

Operating Systems

[10. Virtual Memory]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

Objectives

- ❑ Define virtual memory and describe its benefits
- ❑ Illustrate how pages are loaded into memory using demand paging
- ❑ Apply the FIFO, optimal, and LRU page-replacement algorithms
- ❑ Describe the working set of a process, and explain how it is related to program locality
- ❑ Describe how Linux, Windows 10, and Solaris manage virtual memory

Outline

- ❑ **Background**
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Background (1/2)

❑ Basic requirement

- The instructions being executed must be in physical memory

❑ One approach

- Place the entire logical address space in physical memory
 - Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer
- Limit the size of a program to the size of physical memory

❑ In fact

- In many cases, the entire program is not needed
 - Error handling code, large data structures, rarely-used options and features 😊
- Even if the entire program is needed, it may not all be needed at the same time

Background (2/2)

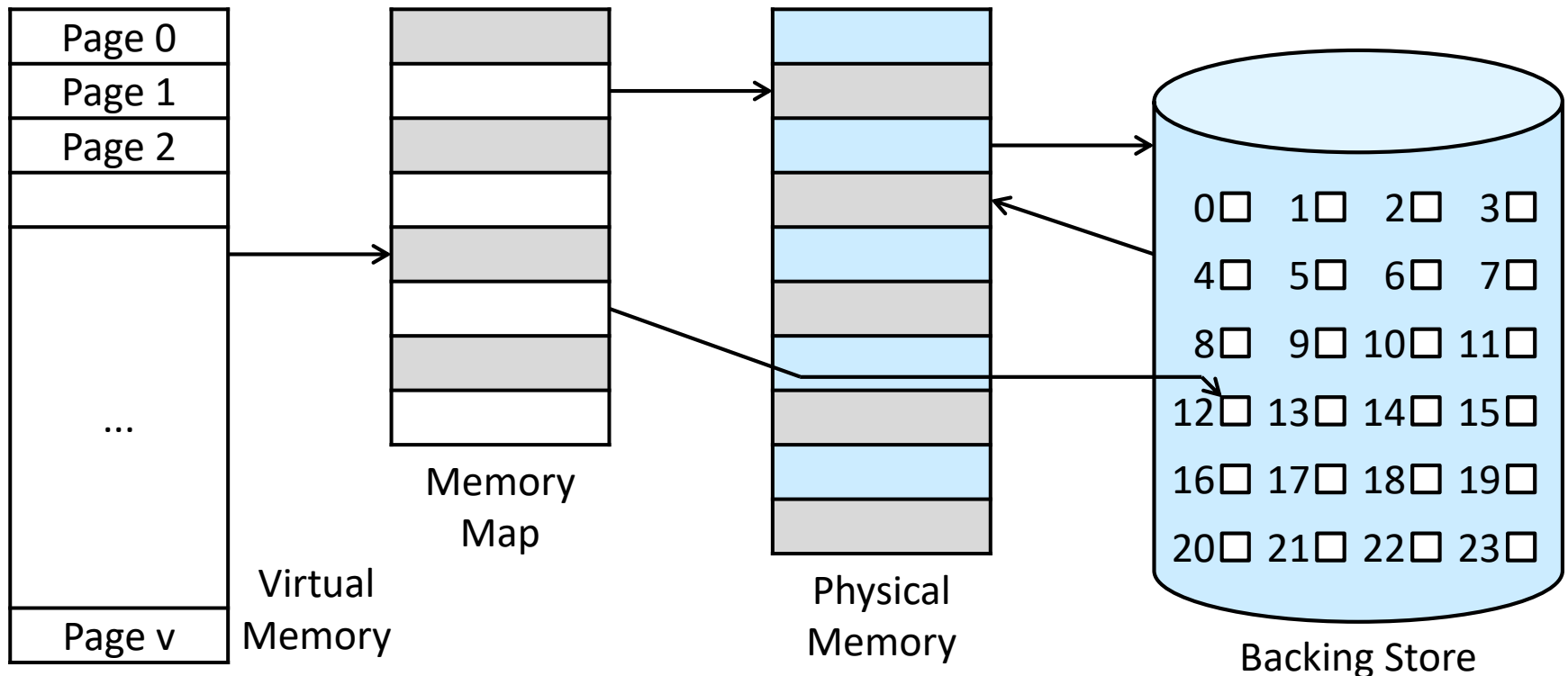
❑ Benefits of executing a program only partially in memory

- A program is no longer constrained by the amount of physical memory
- More programs can be run at the same time
 - Increase in CPU utilization and throughput
 - No decrease in response time or turnaround time
- Less I/O is needed to load or swap portions of programs into memory
 - Each program would run faster

Virtual Memory

❑ Separation of logical memory as perceived by developers from physical memory

- Allow an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available
- Make the task of programming much easier



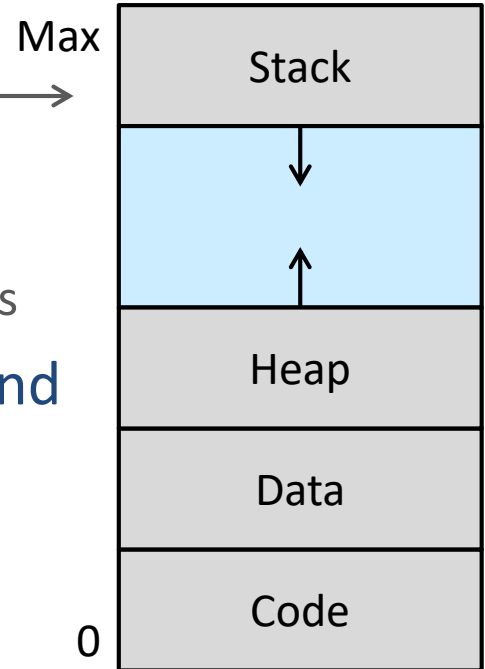
Virtual Memory Space

❑ Logical (or virtual) view of how a process is stored in memory

- A process begins at a certain logical address and exists in contiguous memory
- The physical page frames assigned to the process may not be contiguous
- The MMU maps logical pages to physical page frames

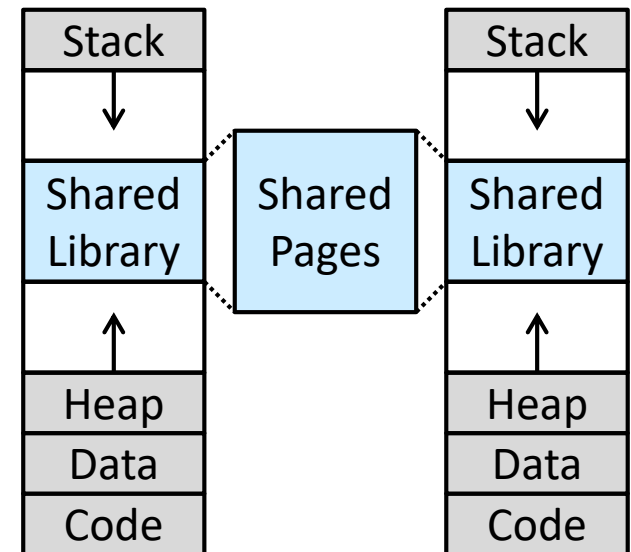
❑ The blank space (or hole) between the heap and the stack is part of the virtual address space

- It requires actual physical pages only if the heap or stack grows
- Virtual address spaces with holes are known as **sparse** address spaces
 - Using a sparse address space is beneficial because the holes can be filled if the stack or heap grows or if we wish to dynamically link libraries



Shared Library Using Virtual Memory

- ❑ Virtual memory also allows files and memory to be shared by two or more processes through page sharing
 - Processes share system libraries through mapping the shared object into a virtual address space
 - The actual physical pages where the libraries (typically read-only) reside are shared
 - Processes share memory
 - One process creates a region of memory and shares it with another process
 - Processes consider it part of their virtual address space, yet the actual physical pages are shared
 - Pages are shared with **fork()** system call
 - Speed up process creation



Outline

- ❑ Background

- ❑ **Demand Paging**

 - Basic Concepts, Free-Frame List, Performance of Demand Paging

- ❑ Copy-on-Write

- ❑ Page Replacement

- ❑ Allocation of Frames

- ❑ Thrashing

- ❑ Memory Compression

- ❑ Allocating Kernel Memory

- ❑ Other Considerations

- ❑ Operating-System Examples

Demand Paging

❑ One approach

- Program starts with a list of available options that the user is to select

❑ Another approach: **demand paging**

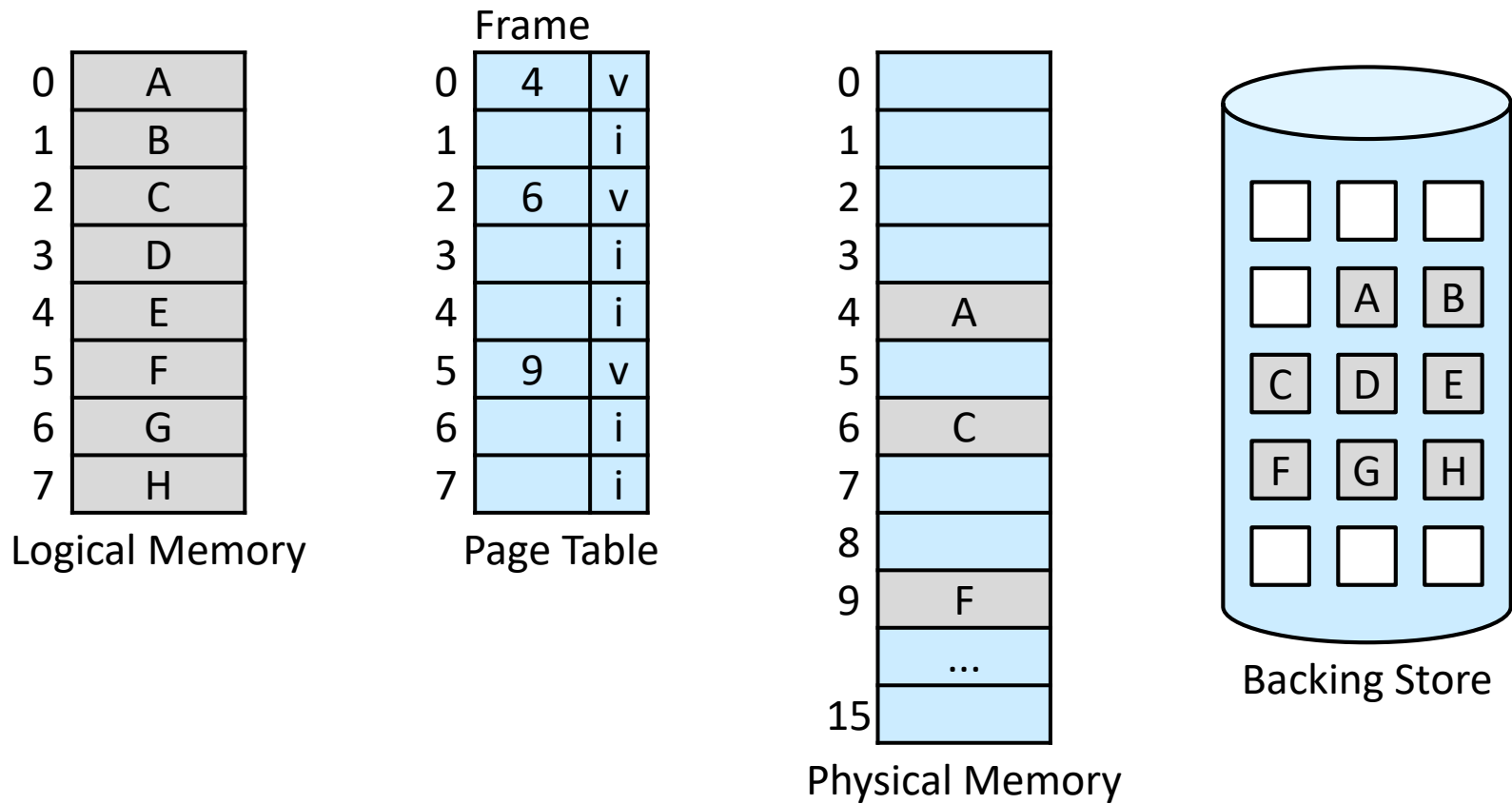
- Pages are loaded only when they are **demanded** during execution
- Pages that are never accessed are never loaded into physical memory
- It is commonly used in virtual memory systems
- Memory is used more efficiently

Outline

- ❑ Background
- ❑ **Demand Paging**
 - **Basic Concepts**, Free-Frame List, Performance of Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

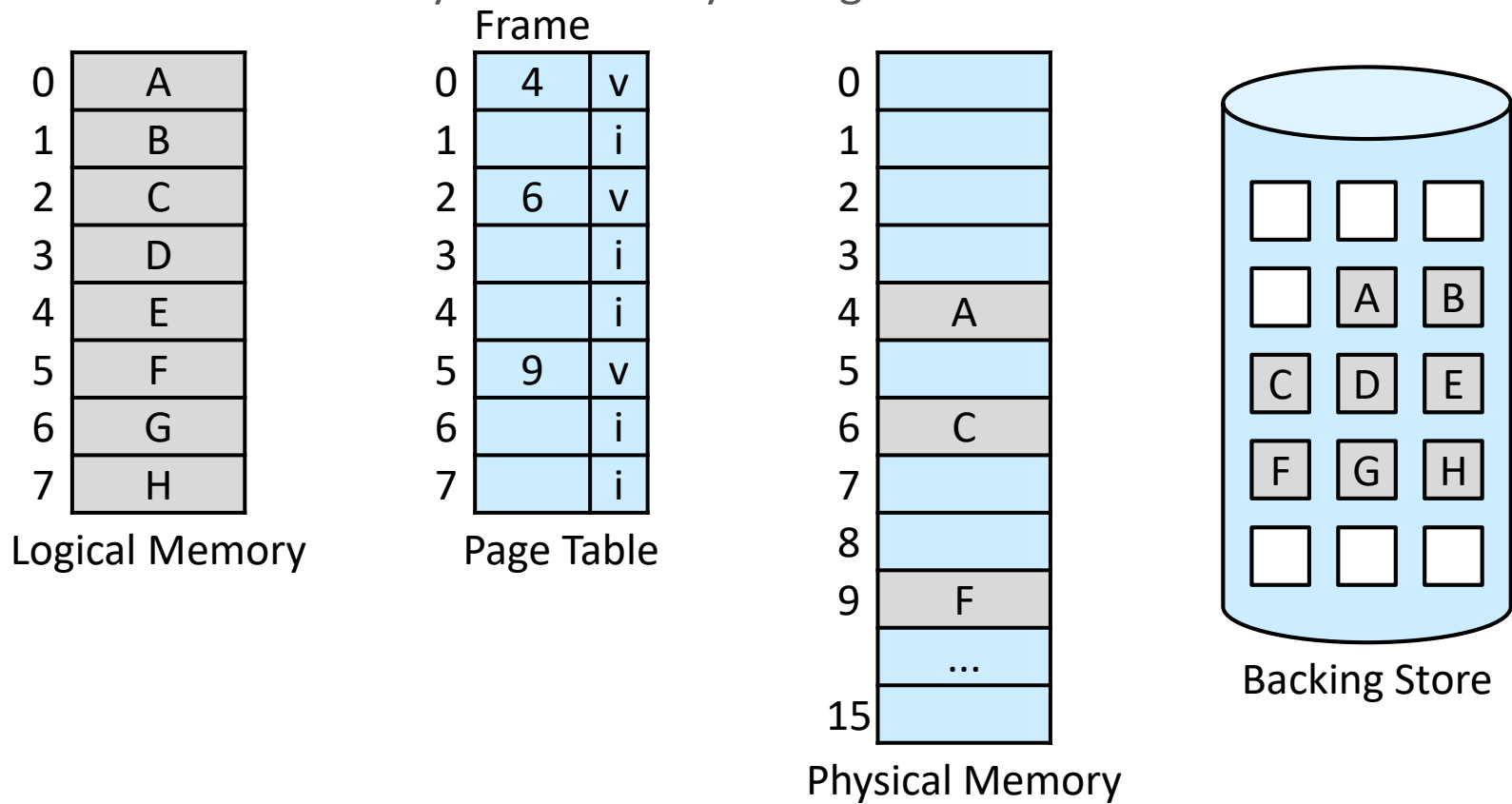
Basic Concepts (1/2)

- ❑ While a process is executing, some pages are in memory, and some pages are in secondary storage
 - We need some form of hardware support to distinguish between them
 - The valid-invalid bit scheme described before can be used



Basic Concepts (2/2)

- ❑ If the bit is valid, the associated page is legal and in memory
- ❑ If the bit is invalid, the page is either
 - Not valid, or
 - Valid but currently in secondary storage



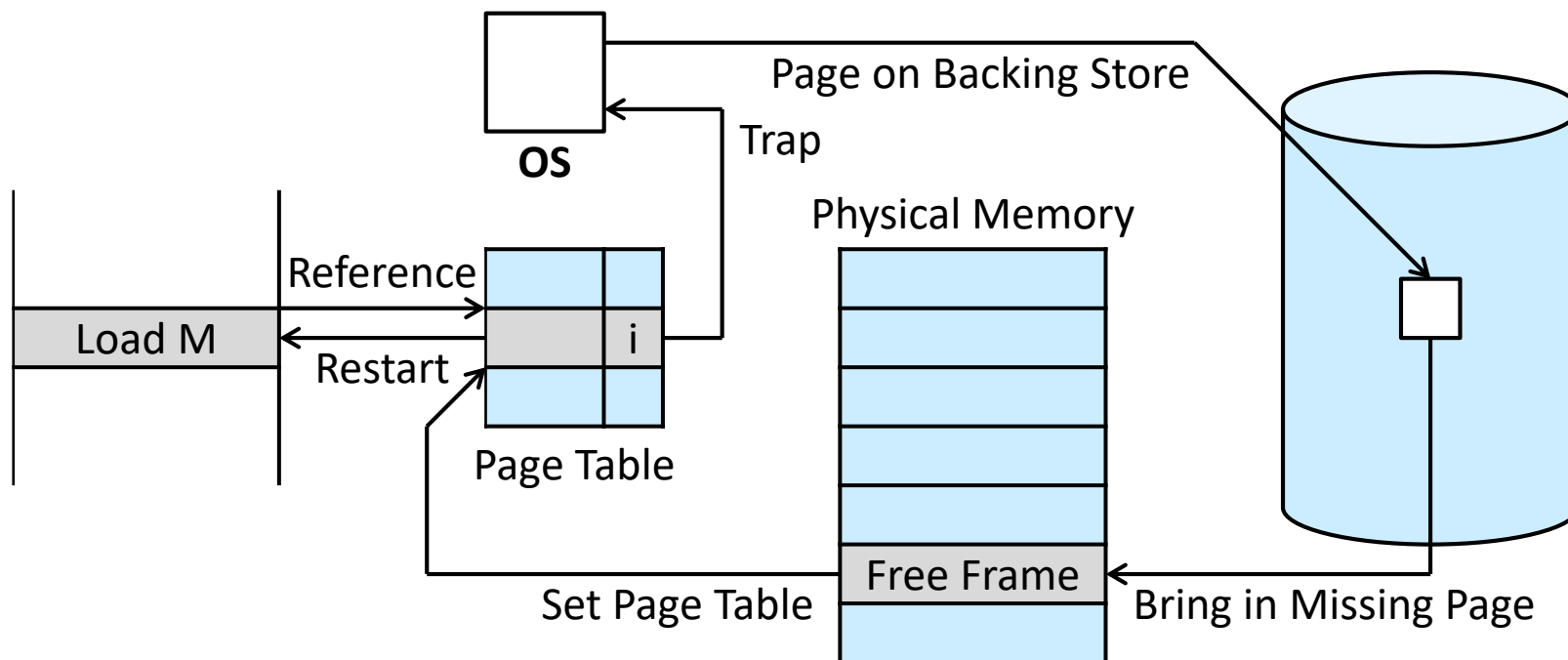
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 1. Check an internal (another) table (usually kept with the process control block) to determine whether the reference was valid or invalid



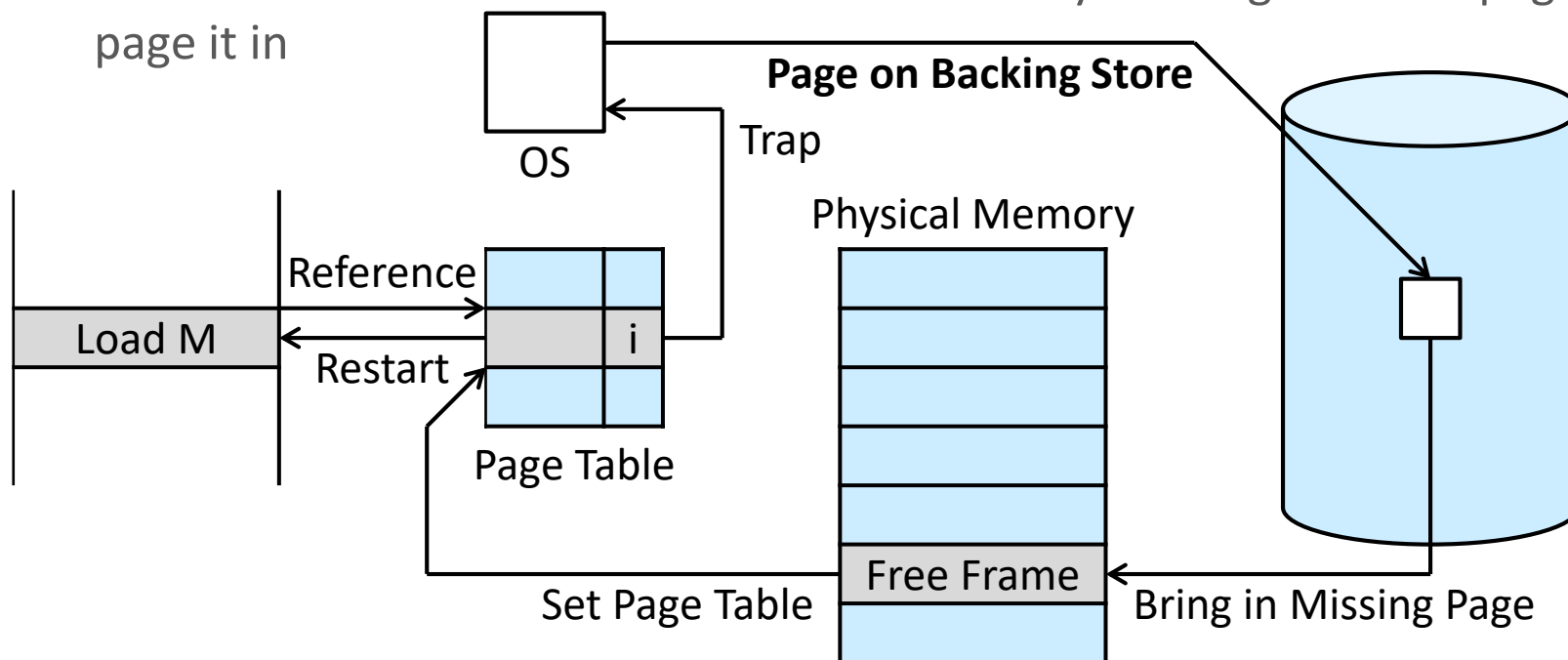
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 2A. If the reference was invalid, terminate the process
- 2B. If the reference was valid but we have not yet brought in that page, page it in



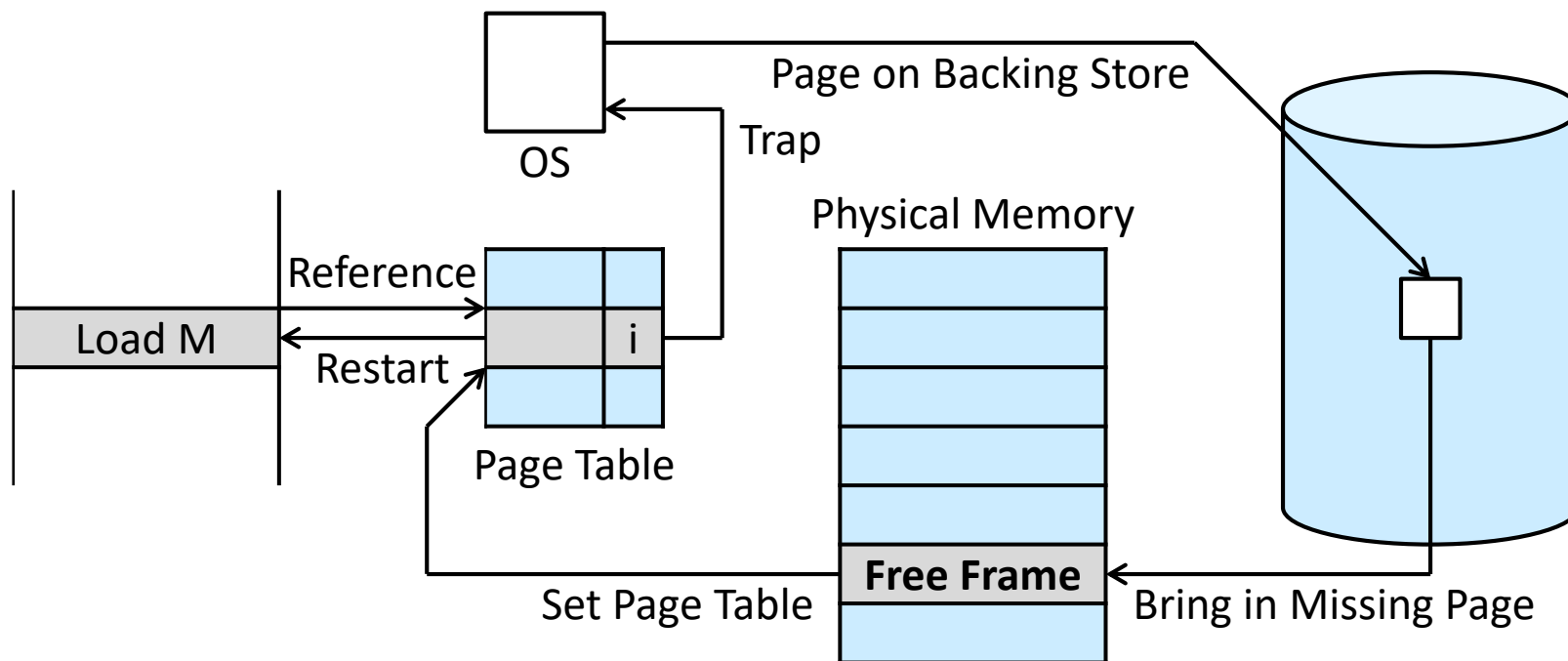
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 3. Find a free frame (by taking one from the free-frame list, for example)
 - The free-frame list will be introduced later



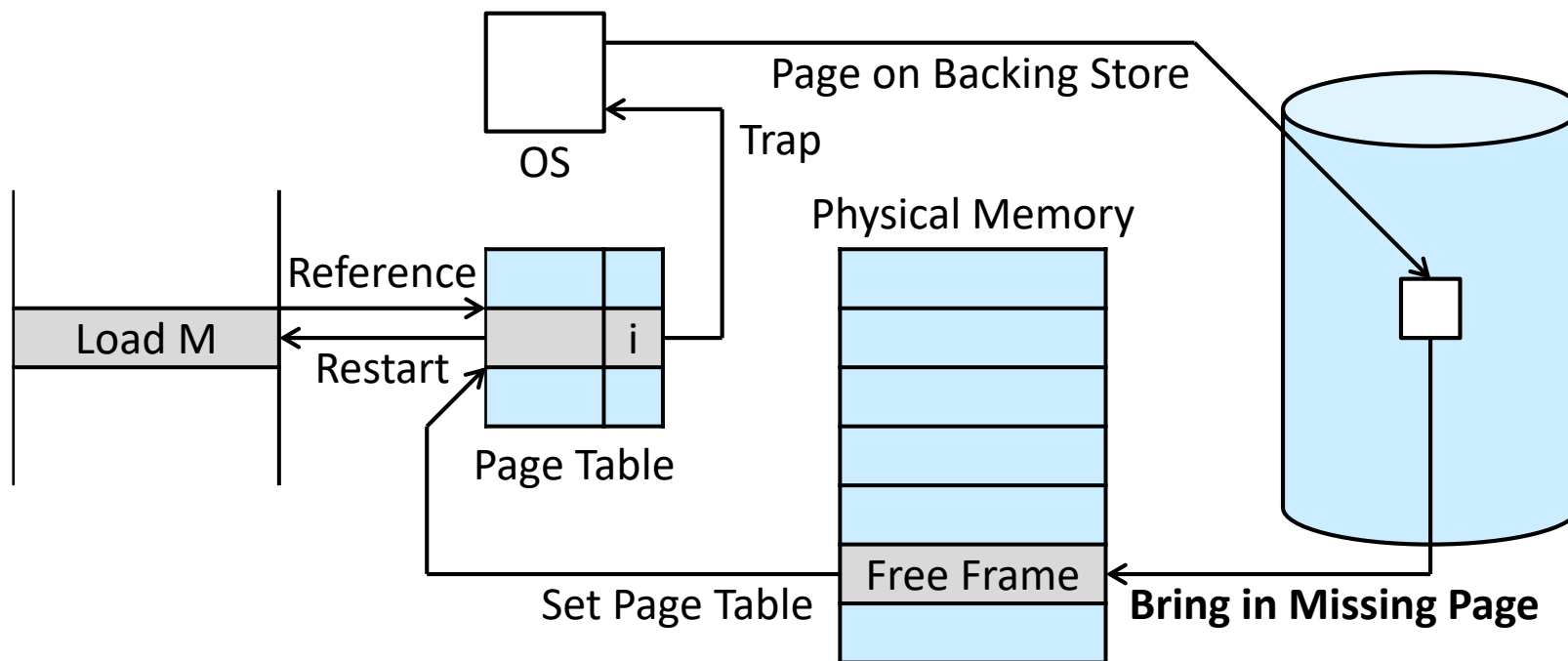
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 4. Schedule a secondary storage operation to read the desired page into the newly allocated frame



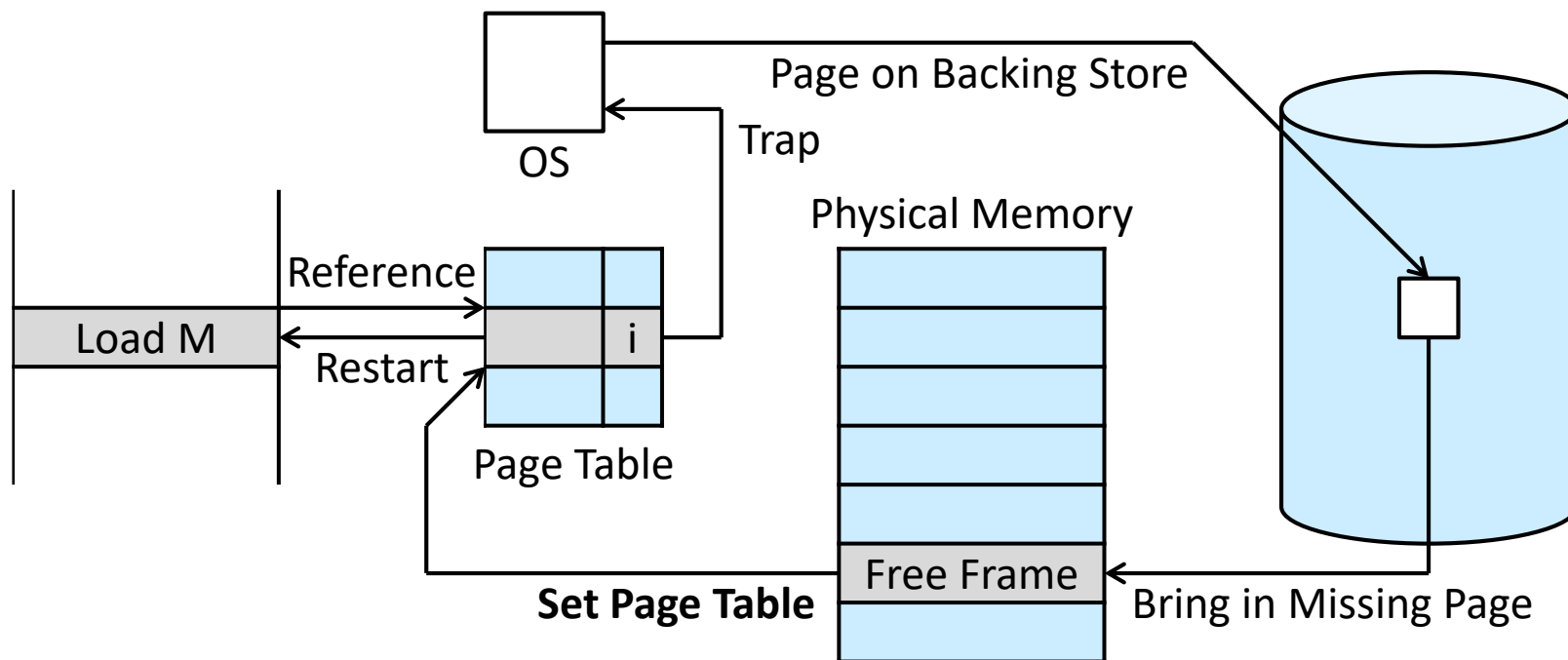
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 5. Modify the internal table kept with the process and the page table to indicate that the page is now in memory



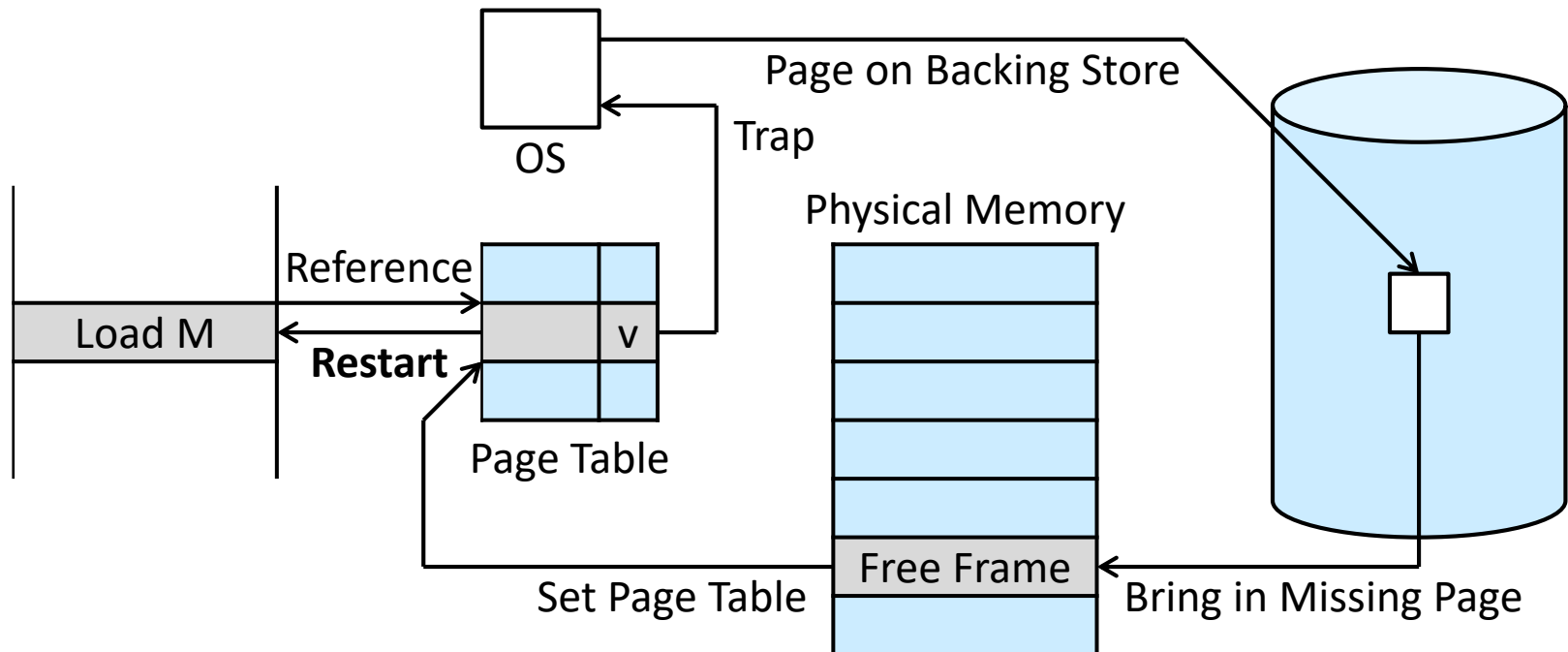
Page Fault Handling

❑ Access to a page marked invalid causes a page fault

- The paging hardware will notice that the invalid bit is set and cause a trap to the operating system

❑ Procedure for handling this page fault

- 6. Restart the instruction that was interrupted by the trap



Aspects of Demand Paging

❑ Pure demand paging

- Never bring a page into memory until it is required
- Start executing a process with no page in memory
- Set the instruction pointer to the first instruction (fault immediately)

❑ Locality of reference

- Theoretically, some programs access several new pages of memory with each instruction execution
 - Example: one page for the instruction and many for data
 - Unacceptable system performance
- Analysis of running processes shows that this behavior is exceedingly unlikely

❑ Hardware support (same as paging and swapping)

- Page table with a valid-invalid bit (or other protection bits)
- Secondary memory (swap device with swap space)

Instruction Restart (1/2)

❑ A page fault may occur at any memory reference

- Occur on the instruction fetch: fetch the instruction again
- Occur on the operand fetch: fetch the instruction again and then fetch the operand

❑ Example

- Fetch and decode the instruction (ADD)
- Fetch A
- Fetch B
- Add A and B
- Store the sum in C
 - If C is in a page not currently in memory, we will have to get the desired page, bring it in, correct the page table, and restart the instruction (all steps above)

❑ Performance is not a major concern

- There is not much repeated work (less than one complete instruction)

Instruction Restart (2/2)

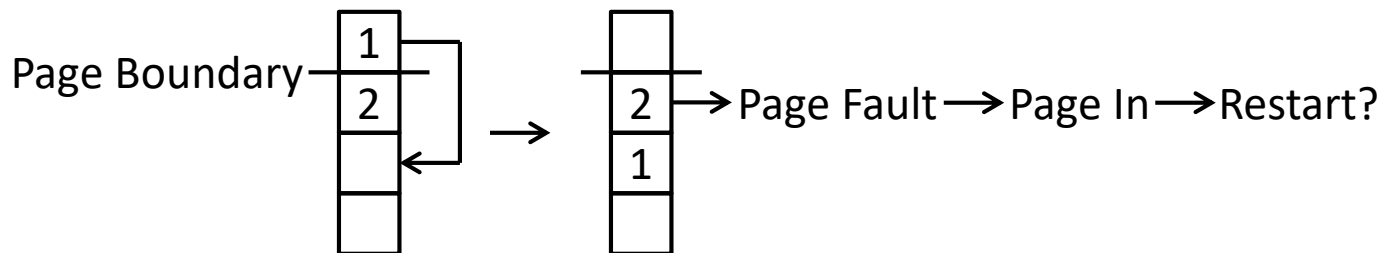
❑ Major difficulty: one instruction modifies different locations

➤ Example: an instruction moves some bytes from one location to another (possibly overlapping) location

- If either block (source or destination) straddles a page boundary, a page fault may occur after the move is partially done
- If the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction

➤ Solutions

- Microcode attempts to access both ends of both blocks and triggers page faults (if any) before anything is modified
- Temporary registers hold the values of overwritten locations, and, if there is a page fault, all the old values are written back into memory before the trap

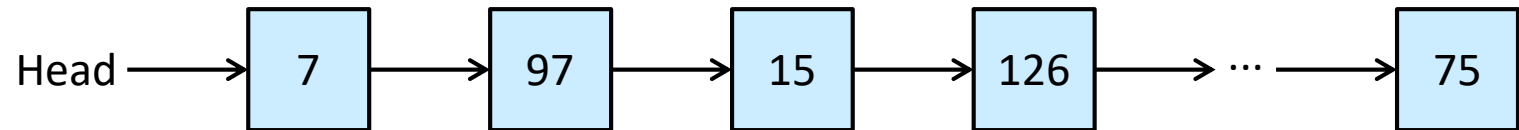


Outline

- ❑ Background
- ❑ **Demand Paging**
 - Basic Concepts, Free-Frame List, Performance of Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Free-Frame List

- ❑ When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
 - Most operating systems maintain a free-frame list
 - When a system starts up, all available memory is placed on the free-frame list



- ❑ Operating systems typically allocate free frames using zero-fill-on-demand
 - Erase their previous contents
 - Consider the potential security implications of not clearing out the contents

Outline

- ❑ Background

- ❑ **Demand Paging**

 - Basic Concepts, Free-Frame List, Performance of Demand Paging

- ❑ Copy-on-Write

- ❑ Page Replacement

- ❑ Allocation of Frames

- ❑ Thrashing

- ❑ Memory Compression

- ❑ Allocating Kernel Memory

- ❑ Other Considerations

- ❑ Operating-System Examples

Sequence of a Page Fault (1/2)

1. Trap to the operating system
2. Save the 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 in secondary storage
5. Issue a read from the storage to a free frame
 - Wait in a queue until the read request is serviced
 - Wait for the device seek and/or latency time
 - Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU core to some other process
7. Receive an interrupt from the storage I/O subsystem (I/O completed)

Sequence of a Page Fault (2/2)

8. Save the registers and process state for the other process
 9. Determine that the interrupt was from the secondary storage device
 10. Correct the page table and other tables to show that the desired page is now in memory
 11. Wait for the CPU core to be allocated to this process again
- ☐ Not all of these steps are necessary in every case

Performance of Demand Paging (1/2)

❑ Three major task components of the page-fault service time

- Service the page-fault interrupt
 - Can be reduced to several hundred instructions or 1--100 microseconds
- Read in the page
 - Probably close to 8 milliseconds with a typical hard disk
 - If a queue of processes is waiting for the device, we have to add queuing time
- Restart the process
 - Can be reduced to several hundred instructions or 1--100 microseconds

Performance of Demand Paging (2/2)

❑ Effective access time (EAT)

- Memory-access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- Probability of a page fault = p
- $EAT = (1 - p) \cdot 200 + p \cdot 8,000,000 = 200 + p \cdot 7,999,800$ (nanoseconds)

❑ If one access out of 1,000 causes a page fault

- $EAT = 8.2$ microseconds
 - Slow down by a factor of 40 because of demand paging

❑ If we want performance degradation to be less than 10%

- $p < 0.0000025$

Swap Space Handling (1/2)

❑ I/O to swap space is generally faster than that to the file system

- Swap space is allocated in much larger blocks
- File lookups and indirect allocation methods are not used (Chapter 11)

❑ For better paging throughput

- First option
 - Copy an entire file image into the swap space at process startup
 - Perform demand paging from the swap space
- Second option
 - Demand-page from the file system initially
 - Write the pages to swap space as they are replaced

Swap Space Handling (2/2)

- ❑ Some systems attempt to limit the amount of swap space used through demand paging of binary executable files
 - When page replacement is called for
 - These frames can simply be overwritten (because they are never modified)
 - The pages can be read in from the file system again if needed
 - However, swap space must still be used for pages not associated with a file (known as **anonymous memory**)
 - These pages include the stack and heap for a process
- ❑ Mobile operating systems typically do not support swapping
 - Demand-page from the file system
 - Reclaim read-only pages (such as code) from applications if memory becomes constrained

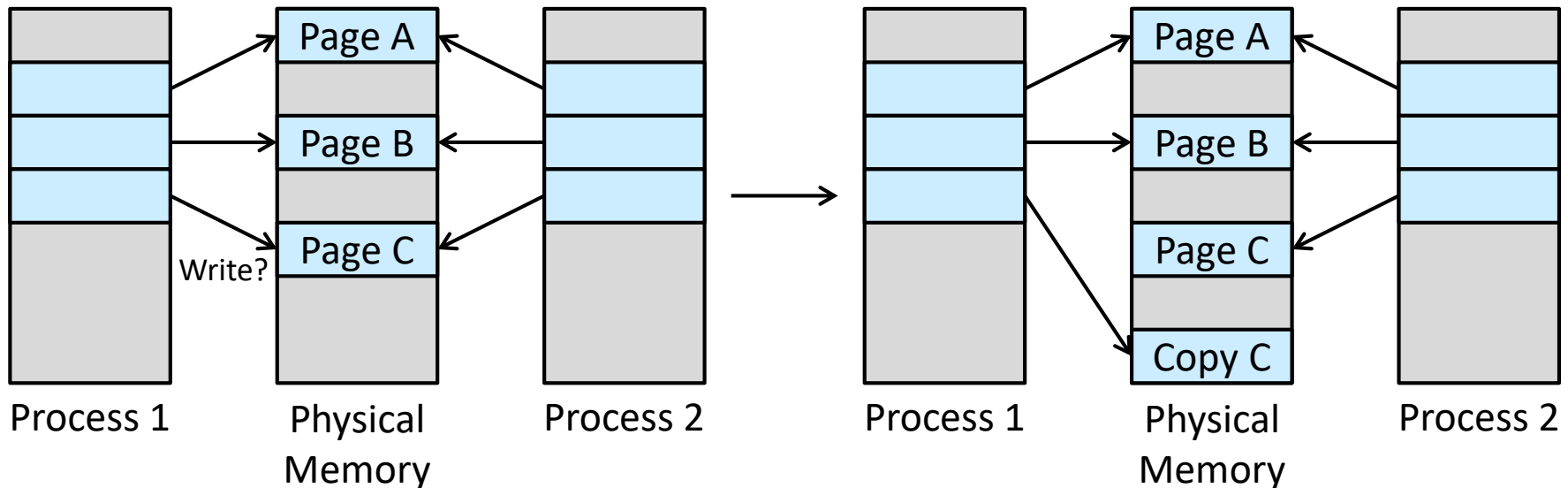
Outline

- ❑ Background
- ❑ Demand Paging
- ❑ **Copy-on-Write**
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Copy-on-Write

❑ Allow the parent and child processes initially to share the same pages

- These shared pages are marked as copy-on-write pages
 - Only pages that can be modified need be marked as copy-on-write
- If either process writes to a shared page, a copy of the shared page is created
 - When the copy-on-write technique is used, only the pages that are modified by either process are copied



Virtual Memory Fork

❑ **vfork ()**

- The parent process is suspended
- The child process uses the address space of the parent

❑ **vfork ()** does not use copy-on-write

- If the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes
- It is intended to be used when the child process calls **exec ()** immediately after creation
 - It must be used with caution
 - It is an extremely efficient method to process creation

❑ Similar concepts with thread duplication of **fork ()** ?

Outline

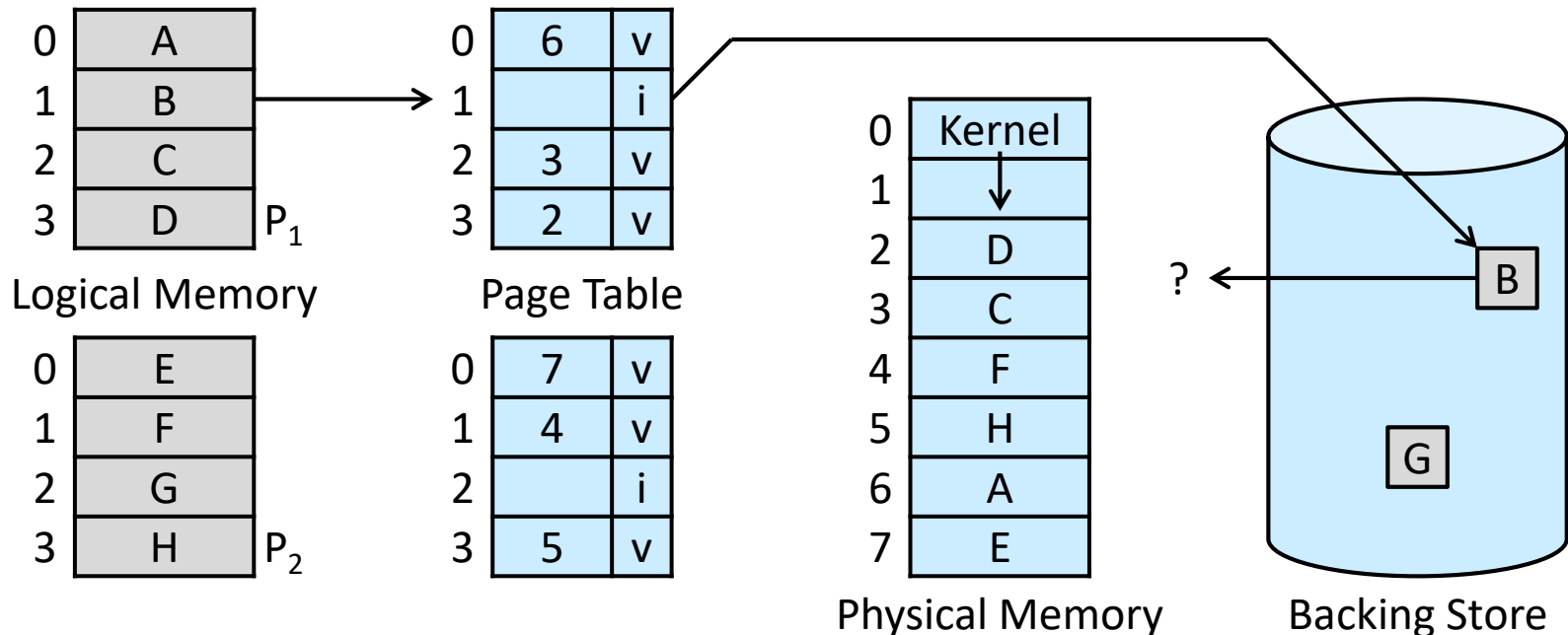
- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Page Replacement

❑ If we increase our degree of multiprogramming, we are **over-allocating** memory

➤ What if there is no free frame on the free-frame list?

- Terminate the process (does not make sense)
- Swap out a process (high overhead)
- Combine swapping pages with **page replacement**

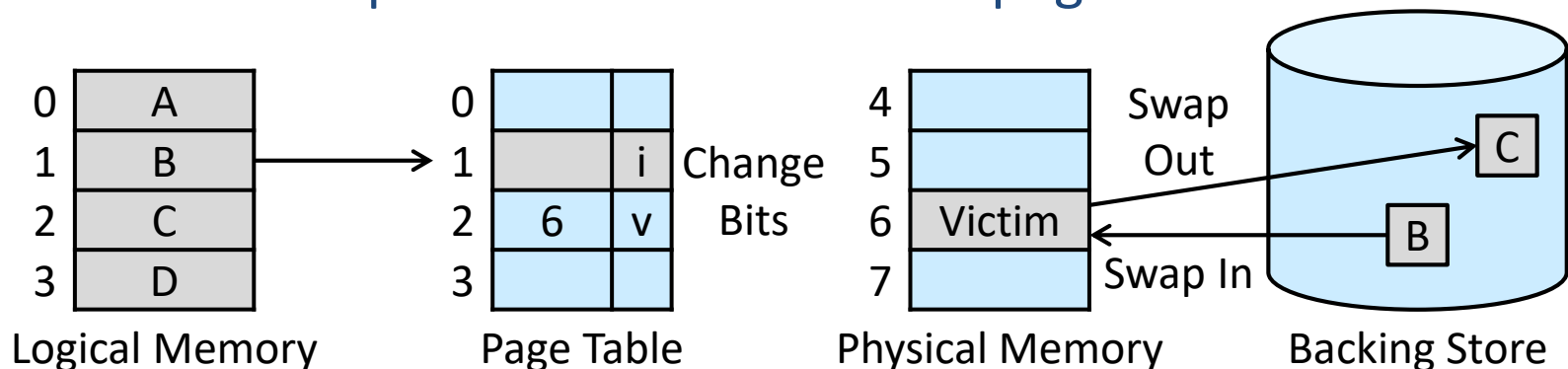


Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - **Basic Page Replacement**
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Basic Page Replacement

1. Find the location of the desired page on secondary storage
2. Find a free frame
 - If there is a free frame, use it
 - If there is no free frame
 - Use a page-replacement algorithm to select a victim frame
 - Write the victim frame to secondary storage (if necessary) and change the page and frame tables accordingly
3. Read the desired page into the newly freed frame and change the page and frame tables
4. Continue the process from where the page fault occurred



Page Replacement Overhead

- ❑ If no frame is free, two page transfers (one for the page-out and one for the page-in) are required
 - Double the page-fault service time
 - Increase the effective access time
- ❑ Reduce this overhead by using a modify bit (or dirty bit)
 - Each page or frame has a modify bit associated with it in the hardware
 - Indicate that the page has been modified
 - When we select a page for replacement
 - If the modify bit is set, we must write the page to storage
 - If the modify bit is not set, we need not write the memory page to storage
 - It reduces I/O time by one-half if the page has not been modified

Two Major Problems: Design

❑ Frame-allocation algorithm

- Decide how many frames to allocate to each process

❑ Page-replacement algorithm

- Select the frames that are to be replaced

❑ Design appropriate algorithms

- It is important task because secondary storage I/O is so expensive
- In general, we want the one with the lowest page-fault rate

Two Major Problems: Analysis

- ❑ Evaluate algorithms by running it on a reference string of memory references and computing the number of page faults
 - Example with 100 bytes per page
 - 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103 is reduced to 1, 4, 1, 6, 1, 6, 1
 - Running example with 3 frames
 - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- ❑ As the number of frames increases, the number of page faults drops to some minimal level

Outline

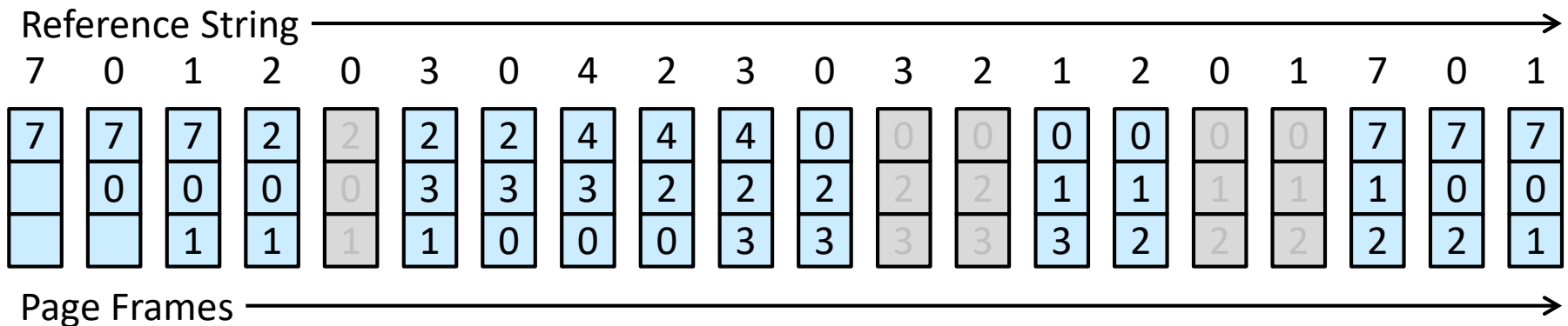
- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - **FIFO Page Replacement**
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

FIFO Page Replacement

❑ First-in, first-out (FIFO) algorithm

- Create a FIFO queue to hold all pages in memory
- Replace a page at the head of the queue
- Insert a page at the tail of the queue

❑ 15 page faults in the running example



Belady's Anomaly

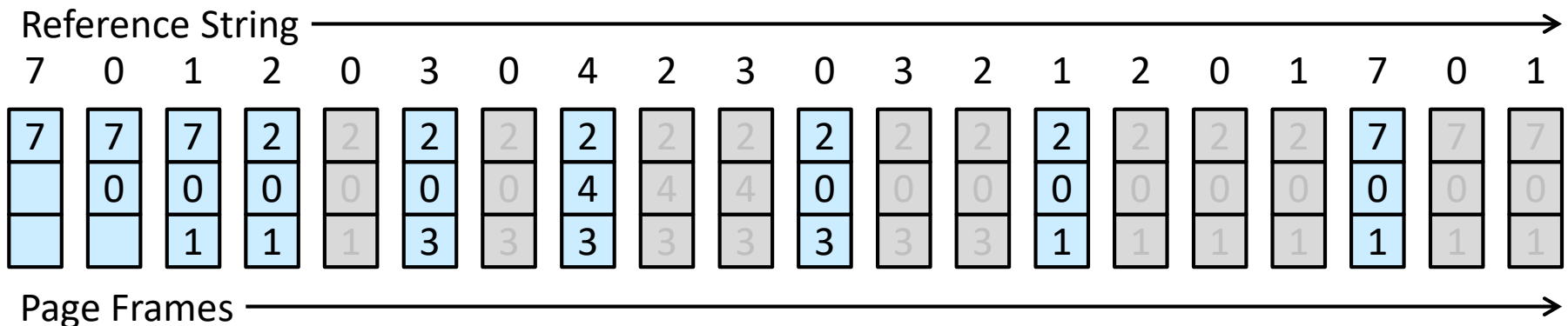
- ❑ For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases
- ❑ Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
 - If # of frames = 1, # of page faults = 12
 - If # of frames = 2, # of page faults = 12
 - If # of frames = 3, # of page faults = 9
 - 1 2 3 4 1 2 5 ☺ ☺ 3 4 ☺
 - If # of frames = 4, # of page faults = 10
 - 1 2 3 4 ☺ ☺ 5 1 2 3 4 5
 - If # of frames ≥ 5 , # of page faults = 5
- ❑ Giving more memory to a process does not necessarily improve its performance

Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - **Optimal Page Replacement**
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Optimal Page Replacement

- ❑ Replace the page that will not be used for the longest period of time
 - Guarantee the lowest number of page faults for a fixed reference string
- ❑ 9 page faults in the running example



- ❑ It requires future knowledge of the reference string
 - The optimal algorithm is used mainly for comparison studies

Outline

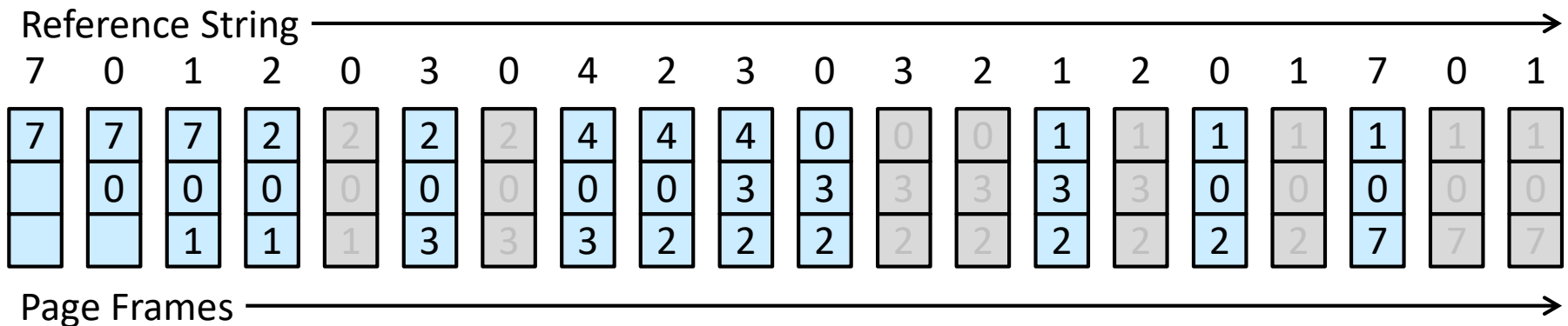
- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - **LRU Page Replacement**
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

LRU Page Replacement

❑ Least recently used (LRU) algorithm

- Replace the page that has not been used for the longest period of time
 - Change the optimal page replacement from "looking forward" to "looking backward"

❑ 12 page faults in the running example



❑ Often used and considered to be good

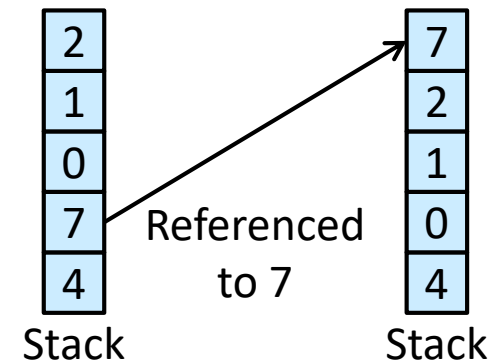
Implementing LRU Page Replacement

❑ Approach 1: counters

- Associate a time-of-use field with each page-table entry
- Add a logical clock (or counter) to the CPU
- If a page is referenced, copy the clock register to its time-of-use field
- Replace the page with the smallest time value
 - Require a search of the page table

❑ Approach 2: stack

- Keep a stack of page numbers
 - Use a doubly linked list with a head pointer and a tail pointer
- If a page is referenced, remove it from the stack and put it on the top
 - The least recently used page is always at the bottom
- Replace the page pointed by the tail pointer
 - No search but a little more expensive update



Stack Algorithms

- ❑ The set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $(n + 1)$ frames
 - Optimal page replacement
 - LRU page replacement
- ❑ They do not suffer from Belady's anomaly

Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - **LRU-Approximation Page Replacement**
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

LRU-Approximation Page Replacement

- ❑ Not many computer systems provide sufficient hardware support for true LRU page replacement
 - The updating of the clock fields or stack must be done for every memory reference
- ❑ Many systems provide help in the form of a reference bit
 - The reference bit (in the page table) of a page is set to 1 by the hardware when the page is referenced
 - We can determine which pages have (or have not) been used after their reference bits are set to 0
 - However, we do not know the order of use
- ❑ Examples
 - Additional-reference-bits algorithm
 - Second-chance algorithm
 - Enhanced second-chance algorithm

Additional-Reference-Bits Algorithm

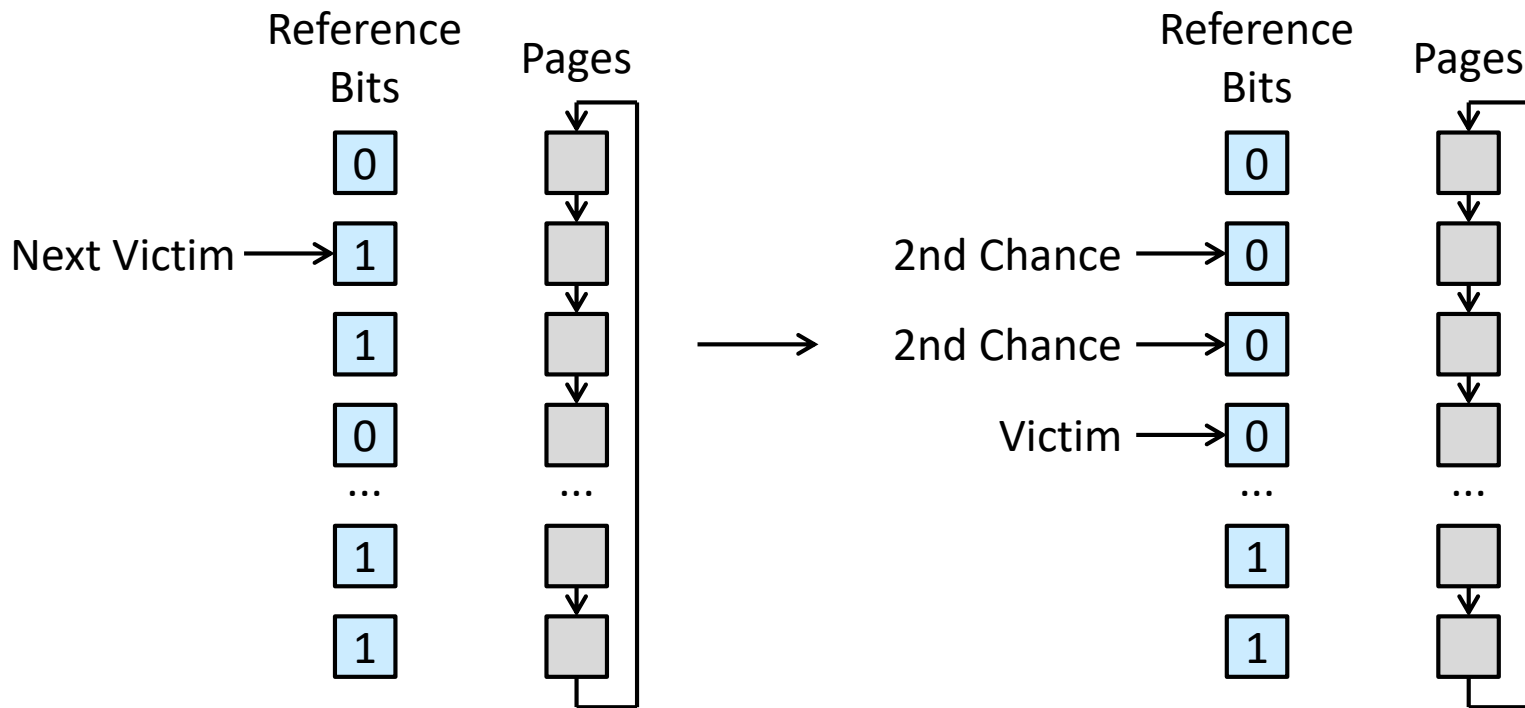
- ❑ Gain additional ordering information by recording the reference bits at shift registers at regular intervals
 - At regular intervals (e.g., every 100 milliseconds), a timer interrupt transfers control to the operating system which
 - Shift the reference bit for each page into the high-order bit
 - Shift the other bits right
 - Example: keep an 8-bit byte for each page in a table in memory
 - 00000000 vs. 11111111 vs. 11000100 vs. 01110111
 - The number of bits can be varied
 - In the extreme case, it can be zero, leaving only the reference bit itself (second-chance page-replacement algorithm)

Second-Chance Algorithm (1/2)

- ❑ A FIFO replacement algorithm with reference bits
- ❑ When a page has been selected, inspect its reference bit
 - If it is 0, replace the page
 - If it is 1, give the page a second chance and select the next FIFO page
- ❑ When a page gets a second chance
 - Its reference bit is cleared to 0
 - Its arrival time is reset to the current time
 - Thus, it will not be replaced until all other pages have been replaced or given second chances
- ❑ Note that the reference bit of a page is set to 1 when the page is referenced
 - If a page is used often enough to keep its reference bit set (to 1), it will never be replaced

Second-Chance Algorithm (2/2)

- ❑ One way to implement the second-chance algorithm is as a circular queue
 - Sometimes referred to as the clock algorithm
- ❑ In the worst case, if all bits are set (to 1), the second chance replacement degenerates to FIFO replacement



Enhanced Second-Chance Algorithm

- ❑ Consider the reference bit and the modify bit (described earlier) as an ordered pair
 - (0, 0): neither recently used nor modified
 - Best page to replace
 - (0, 1): not recently used but modified
 - Not quite as good, because 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 need to be written out before replacement
- ❑ Use the same scheme as in the clock algorithm and examine the class to which that page belongs

Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - **Counting-Based Page Replacement**
 - Page-Buffering Algorithms
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Counting-Based Page Replacement

- ❑ Keep a counter of the number of references that have been made to each page
 - The implementation of these algorithms is expensive
 - They do not approximate the optimal replacement well
- ❑ **Least frequently used (LFU) page-replacement algorithm**
 - Require that the page with the smallest count be replaced
 - An actively used page should have a large reference count
 - What if a page is used heavily only during the initial phase of a process? Shift the counts right by 1 bit at regular intervals
- ❑ **Most frequently used (MFU) page-replacement algorithm**
 - Require that the page with the largest count be replaced
 - The page with the smallest count was probably just brought in and has yet to be used

Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - **Page-Buffering Algorithms**
 - Applications and Page Replacement
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Page-Buffering Algorithms

❑ 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
 - Restart the process as soon as possible

❑ Maintain a list of modified pages

- When the paging device is idle, a modified page is selected and written to secondary storage, and its modify bit is then reset
 - Increase the probability that a page will be clean

❑ Keep a pool of free frames but remember which page was in each frame

- When a page fault occurs, check whether the desired page is in the free-frame pool
 - If not, select a free frame and read into it

Outline

- ❑ Background, Demand Paging, Copy-on-Write
- ❑ **Page Replacement**
 - Basic Page Replacement
 - FIFO Page Replacement
 - Optimal Page Replacement
 - LRU Page Replacement
 - LRU-Approximation Page Replacement
 - Counting-Based Page Replacement
 - Page-Buffering Algorithms
 - **Applications and Page Replacement**
- ❑ Allocation of Frames, Thrashing, Memory Compression
- ❑ Allocating Kernel Memory, Other Considerations, Operating-System Examples

Applications and Page Replacement

- ❑ Applications accessing data through the operating system's virtual memory may perform worse
 - Some applications understand their memory use and storage use better than an operating system
 - Examples: database, data warehouses
 - Twice the memory is being used for a set of I/O
 - Both the operating system and the application are buffering I/O
- ❑ Some operating systems provide a **raw disk**
 - A large sequential array of logical blocks, without any file-system data structure
 - Bypass all the file-system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ **Allocation of Frames**
 - Minimum Number of Frames, Allocation Algorithms, Global versus Local Allocation, Non-Uniform Memory Access
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Recap: Two Major Problems

- ❑ Covered: page-replacement algorithm

- Select the frames that are to be replaced

- ❑ Frame-allocation algorithm

- Decide how many frames to allocate to each process

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ **Allocation of Frames**
 - **Minimum Number of Frames**, Allocation Algorithms, Global versus Local Allocation, Non-Uniform Memory Access
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Minimum Number of Frames

- ❑ Must allocate at least a minimum number of frames
 - Performance: as the number of frames allocated to each process decreases, the page-fault rate increases
 - Instruction restart: we must hold all the different pages that any single instruction can reference
- ❑ The minimum number is defined by the computer architecture
 - Example: six frames for an instruction
 - The move instruction for a given architecture itself straddles two frames
 - Each of its two operands is an indirect reference (e.g., via a page table)
 - Example: what if the move instruction allows data to move only from register to register and between registers and memory?
- ❑ The maximum number is defined by the amount of available physical memory

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ **Allocation of Frames**
 - Minimum Number of Frames, **Allocation Algorithms**, Global versus Local Allocation, Non-Uniform Memory Access
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Allocation Algorithms

❑ Equal allocation (ignoring frames needed by OS now)

- Example: split 93 frames among 5 processes
 - Give each process 18 frames
 - Leave 3 frames as a free-frame buffer pool

❑ Proportional allocation

- Allocate available memory to each process according to its size
- Adjust to meet the minimum number of frames
- Example: split 62 frames among 2 processes with 10 and 127 pages
 - $10 / (10 + 127) * 62 \approx 4$ (frames)
 - $127 / (10 + 127) * 62 \approx 57$ (frames)
- Variation
 - The ratio of frames depends not on the relative sizes but rather on the priorities (or on a combination of size and priority)

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ **Allocation of Frames**
 - Minimum Number of Frames, Allocation Algorithms, **Global versus Local Allocation**, Non-Uniform Memory Access
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Global versus Local Allocation

❑ Global replacement

- A process selects a replacement frame from the set of all frames, and thus it can take a frame from another
 - Greater system throughput

❑ Local replacement

- A process selects from only its own set of allocated frames
 - More consistent per-process performance

❑ Global replacement is the more commonly used method

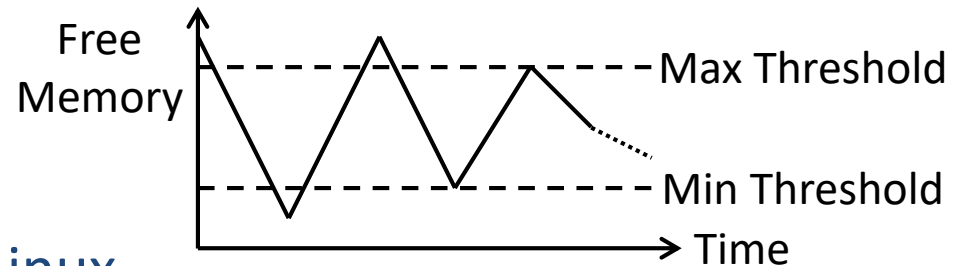
Reclaiming Pages

- ❑ A global page-replacement policy: keep the amount of free memory above a minimum threshold
 - When it drops below the threshold, a kernel routine is triggered that begins reclaiming pages from all processes in the system
 - Such kernel routines are often known as reapers
 - When the amount of free memory reaches a maximum threshold, the reaper routine is suspended

❑ It may adopt any page-replacement algorithm

❑ A more extreme example in Linux

- If the amount of free memory is very low, a routine, out-of-memory (OOM) killer, selects a process to terminate by OOM scores



Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ **Allocation of Frames**
 - Minimum Number of Frames, Allocation Algorithms, Global versus Local Allocation, **Non-Uniform Memory Access**
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Non-Uniform Memory Access (NUMA)

❑ Non-uniform memory access systems with multiple CPUs

➤ Not all main memory is created equal (or accessed equally)

❑ The goal is to have memory frames allocated "as close as possible" to the CPU on which the process is running

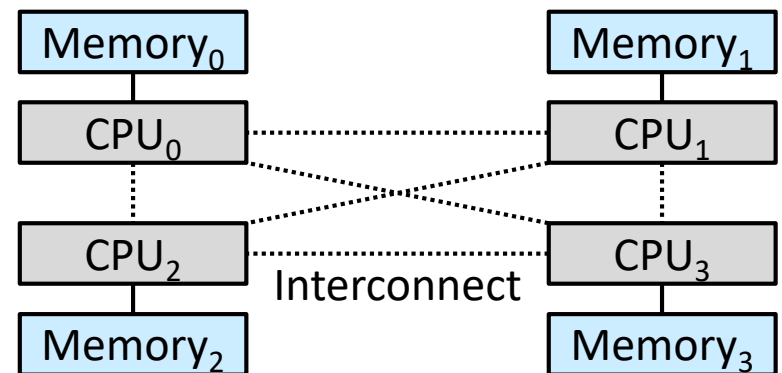
❑ More complicated with threads

➤ Linux

- Migrate threads to the same domain
- Have a separate free-frame list for each NUMA node

➤ Solaris

- Each lgroup (locality groups) gathers together CPUs and memory
- Each CPU in that group can access any memory in the group within a defined latency interval



Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ **Thrashing**
 - Cause of Thrashing, Working-Set Model, Page-Fault Frequency, Current Practice
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Thrashing

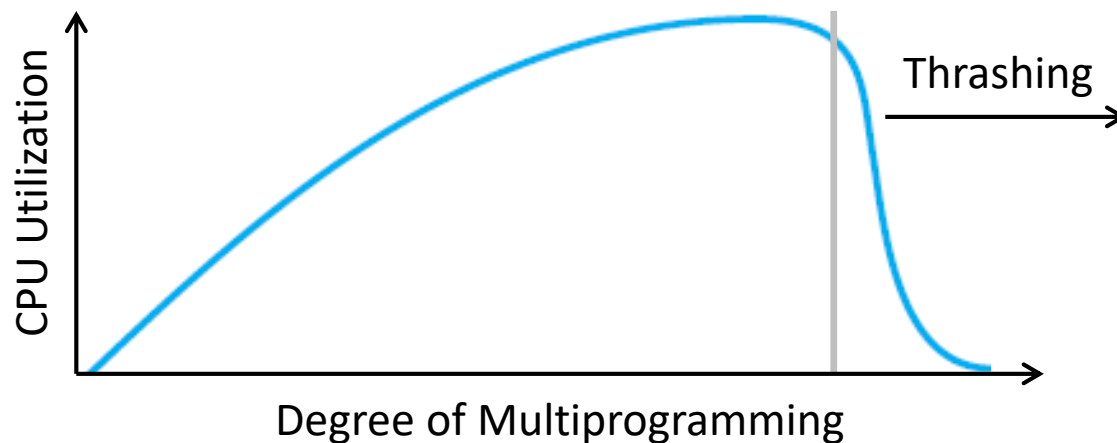
- ❑ What if a process does not have the minimum number of frames it needs to support pages in the working set
 - Quickly page-fault
 - Replace a page that will be needed again right away
 - Quickly page-faults again, and again, and again
- ❑ A process is **thrashing** if it is spending more time paging than executing
 - Thrashing results in severe performance problems

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ **Thrashing**
 - **Cause of Thrashing**, Working-Set Model, Page-Fault Frequency, Current Practice
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Cause of Thrashing

- ❑ A global page-replacement algorithm is used
 - A process needs more frames
 - It faults and takes frames from other processes
 - These processes also fault and take frames from other processes
 - As they queue up for the paging device, the ready queue empties
 - CPU utilization decreases
 - The CPU scheduler increases the degree of multiprogramming ...



Locality Model

- ❑ We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm)
 - The problem is not entirely solved (even if there is no "frame-stealing")
 - Thrashing processes are in the queue for the paging device most of the time
 - The effective access time will increase even for a process that is not thrashing
- ❑ **Locality model**
 - As a process executes, it moves from locality to locality
 - A locality is a set of pages that are actively used together
 - Example: the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}
 - Example: the set of pages {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}
 - Different localities may overlap
 - If there are not enough frames to accommodate the size of the current locality, the process will thrash

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ **Thrashing**
 - Cause of Thrashing, **Working-Set Model**, Page-Fault Frequency, Current Practice
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Working-Set Model (1/2)

❑ **Working set** (WS): the set of pages in the most recent Δ page references

➤ Δ : **working-set window**

- If Δ is too small, it will not encompass the entire locality
- If Δ is too large, it may overlap several localities
- If Δ is infinite, it is the set of pages touched during the process execution

➤ Example

- Page reference: 2 6 1 5 7 7 7 7 5 1 (t_1) 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 (t_2)
- If $\Delta = 10$, then $WS(t_1) = \{1, 2, 5, 6, 7\}$ and $WS(t_2) = \{3, 4\}$

❑ **D: total demand for frames**

- $D = \sum_i$ (working-set size of each process i)
- If $D >$ the total number of available frames, thrashing will occur

Working-Set Model (2/2)

- ❑ This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
- ❑ Difficulty: keeping track of the working set
 - The working-set window is a moving window
 - A page is in the working set if it is referenced anywhere in the window
- ❑ Approximation: a timer interrupt and a reference bit
 - Example: $\Delta = 10,000$ references
 - Interrupt every 5,000 references
 - Keep 2 additional bits for each page
 - When the timer interrupts, copy and clear the reference bit for each page
 - Examine the current reference bit and two additional bits
 - Trade-off: interrupt every 1,000 references and keep 10 additional bits

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ **Thrashing**
 - Cause of Thrashing, Working-Set Model, **Page-Fault Frequency**, Current Practice
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Page-Fault Frequency

- ❑ Knowledge of the working set is useful for prepaging, but it seems a clumsy way to control thrashing
- ❑ The page-fault frequency (PFF) takes a more direct approach
 - Control the page-fault rate
 - Establish upper and lower bounds on the desired page-fault rate
 - Allocate a frame to a process if the page-fault rate exceeds the upper bound
 - Remove a frame from a process if the page-fault rate falls below the lower bound

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ **Thrashing**
 - Cause of Thrashing, Working-Set Model, Page-Fault Frequency, **Current Practice**
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Current Practice

- ❑ Thrashing and the resulting swapping have a high impact on performance
- ❑ The best practice is to include enough physical memory to avoid thrashing and swapping
 - Keeping all working sets in memory concurrently, except under extreme conditions, provides the best user experience

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ **Memory Compression**
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples

Memory Compression (1/2)

❑ Compress several frames into a single frame

- Rather than paging out modified frames to swap space

❑ Example

- Before compression

- Free-frame list: Head → 7 → 2 → 9 → 21 → 27 → 16
- Modified frame list: Head → 15 → 3 → 35 → 26

- Frames 15, 3, and 35 are compressed and stored in frame 7

- After compression

- Free-frame list: Head → 2 → 9 → 21 → 27 → 16 → 15 → 3 → 35
- Modified frame list: Head → 26
- Compressed frame list: Head → 7

- If one compressed frame is later referenced, a page fault occurs

- The compressed frame is decompressed, restoring the pages 15, 3, and 35

Memory Compression (2/2)

- ❑ Memory compression is an integral part of the memory-management strategy for most mobile operating systems
 - Android and iOS
 - In addition, both Windows 10 and macOS also support it
- ❑ Trade-off between
 - The speed of the compression algorithm
 - The amount of reduction that can be achieved (compression ratio)

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ **Allocating Kernel Memory**
 - Buddy System, Slab Allocation
- ❑ Other Considerations
- ❑ Operating-System Examples

Allocating Kernel Memory

- ❑ Kernel memory is often allocated from a free-memory pool different from the list used for ordinary user-mode processes
- ❑ Reasons
 - The kernel requests memory for data structures of varying sizes, some of which are less than a page in size
 - Internal fragmentation
 - Certain hardware devices interact directly with physical memory and consequently require memory residing in physically contiguous pages
 - No benefit of a virtual memory interface

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ **Allocating Kernel Memory**
 - **Buddy System**, Slab Allocation
- ❑ Other Considerations
- ❑ Operating-System Examples

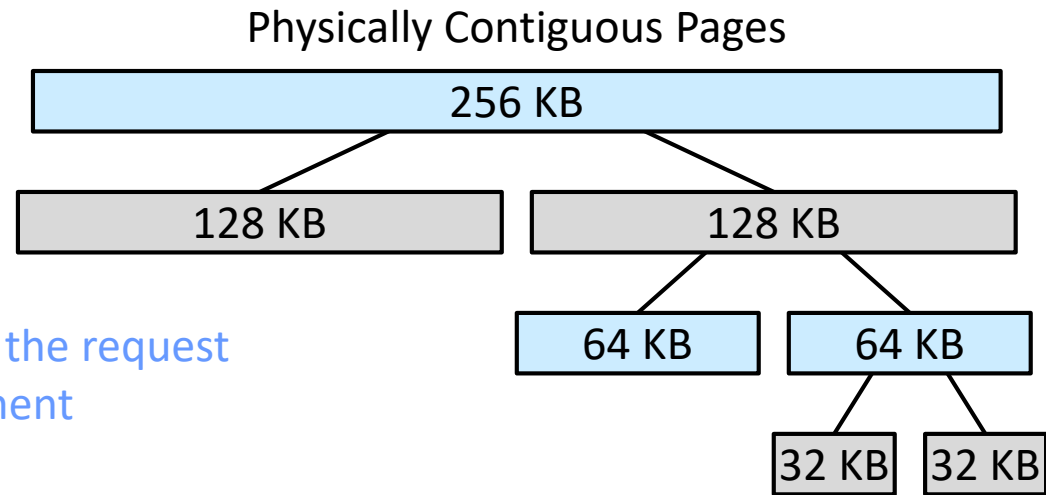
Buddy System

❑ Allocate memory from a fixed-size segment consisting of physically contiguous pages

➤ Memory is allocated from this segment using a **power-of-2 allocator**

➤ Example

- A 256-KB memory segment
- A 21-KB request from kernel
- Two **buddies** divided from the segment
- Buddies further divided until the request is satisfied with a 32-KB segment



➤ Advantage

- Adjacent **buddies** can be combined to form larger segments using **coalescing**

➤ Disadvantage

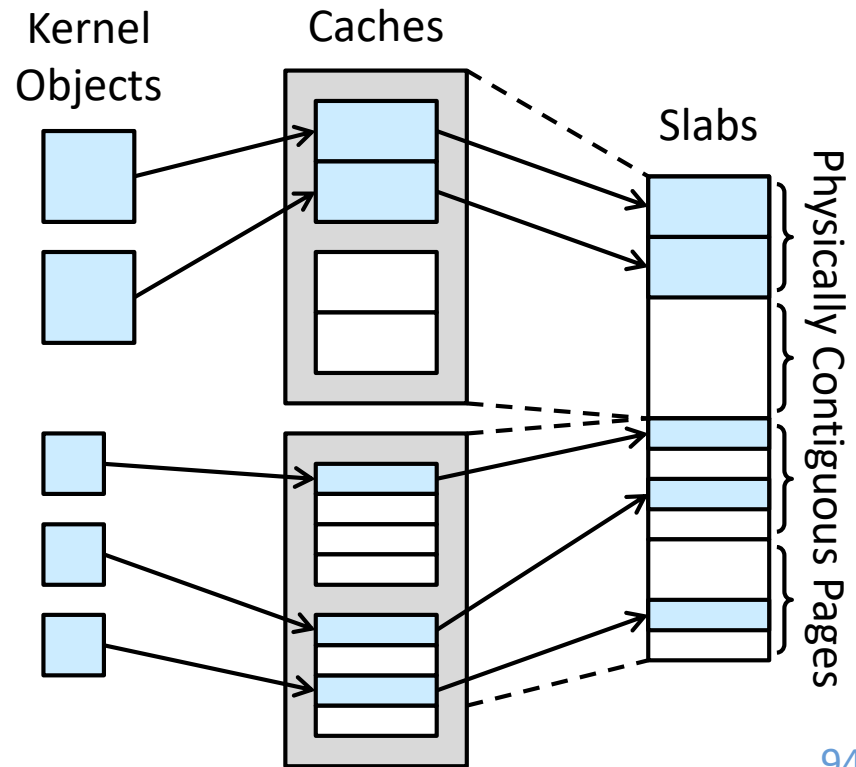
- Internal fragmentation

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ **Allocating Kernel Memory**
 - Buddy System, **Slab Allocation**
- ❑ Other Considerations
- ❑ Operating-System Examples

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
 - Examples: process descriptors (process control blocks), file objects, semaphores (Chapter 6)
- ❑ Each cache is populated with **objects**
 - Instantiations of the kernel data structure that the cache represents
- ❑ The allocator
 - Assign any **free** object from the cache to satisfy a request
 - Mark the object as **used**



Slab Allocation: Benefits

❑ No memory is wasted due to fragmentation

- The slab allocator returns the exact amount of memory required to represent an object
 - Each unique kernel data structure has an associated cache
 - Each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented

❑ Memory requests can be satisfied quickly

- Objects are quickly allocated from the cache
 - They are created in advance
- Objects are immediately available for subsequent requests
 - When the kernel has finished with them, they are marked as free and returned to the cache

Slab Allocator in Linux (1/2)

❑ Example

- A process descriptor is of the type `struct task_struct`
 - Approximately 1.7 KB of memory
- The kernel requests memory for a `struct task_struct` object from its cache
- The cache fulfills the request using a `struct task_struct` object that has been allocated in a slab and marked as free

❑ Three possible states of a slab

- Full: all objects are used
- Empty: all objects are free
 - Second choice
- Partial: the slab consists of both used and free objects
 - First choice
- If no empty slab exists, a new slab is allocated and assigned to a cache

Slab Allocator in Linux (2/2)

- ❑ The slab allocator first appeared in the Solaris 2.4 kernel
- ❑ The Linux kernel adopted the SLAB allocator from Version 2.2
 - Originally used the buddy system
- ❑ Recent Linux versions include other kernel memory allocators
 - The SLOB (simple list of blocks) allocator is designed for systems with limited memory, such as embedded systems
 - Three lists of objects: small (<256 bytes), medium (<1,024 bytes), and large (<the size of a page)
 - The SLUB allocator (from Version 2.6.24) reduces much of the overhead required by the SLAB allocator
 - Do not store certain metadata with each slab
 - Do not include the per-CPU queues

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ **Other Considerations**
 - Prepaging, Page Size, TLB Reach, Inverted Page Tables, Program Structure, I/O Interlock and Page Locking
- ❑ Operating-System Examples

Prepaging

- ❑ Bring some (or all) of the pages that will be needed into memory at one time
 - A large number of page faults occur when a process is started
 - Prepaging is an attempt to prevent this high level of initial paging
- ❑ The question: whether the cost of using prepaging is less than the cost of servicing the corresponding page faults?
 - Prepaging a file may be more predictable
 - Files are often accessed sequentially
 - The Linux `readahead()` system call prefetches the contents of a file into memory

Page Size

❑ Page sizes are invariably powers of 2

- Generally ranging from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes

❑ Factors of page size selection

- Page table size → a larger page is better
- Internal fragmentation → a smaller page is better
- I/O time → a larger page is better
- Resolution and locality → a smaller page is better
- Number of page faults → a larger page is better
- Others

❑ The historical trend is toward larger page sizes

- Even for mobile systems

TLB Reach

❑ Translation look-aside buffer (TLB) reach

- The amount of memory accessible from the TLB
- The number of entries multiplied by the page size

❑ Ideally, the working set for a process is stored in the TLB

❑ To increase the TLB reach

- Increase the number of entries (in the TLB)
 - Still insufficient for storing the working set of a memory-intensive application
- Increase the page size
 - Increase internal fragmentation for an application that does not require such a large page size
- Provide multiple page sizes
 - Allow an application to use a large page size without increasing internal fragmentation
 - Require the operating system, rather than hardware, to manage the TLB

Inverted Page Tables

- ❑ Do not contain complete information about the logical address space of a process
 - The information is required if a referenced page is not in memory
- ❑ An external page table (one per process) must be kept
 - It is like the traditional per-process page table and contains information on where each virtual page is located
 - It does not need to be available quickly, but it may generate another page fault for itself
 - This special case requires careful handling in the kernel and a delay in the page-lookup processing

Program Structure

❑ `int[128][128] data;`

- Assume that each row is stored in one page and the array is row major
 - In memory: `data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]`

❑ Possible $128 \times 128 = 16,384$ faults

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

❑ At most 128 faults

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

❑ Compiler and loader can have a significant effect on paging

I/O Interlock and Page Locking

- ❑ When demand paging is used, we sometimes need to allow some of the pages to be locked in memory
- ❑ Bad example
 - The page containing the memory buffer of a process is paged out (by another process) during waiting I/O completion
- ❑ Solutions
 - Execute I/O to system memory (not user memory)
 - Allow pages to be locked
 - Frequently, some or all of the operating-system kernel is locked into memory
 - User processes may also need to lock pages into memory (e.g., pinning of pages by a database process)

Outline

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory Compression
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ **Operating-System Examples**
 - Linux, Windows, Solaris

Linux

- ❑ Manage kernel memory using slab allocation
- ❑ Use demand paging
 - Allocate pages from a list of free frames
- ❑ Use a global page-replacement policy
 - Similar to the LRU-approximation clock algorithm
- ❑ Maintain two types of page lists
 - Active list
 - Contain the pages that are considered in use
 - Inactive list
 - Contain the pages that have not recently been referenced and are eligible to be reclaimed

Windows

❑ Clustering for demand paging

- Bring in not only the faulting page but also the pages preceding and following the faulting page

❑ Working-set management

- Working-set minimum: the minimum number of assigned pages
- Working-set maximum: the maximum number of assigned pages
- May ignore the values unless a process is with hard working-set limits

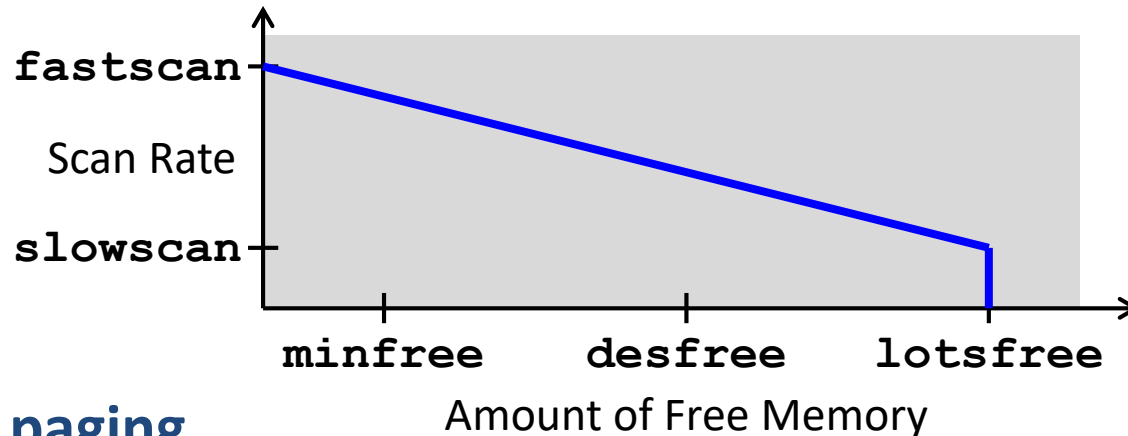
❑ Automatic working-set trimming

- When the amount of free memory falls below the threshold, use a global replacement tactic to restore it above the threshold

Solaris

❑ Pageout algorithm

- Use two hands to scan pages
 - The front hand of the clock sets the reference bits of all pages to 0
 - The back hand of the clock appends a page to the free list (if the reference bit is still 0) or writes its contents to secondary storage (otherwise)
- Control the rate at which pages are scanned



❑ Priority paging

- Skip pages belonging to libraries that are shared by several processes
- Distinguish between pages allocated to processes and regular data files

Objectives

- ❑ Define virtual memory and describe its benefits
- ❑ Illustrate how pages are loaded into memory using demand paging
- ❑ Apply the FIFO, optimal, and LRU page-replacement algorithms
- ❑ Describe the working set of a process, and explain how it is related to program locality
- ❑ Describe how Linux, Windows 10, and Solaris manage virtual memory

Q&A