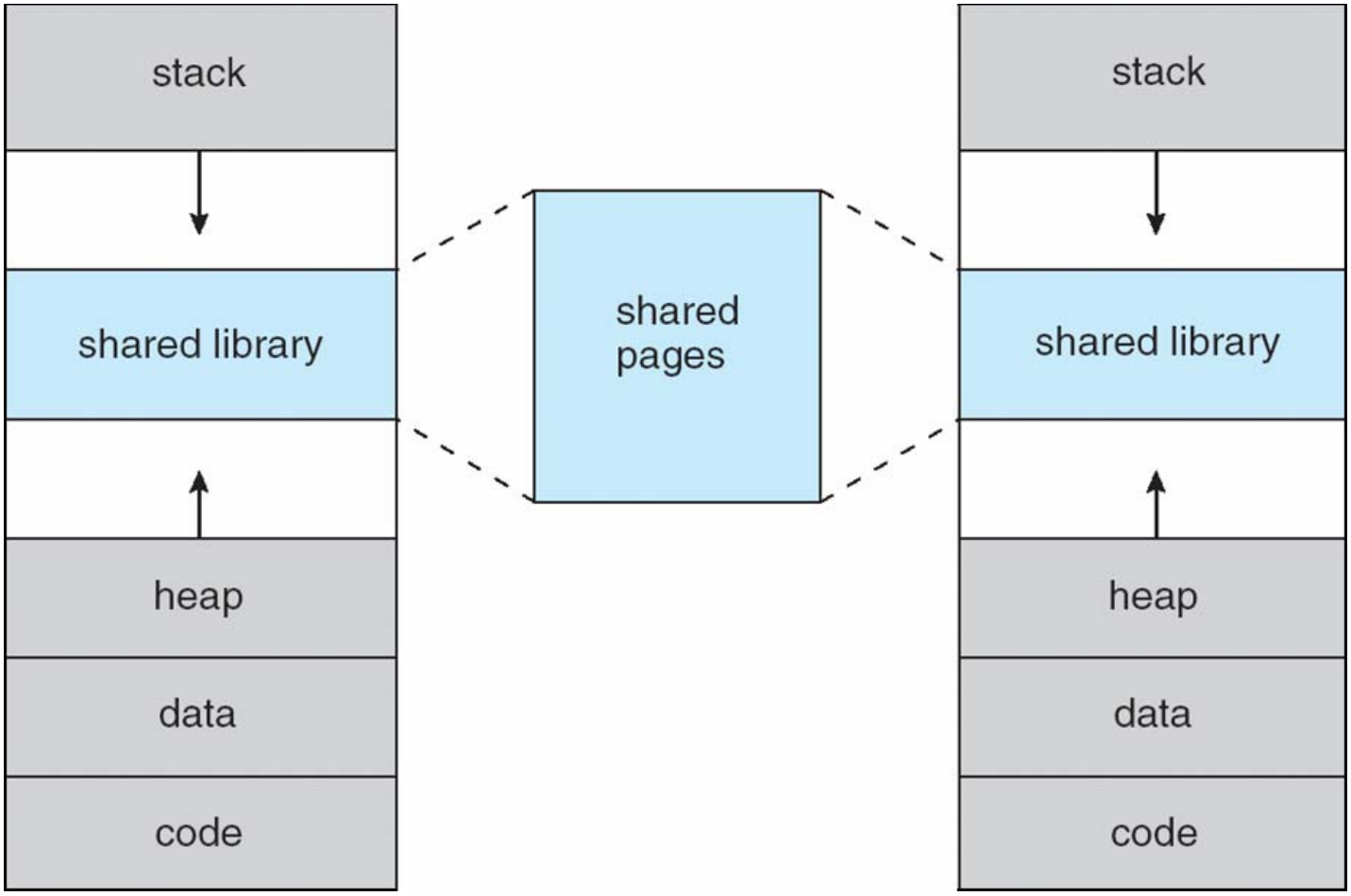


This diagram shows how a user program and its runtime data structures are organized in the virtual address space.
The configuration of virtual address space differs among various operating systems.

Virtual Address Space

- This is where a process works
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space for interprocess communication
- Pages can be shared during `fork()`, speeding up process creation

Shared Library Using Virtual Memory



Demand Paging

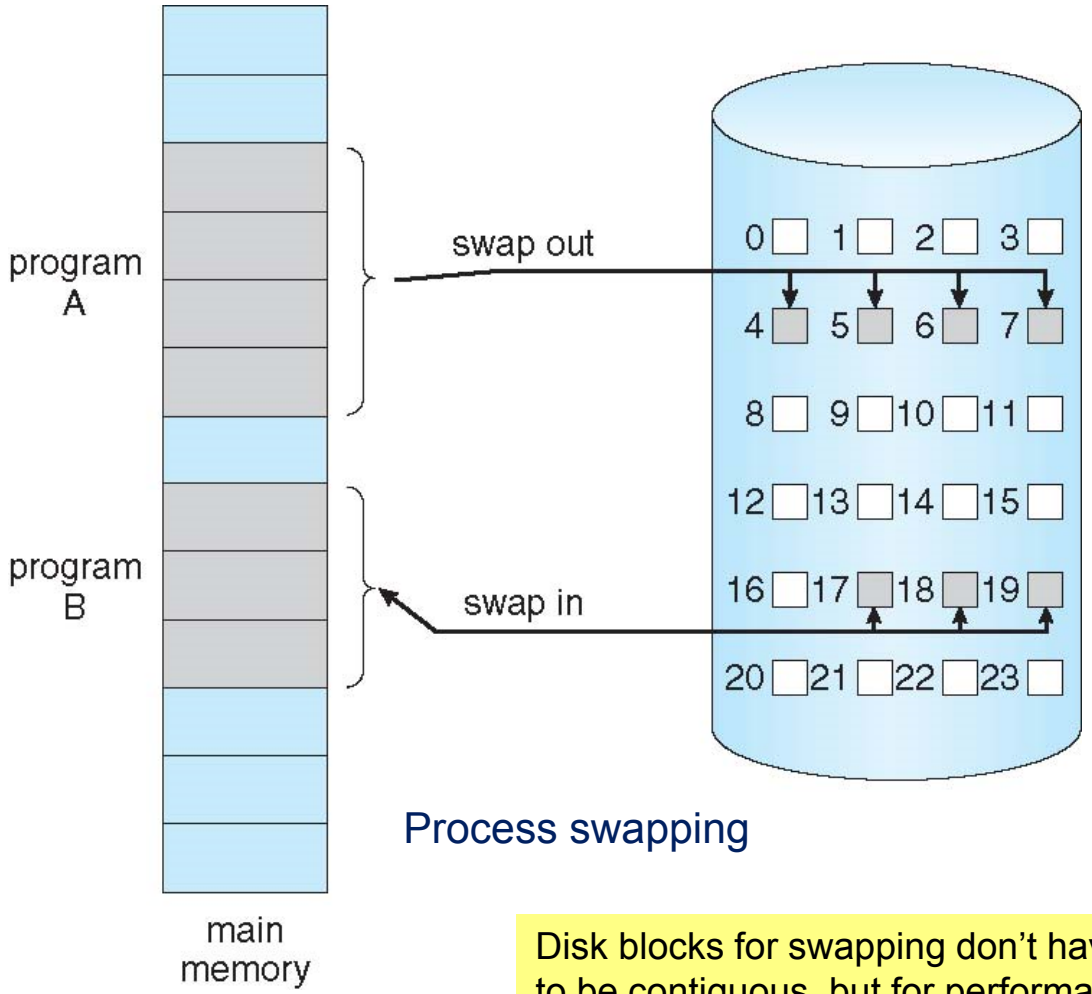
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Strictly speaking, a **swapper** manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process.

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
  - Sometimes, swapper and pager are used interchangeably

Transfer of a Paged Memory to Contiguous Disk Space

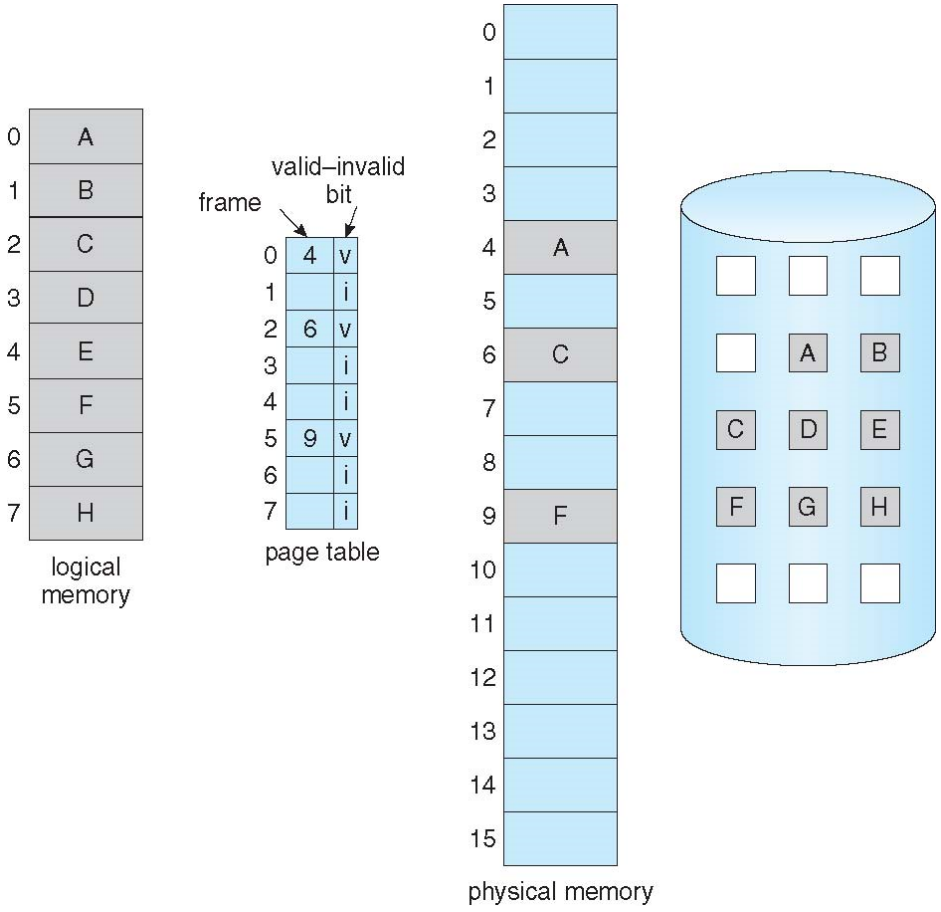


Process swapping

Disk blocks for swapping don't have to be contiguous, but for performance reason their contiguity is important.

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ **page fault**

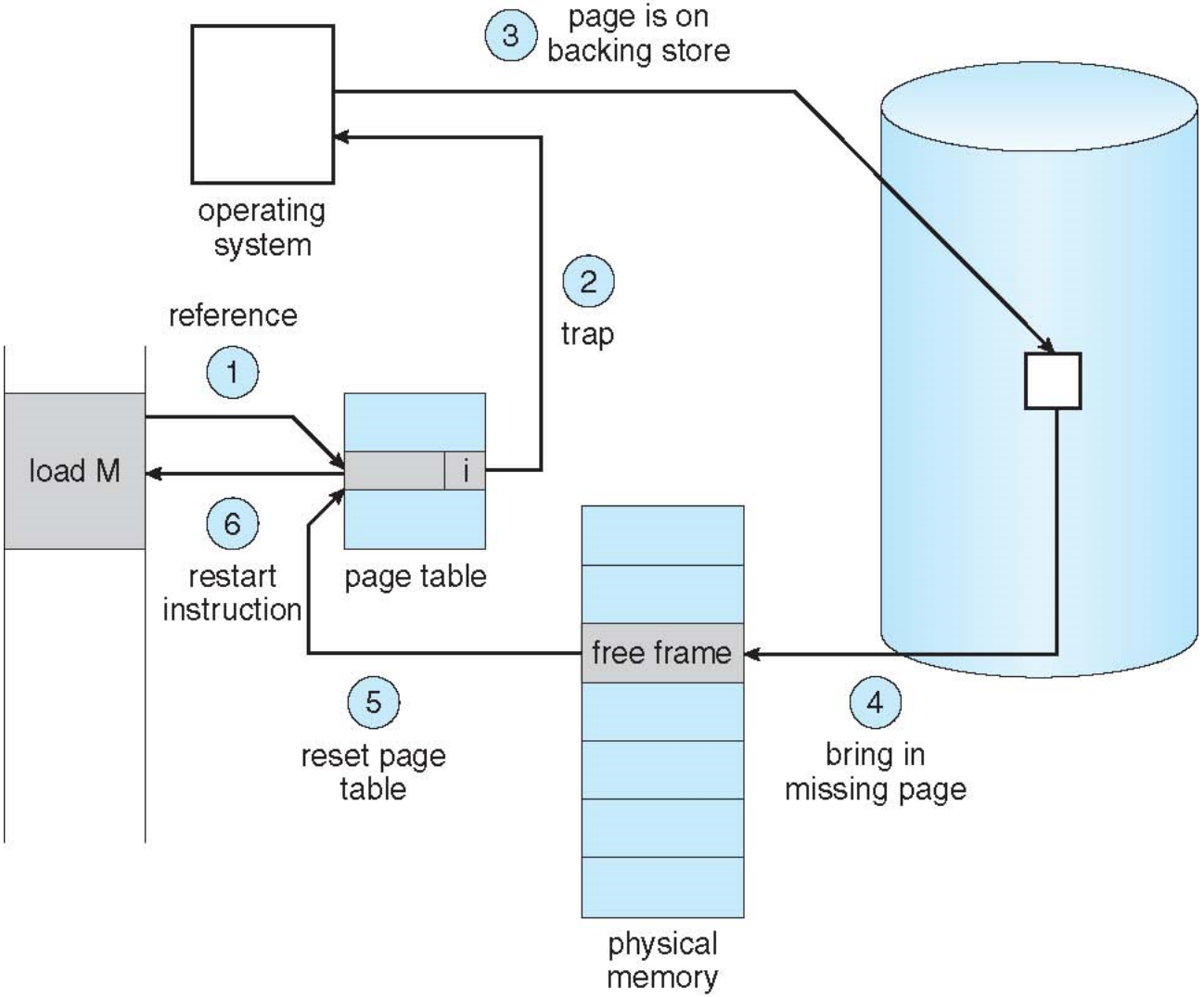


Page table when some pages are not in main memory

Page Fault

- Rough description
  - Memory management unit (MMU) generates a "page fault" interrupt if a page-frame mapping indicates an absence of the page in physical memory
  - OS traps this fault and the interrupt handler services the fault by initiating a disk I/O to read in the page
  - Once brought into main memory, page-table entry is updated and the process which faulted is restarted.
- Handling a page fault – Step-by-step procedure
  1. Operating system looks at another table to decide:
     - Invalid reference ⇒ abort
     - Just not in memory ⇒ page it in
  2. Get empty frame for the page to be brought in
  3. Swap page into frame via scheduled disk operation
  4. Reset tables to indicate page now in memory
     Set validation bit = **v**
  5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault

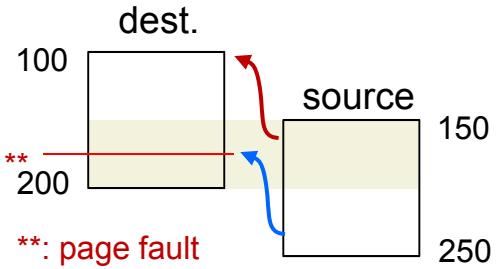


Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process, pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory to hold those pages that are not in main memory
    - usually a high-speed disk known as swap device
    - **swap space**: the section of disk with special format used for this purpose, different from file system
- Instruction restart (next slide)
  - Restart an instruction from the very beginning of the instruction which has caused the page fault

Instruction Restart

- The ability to restart any instruction after page fault is crucial
- Worst-case scenario: Consider an instruction that could access several different locations
  - block move: IBM System 360/370 MVC (move character) instruction
    - move up to 256 bytes from one location to another (possibly overlapping) location



  - What if either block (source or destination) crosses a page boundary? → page fault
    - if blocks are overlapped → source block may have been modified
  - a solution: upon instruction execution, check if a page fault will occur, and bring in necessary pages in advance

Note: Page fault is detected in the midst of instruction execution, not at the boundary of instruction. What if instruction execution causes side effect before the page fault is detected? We may perform either roll back or inspection upon instruction execution.

Performance of Demand Paging (Cont.)

- Page fault rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p x page fault time

Three major components of the page fault service time:
1. Service the page-fault interrupt
2. Read in the page
3. Restart the process

page fault time = page fault overhead
+ swap page out
+ swap page in
+ restart overhead

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p x 8,000,000 (nanoseconds)
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

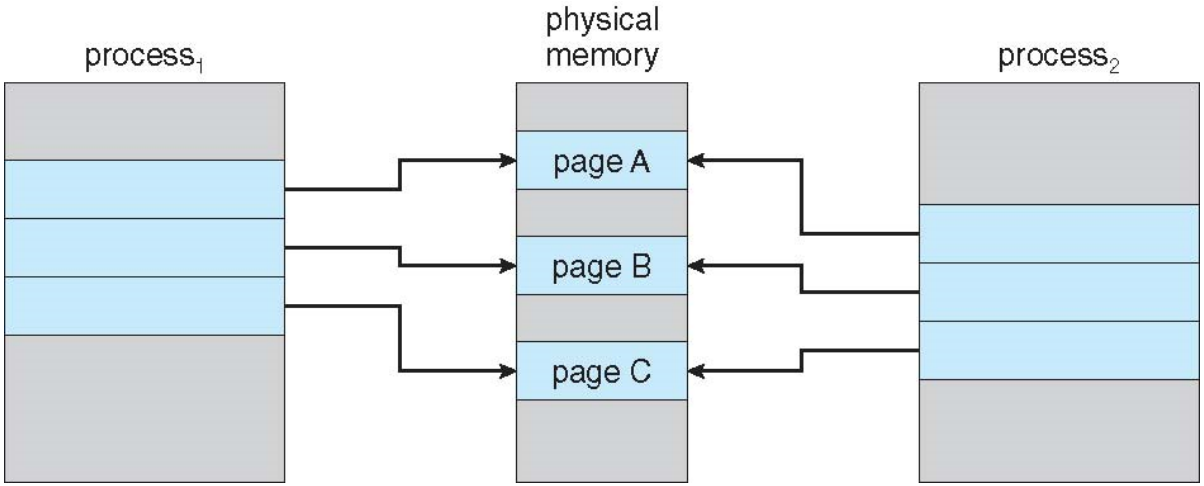
Demand Paging Optimizations

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand-page in program binary from the file system. Swap space is used for pages that include the stack and heap, not associated with a file
  - i.e., load module or executable file (static) is in file system while pages of anonymous memory, such as stack, heap, etc. (dynamic), are in swap space
  - Used in Solaris and current BSD
- Mobile OS
  - typically do not support swapping
  - demand-page from the file system

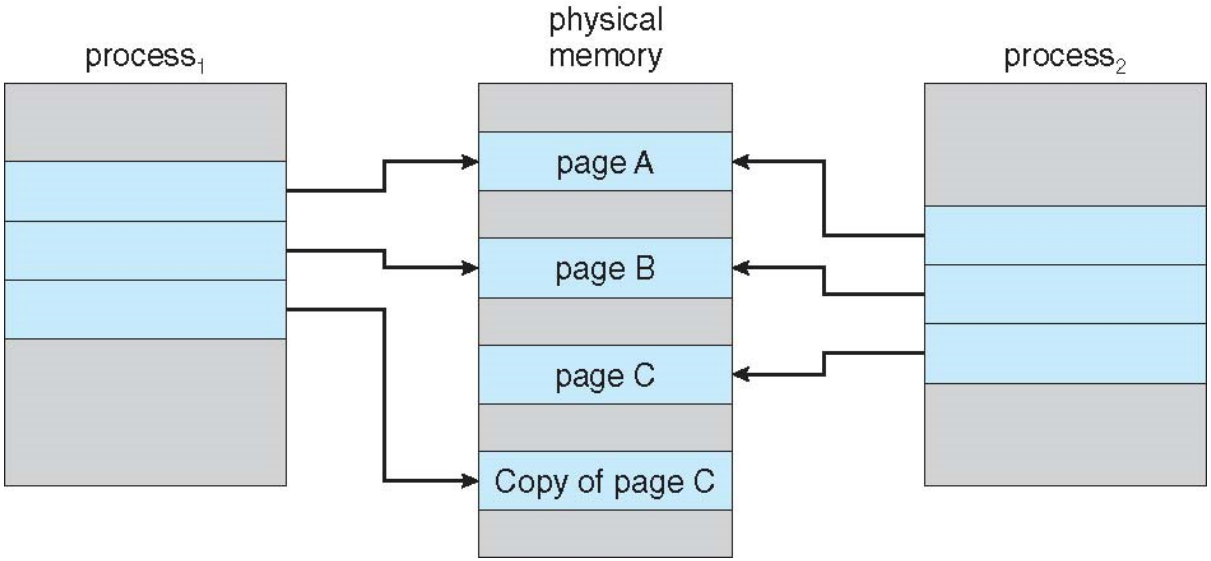
Copy-on-Write

- **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
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Why zero-out a page before allocating it?

Example: Copy on Write



Before Process 1 modifies page C



After Process 1 modifies page C

Page Replacement

What happens if there is no free frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?

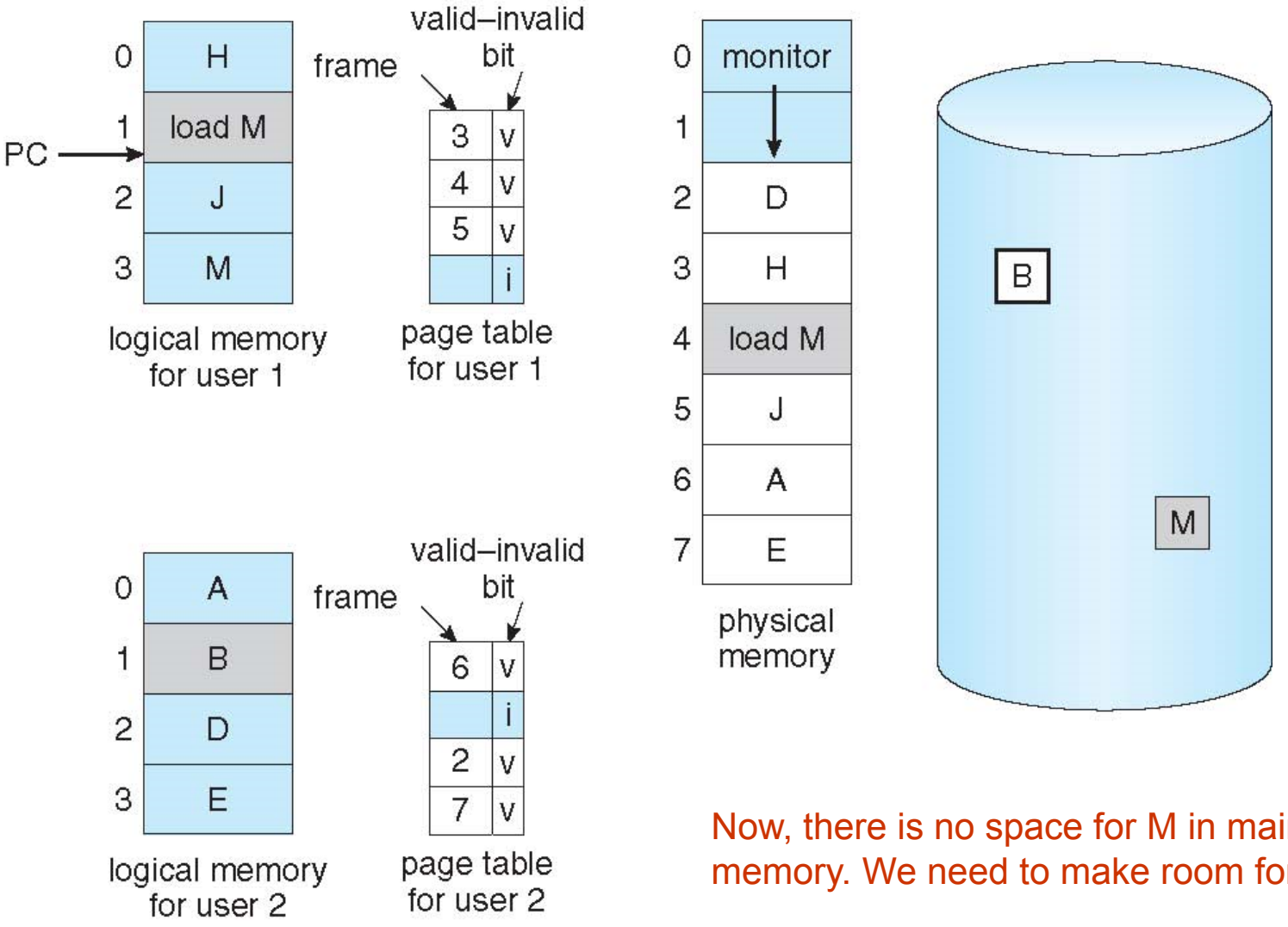
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times – This must be avoided

Page Replacement (Cont.)

- 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
  - i.e., the virtual memory system works thanks to page replacement measures

Need For Page Replacement



Now, there is no space for M in main memory. We need to make room for M.

Basic Page Replacement

Modify the page-fault service routine to include page replacement

1. Find the location of the desired page on disk

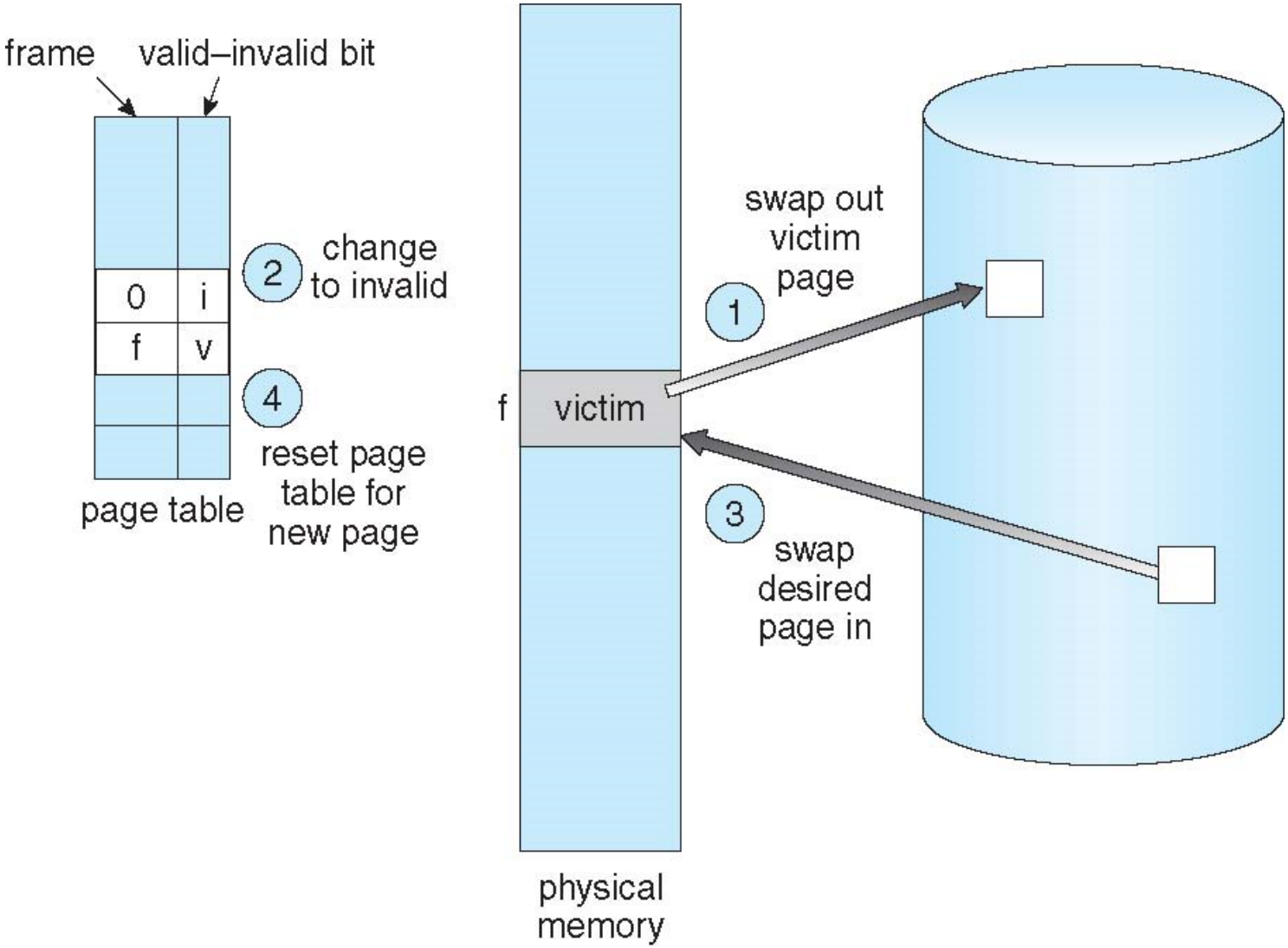
2. Find a free frame:
   a. If there is a free frame, use it
   b. If there is no free frame, use a page replacement algorithm to select a victim frame
   c. Write victim frame to disk if dirty

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

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm**
  - Determines which frames to replace
  - Want lowest page-fault rate on both first access and re-access

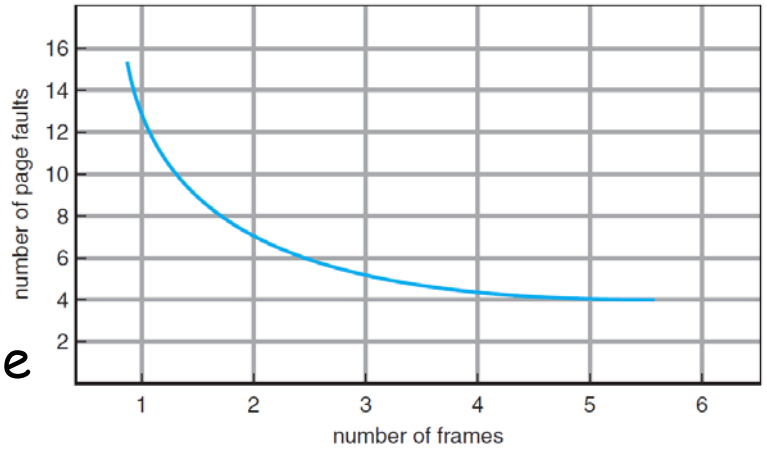


Figure 9.11 Graph of page faults versus number of frames.

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string based on two facts:
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
  - for a memory with three frames

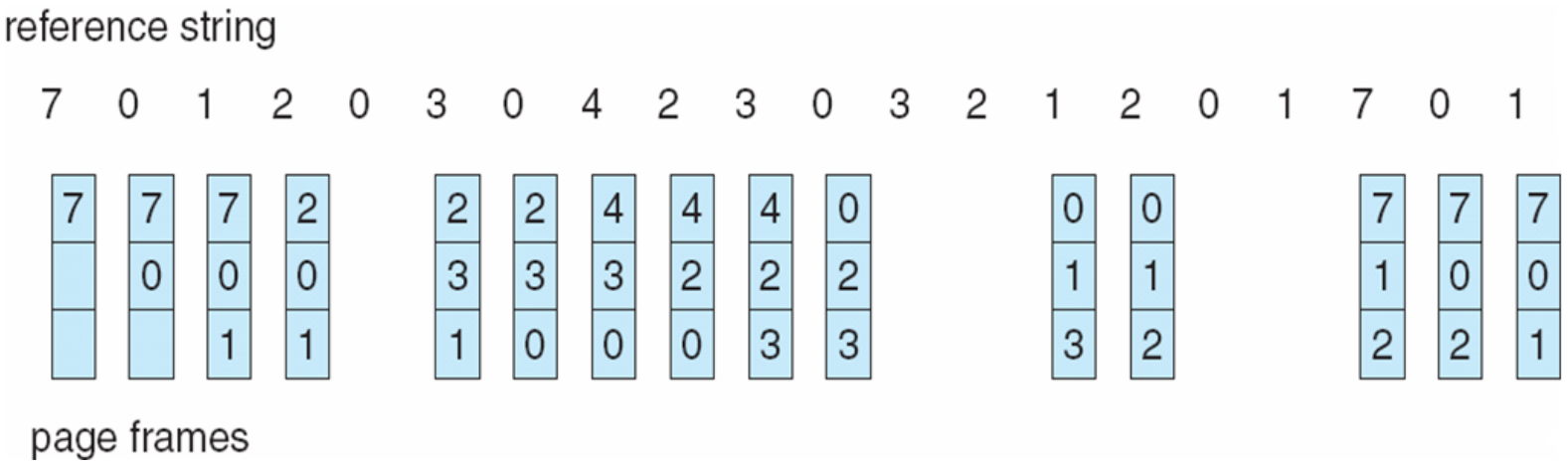
First-In-First-Out (FIFO) Algorithm

- Reference string:
  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)

1	7	2	4	0	7	
2	0	3	2	1	0	15 page faults (including the initial loading)
3	1	0	3	2	1	

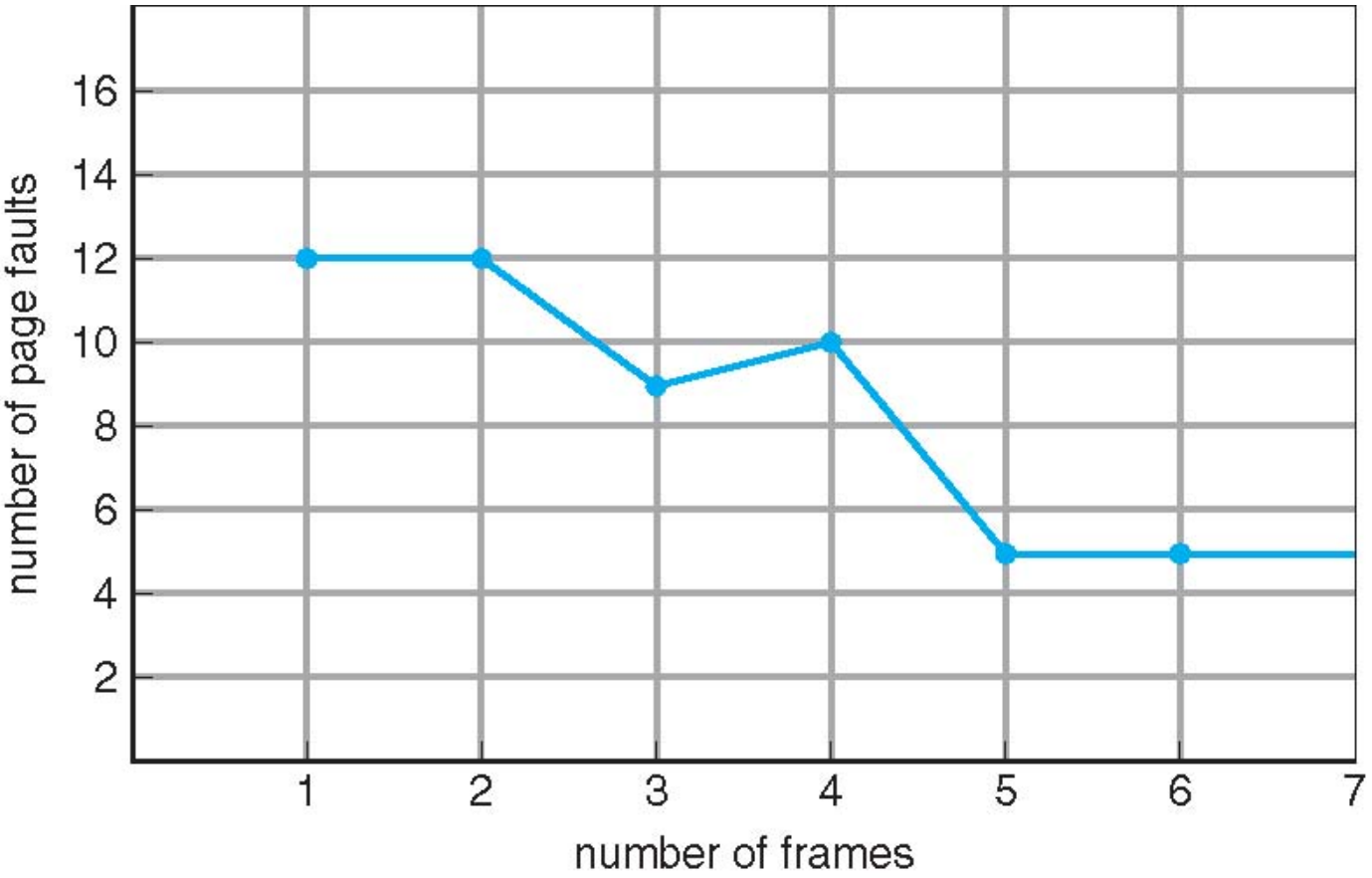
- Can vary by reference string: consider
  1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - Belady's Anomaly
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Page Replacement



Next slide: Examples of Belady's anomaly
(1) 1,2,3,4,1,2,5,1,2,3,4,5
3 frames: FFFFFFFGGFF {G: No fault}
4 frames: FFFFGGFFFFF
(2) 0,1,2,3,0,1,4,0,1,2,3,4
3 frames: FFFFFFFGGFFG (9 faults)
4 frames: FFFFGGFFFFFF (10 faults)

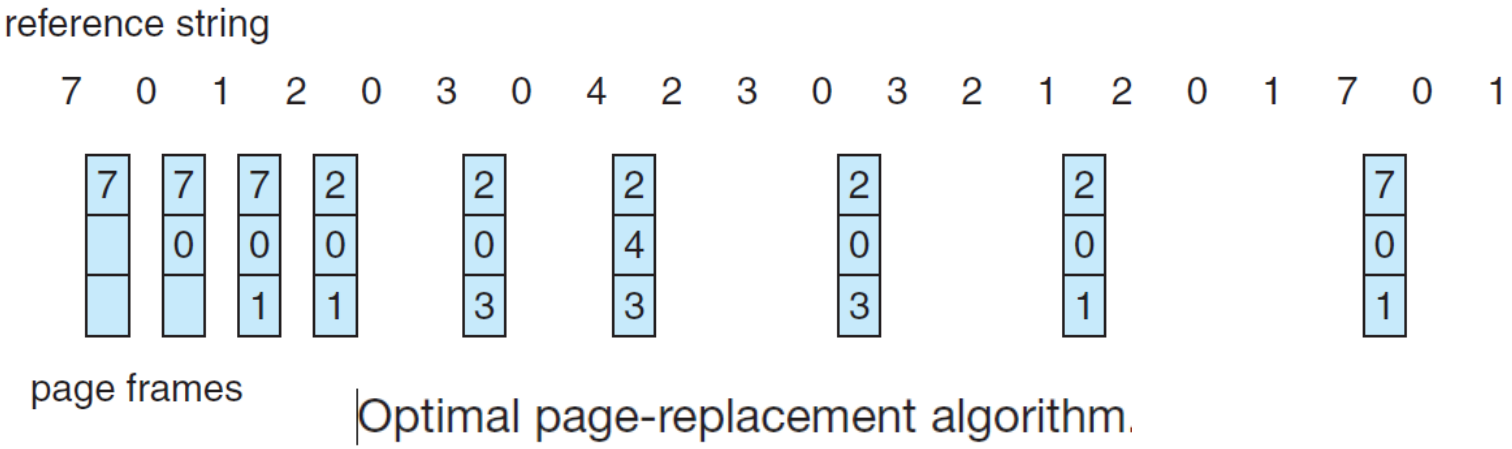
FIFO Illustrating Belady's Anomaly



Belady's anomaly: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases.

Optimal Algorithm

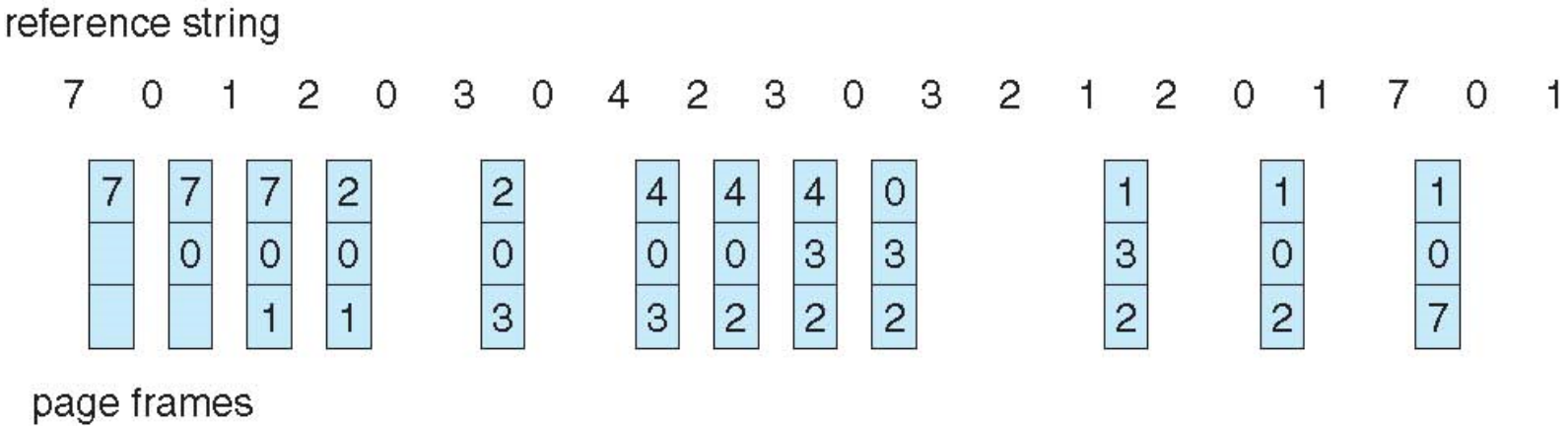
- Replace page that will not be used for longest period of time
  - 9 is optimal for the given example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



Optimal page-replacement algorithm.

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

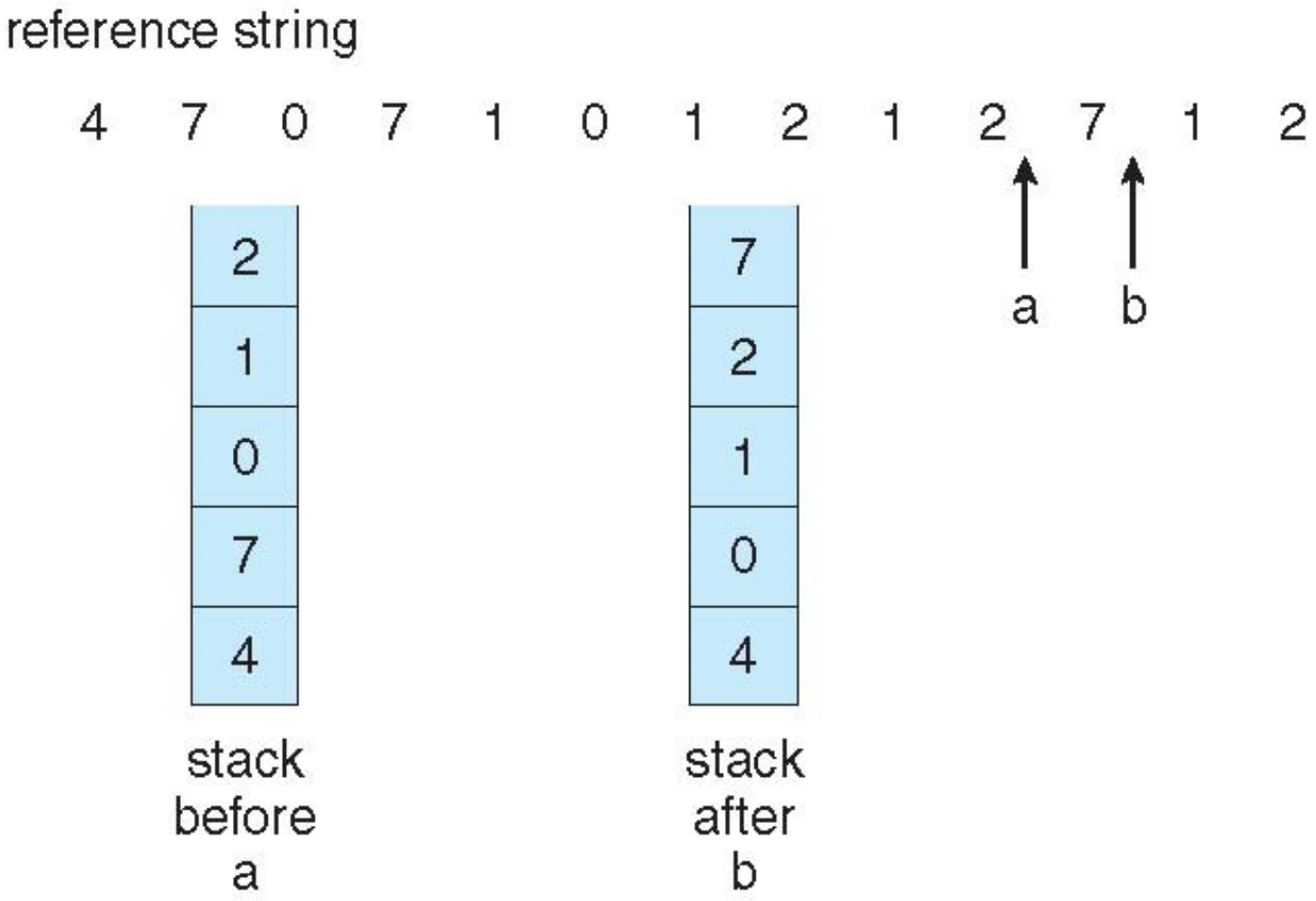


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

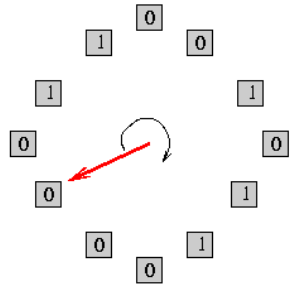
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search needed for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record The Most Recent Page References

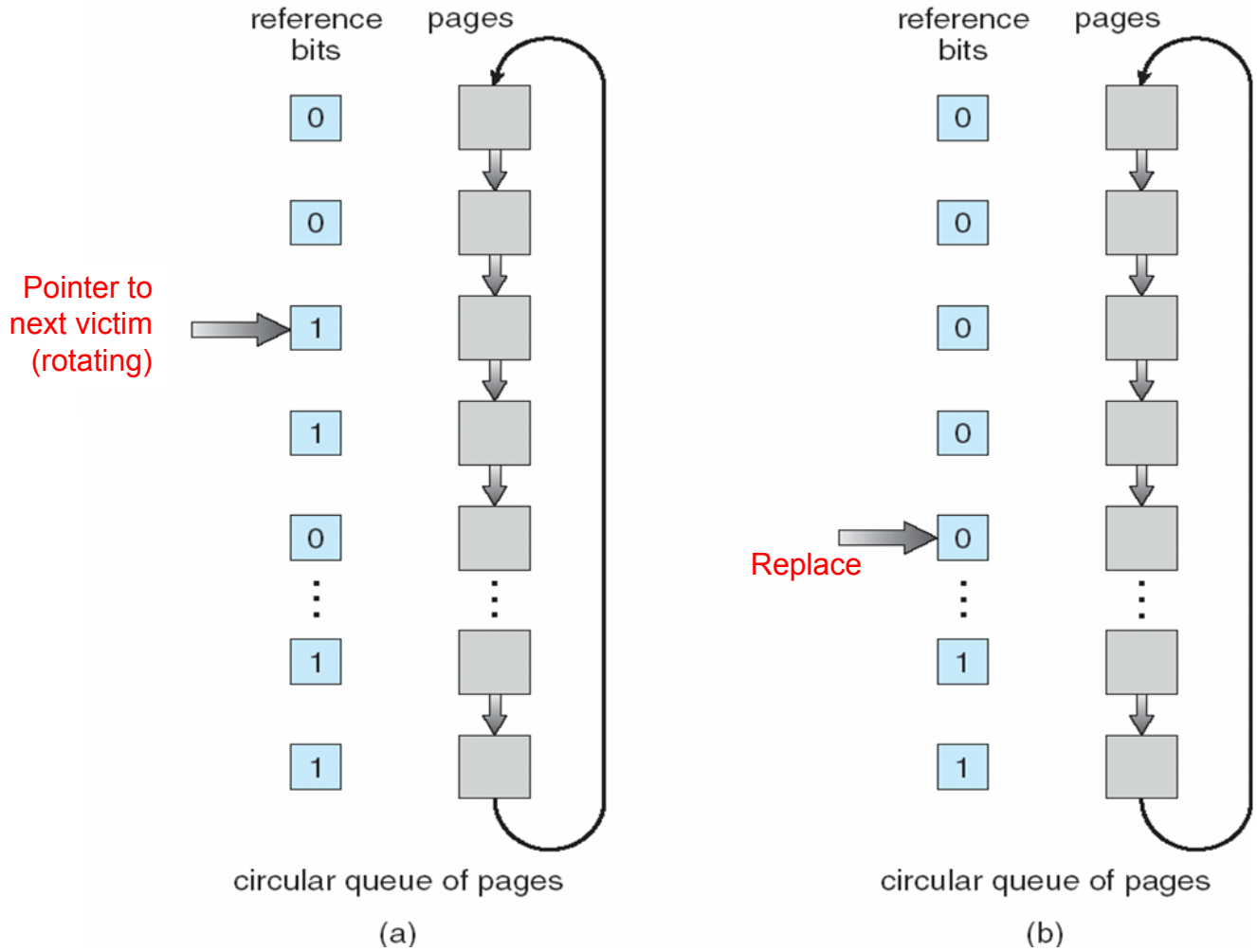


LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance (clock) algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - try to replace next page, 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
  - Not common
- **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
- Neither is often used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - First, read page into free frame and later select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Some OSs give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures.
  - This array is sometimes called the **raw disk**, and I/O to this array is termed raw I/O.
  - Raw I/O bypasses all the filesystem services, such as file I/O demand paging, file locking, etc

Allocation of Frames

- We must allocate at least a *minimum* number of frames
  - for performance reason
- The minimum number of frames
  - defined by computer architecture
  - Example: IBM 370 MVC instruction – 6 pages (worst case)
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle *from*
    - 2 pages to handle *to*
- *Maximum*, of course, is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
  - ✓ but, many variations

Allocation Algorithms

- Equal allocation (Fixed allocation)
  - For example, if there are 103 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some (leftover frames, 3 frames in this example) as free frame buffer pool
- Proportional allocation (Fixed allocation)
  - Allocate according to the size of process – fixed allocation
  - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

- Proportional allocation (Priority allocation)
  - Use priorities rather than size for the ration of frames

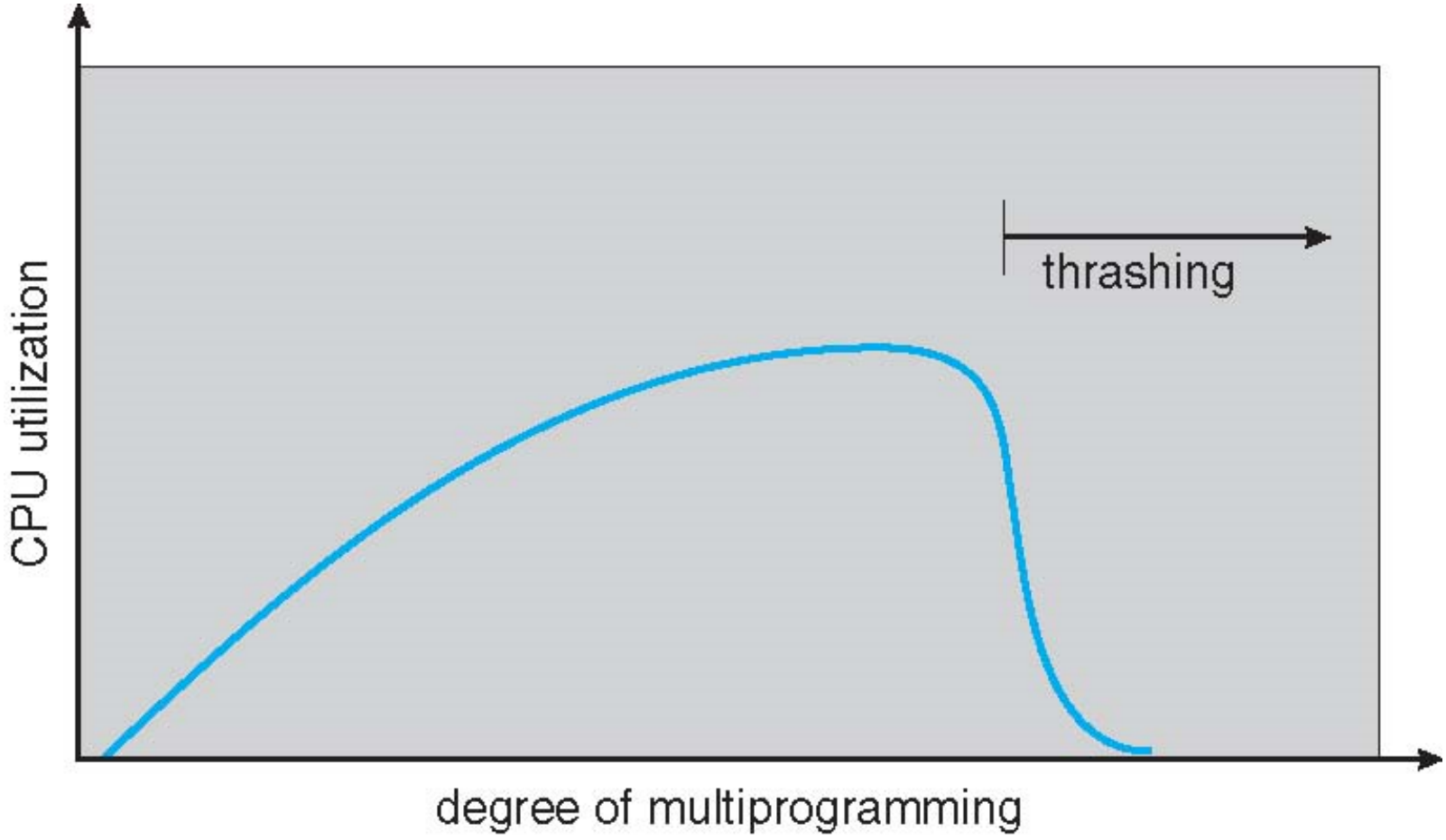
Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then execution times of a process can vary greatly
    - because it depends on the paging behavior of other processes
  - But greater throughput so more common
- Local replacement – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

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
    - Operating system thinking 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
  - Or, a process suffers from page faults continuously

Thrashing (Cont.)

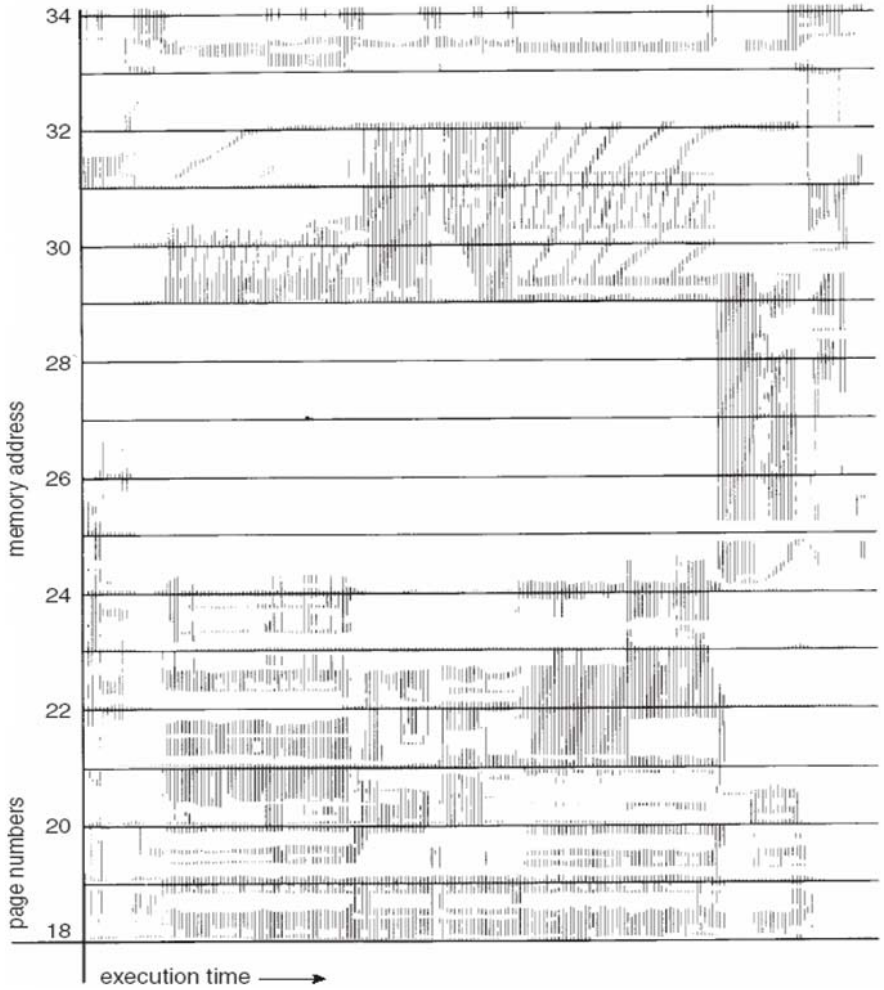


Demand Paging and Thrashing

- Why does demand 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
  - Limit effects by using local or priority page replacement algorithm

Local replacement (priority page replacement):
If one process starts thrashing, it can not steal frames from another process (which will cause the latter to thrash as well).

Locality In A Memory-Reference Pattern



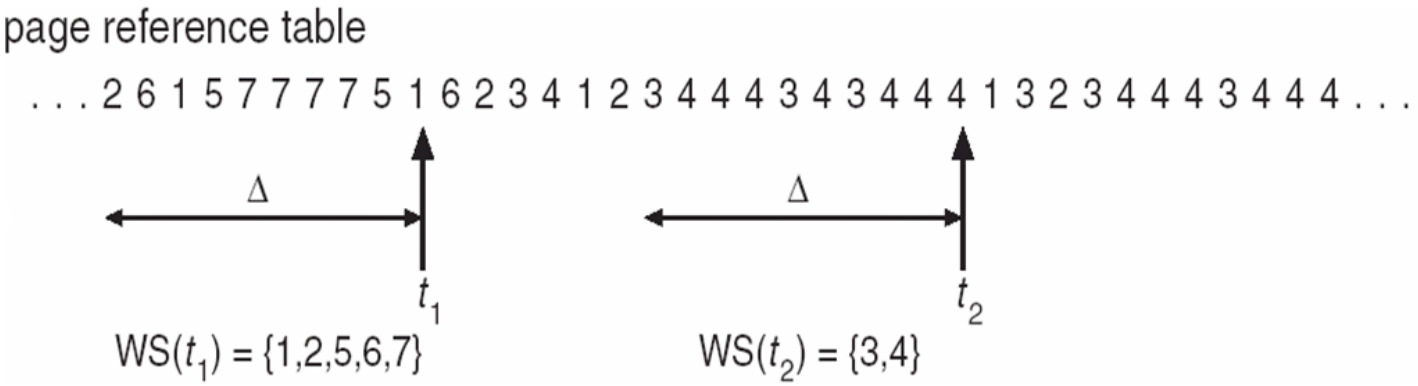
The locality model states that, as a process executes, it moves from one locality to another.

A locality is a set of pages that are actively used together.

We must allocate enough frames to accommodate the size of the current locality.

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSSi (working set of Process Pi) =
  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

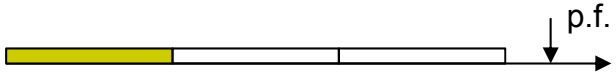


Working-Set Model (Cont.)

- $D = \Sigma\ WSSi \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
  - m: the total number of available frames
- Policy if $D > m$, then suspend or swap out one of the processes

Keeping Track of the Working Set

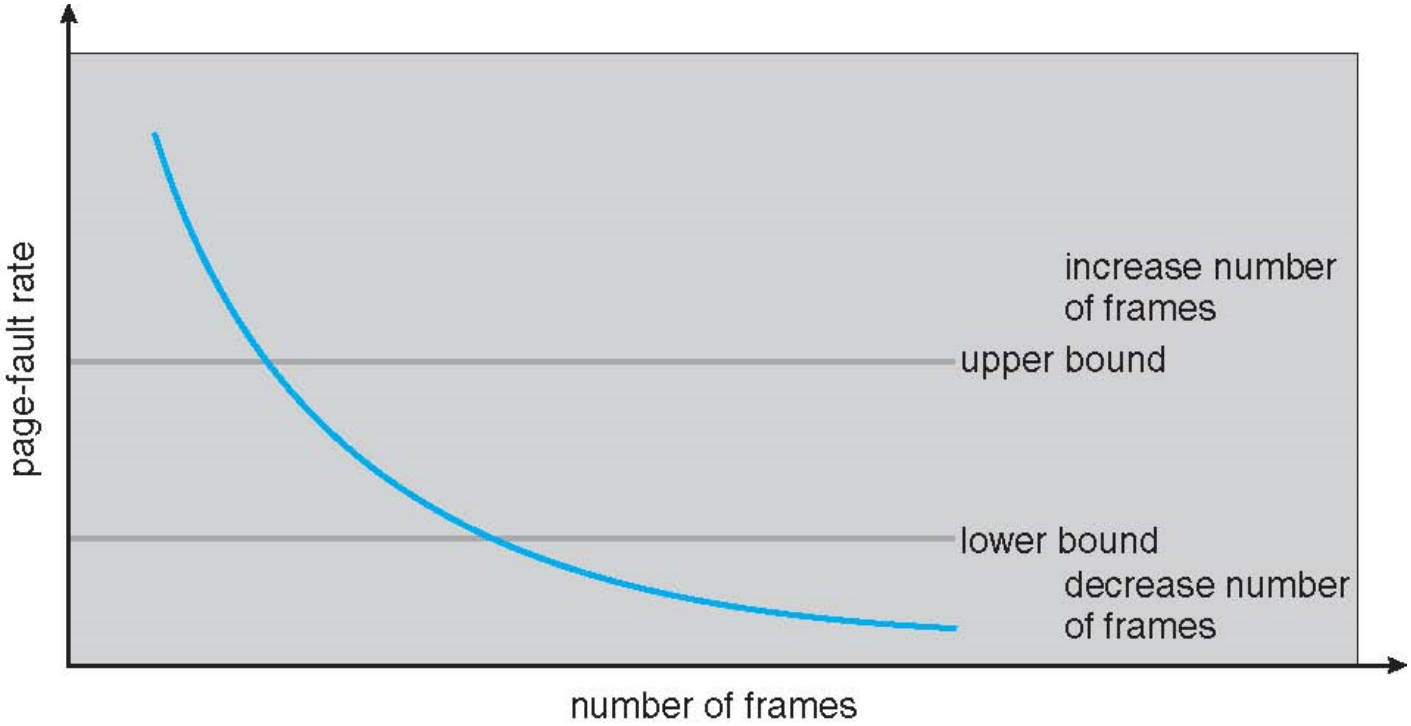
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page (in addition to ref. bit)
  - Whenever a timer interrupts, copy the values of all reference bits into one of the two bits (alternately) and set them to 0
  - If a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10000 to 15000 references.
  - That is, if one of the bits in memory = 1 $\Rightarrow$ page in working set



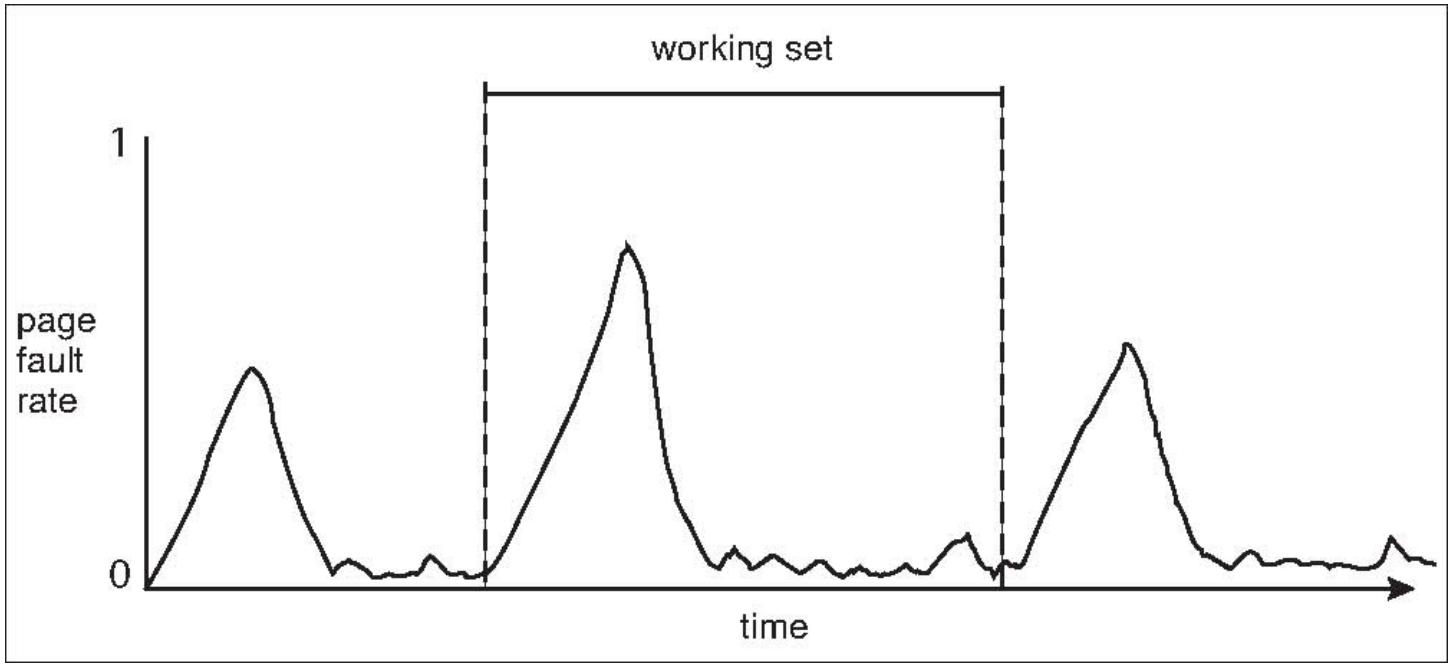
- Why is this not completely accurate?
  - Because we cannot tell where, within an interval of 5000, a reference occurred
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" page-fault frequency rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates



The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

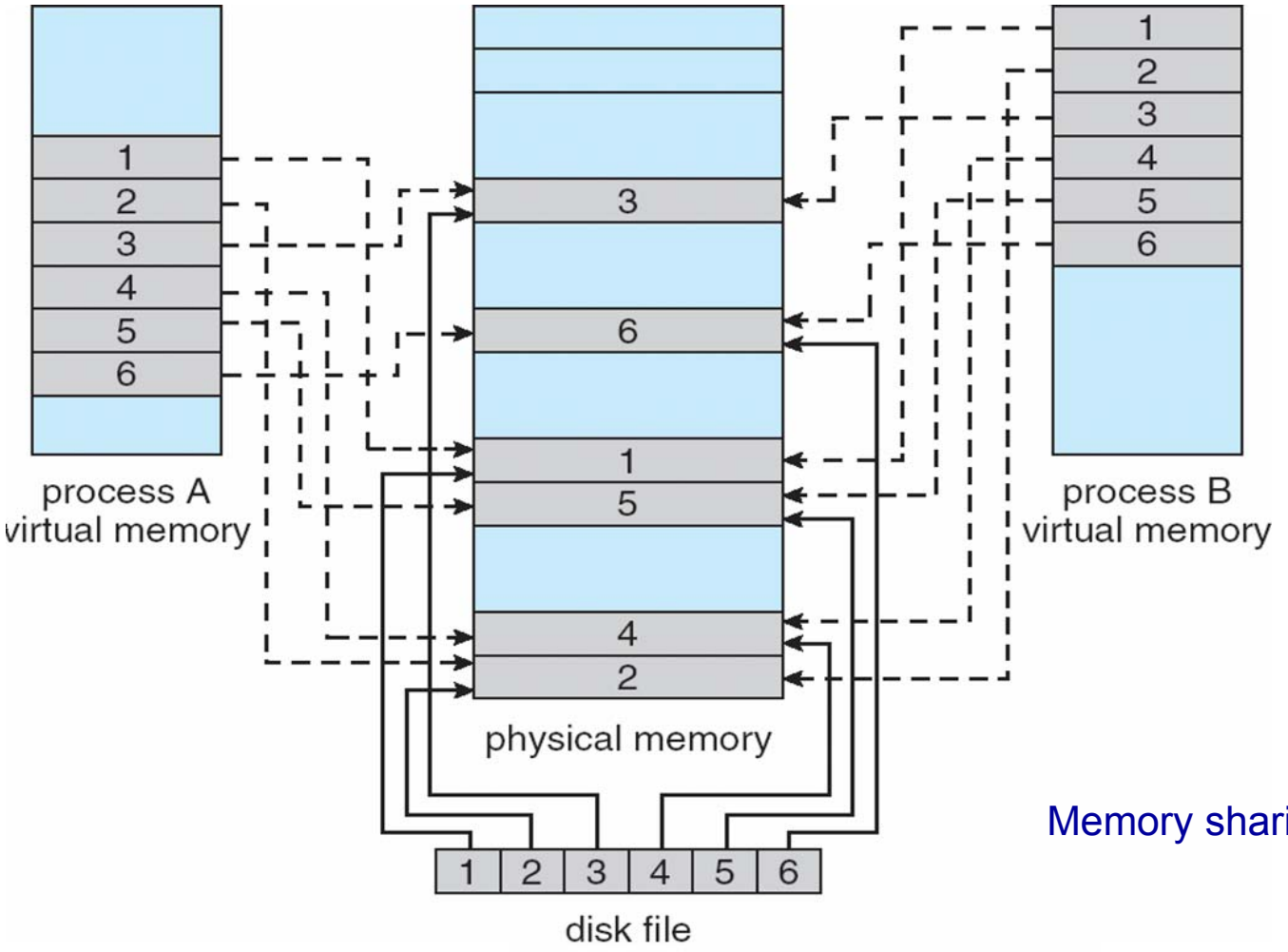
Memory-Mapped Files

- 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 file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

Memory-Mapped File Technique for All I/O

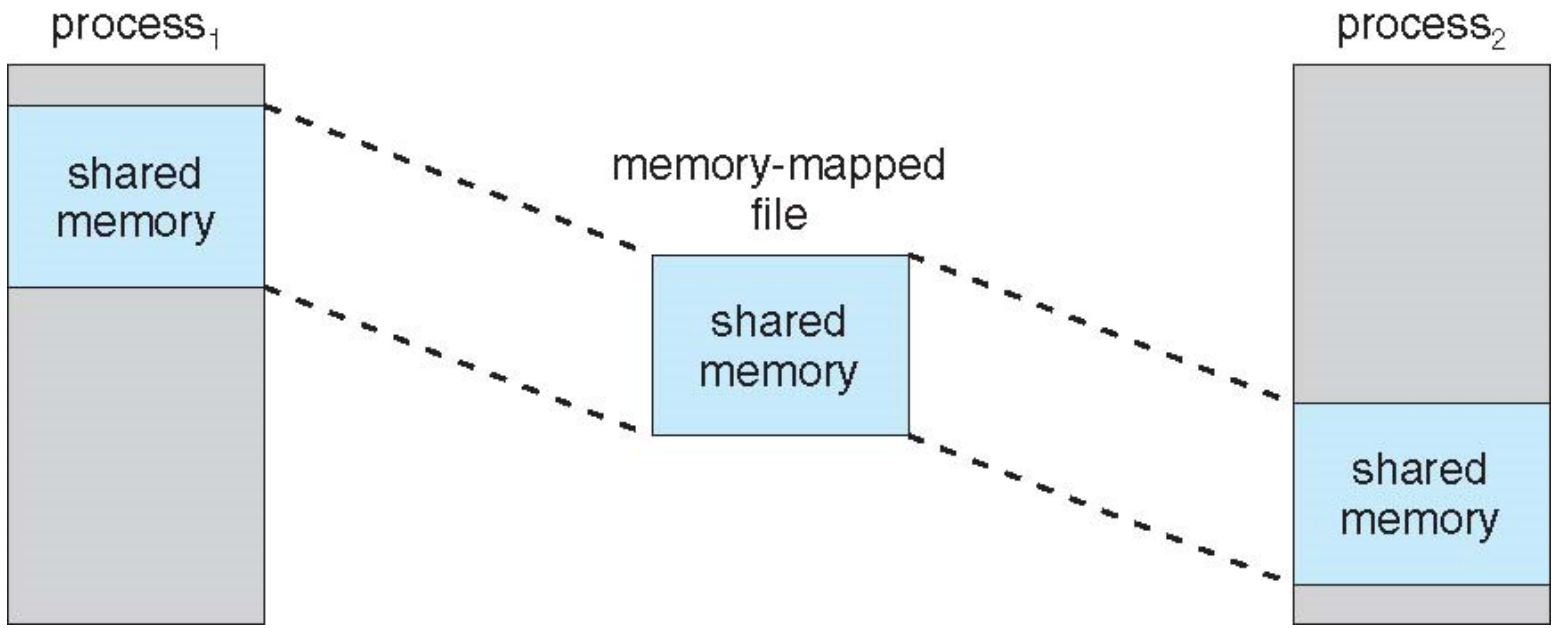
- Some OSes (such as Solaris) uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), memory-map the file, anyway
  - But map file into kernel address space
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory Mapped Files



Memory sharing

Memory-Mapped Shared Memory in Windows



File mapping

Allocating Kernel Memory

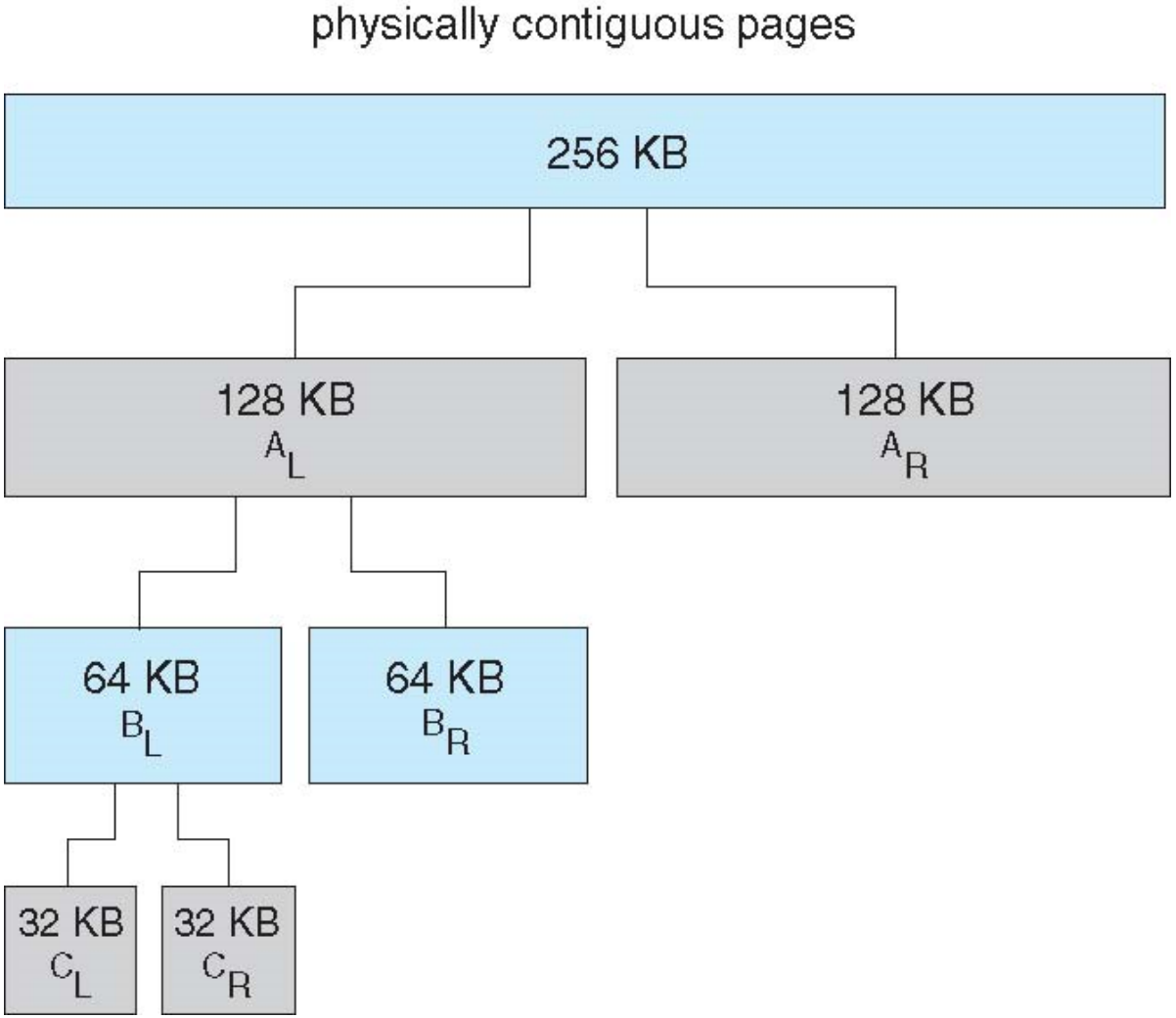
- Treated differently from user memory

- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- E.g., assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_r of 128KB each (next slide)
    - One further divided into B_L and B_R of 64KB
    - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

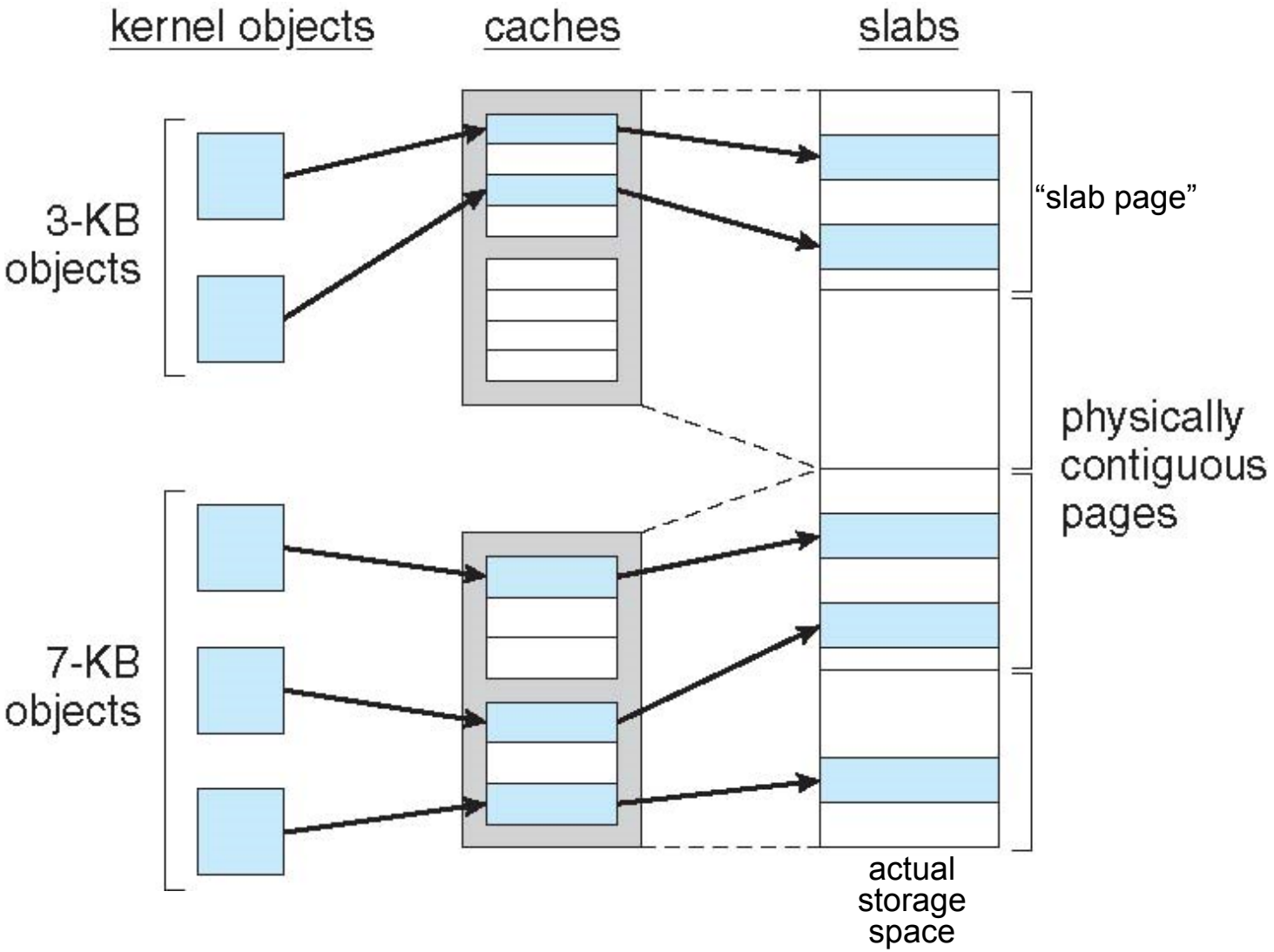
Buddy System Allocator



Slab Allocator

- Alternate strategy to buddy allocator
- **Slab** occupies one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- 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 include no fragmentation, fast memory request satisfaction

Slab Allocation



Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occur at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume *s* pages are prepaged and a fraction *a* of the pages is used
    - Is cost of *s* * *a* save pages faults > or < than the cost of prepaging *s* * *(1- a)* unnecessary pages?
    - *a* near zero ⇒ prepaging loses

Other Issues – Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- 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 the opportunity to use them without an increase in fragmentation

Other Issues – Program Structure

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

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

    128 x 128 = 16,384 page faults

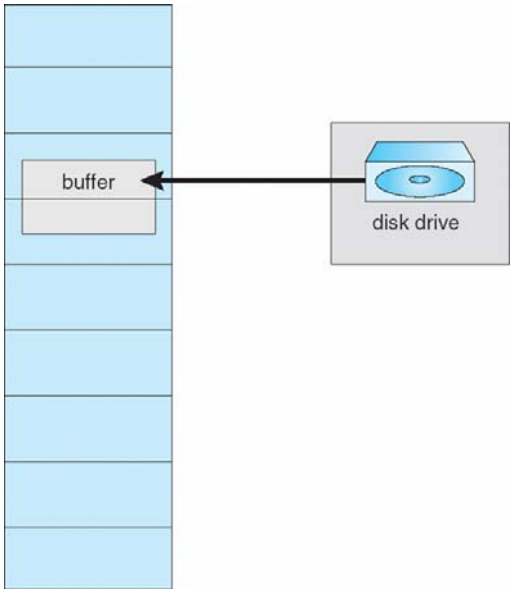
  - Program 2

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

    128 page faults

Other Issues – I/O interlock

- Problem: The process requesting I/O may be swapped out
- Two common solutions to this problem.
  a. Never execute I/O directly to user memory. Instead, use system memory as intermediary between I/O device and user memory. Extra copying may result in unacceptably high overhead.
  b. Allow pages to be locked into memory so that they cannot be selected for replacement. → **I/O Interlock**



The reason why frames used for I/O must be in memory

OS Example – *Windows XP*

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Summary

- 가상기억(virtual memory)
  - 프로그래머가 논리주소공간에 프로그램하도록 함
- 디맨드 페이징(demand paging)
  - 해당 페이지가 필요하게 될 때 가져옴. 페이지 폴트
  - 필요한 구조, 절차, 성능분석
  - 페이지 폴트 처리 후 다시 실행될 때, 명령어의 재시작 문제
- Copy-on-write
  - 두 프로세스가 한 페이지를 공유하다가, 쓰기작업시에 새로운 페이지를 생성
  - 복사에 소요되는 시간과 공간 절약
- 페이지의 재배치(replacement)
  - 자신에게 할당된 프레임을 다 사용하고 있을 때, 새로운 페이지의 반입을 위해 희생양을 선택하여 퇴출
  - 절차
  - 페이지 재배치 알고리즘: FIFO, LRU, optimal, page-buffering
  - LRU와 구현: reference bit, second chance (clock) algorithm, 등

- Virtual memory
  - allow the programmer to write a program in logical address space
- Demand paging
  - get a page when needed. page fault
  - structure, procedure, performance
  - instruction restart: when the control is returned to the program after page fault processing
- Copy-on-write
  - A page is shared by two processes until write operations start
  - save time & space required for copy
- Page replacement
  - When all frames allocated are used, victim frame must be selected to get a new page
  - processing steps
  - replacement algorithm: FIFO, LRU, optimal, page-buffering
  - LRU and implementation: reference bit, second chance (clock) algorithm, etc.

Summary (Cont.)

- 페이지 프레임의 할당
  - 각 프로세스는 최소한의 프레임 [갯수] 필요
  - 방법: fixed allocation, priority allocation, global vs. local
- 쓰래싱(thrashing)
  - what and why?
  - locality 이슈: locality에 맞는 할당. locality가 옮겨다님
  - working set model: 일정시간 동안의 페이지 참조를 고려
  - page-fault frequency의 추적
- memory-mapped files
  - 메모리에 파일을 옮겨 사용
- 커널 메모리의 할당
  - 커널 데이터를 위한 효율적 이용
- 기타 고려사항
  - 페이지 크기, TLB reach, I/O interlock, 등
- 사례: Windows XP

- Allocation of frames
  - a process needs min number of frames
  - approaches: fixed allocation, priority allocation, global vs. local
- Thrashing
  - what and why?
  - locality issue: allocation for locality. moves from one locality to another
  - working set model: consider the page references during the given time interval
  - page-fault frequency
- memory-mapped files
  - maps a file to the memory
- Kernel memory allocation
  - efficient usage for kernel data
- Other considerations
  - page size, TLB reach, I/O interlock, etc.
- Examples: Windows XP