Operating Systems
[10A. Virtual Memory]

Chung-Wei Lin
cwlin@csie.ntu.edu.tw
CSIE Department
National Taiwan University

From Operating System Concepts (10th Edition)

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)

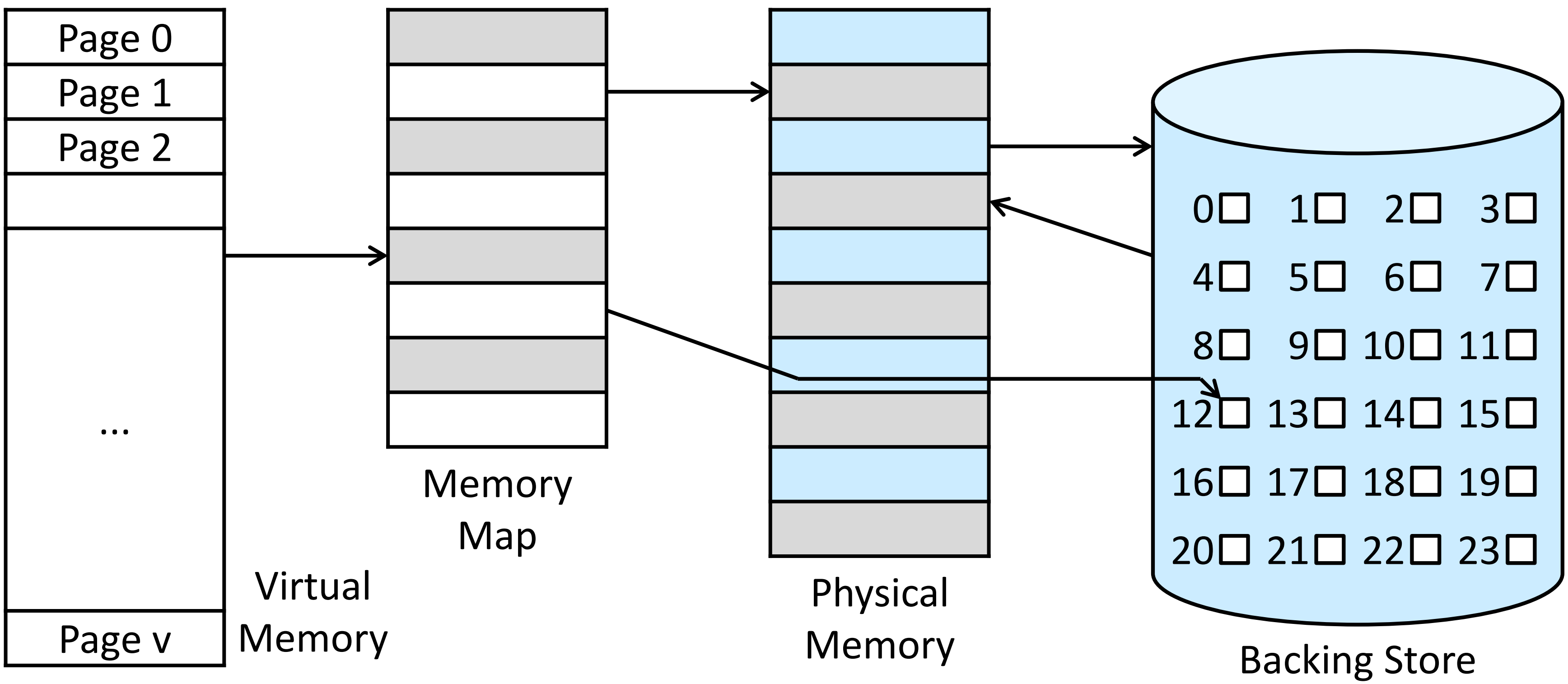
- Basic requirement
  - The instructions being executed must be in physical memory
- One approach
  - Place the entire logical address space in physical memory
    - Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer
  - Limit 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

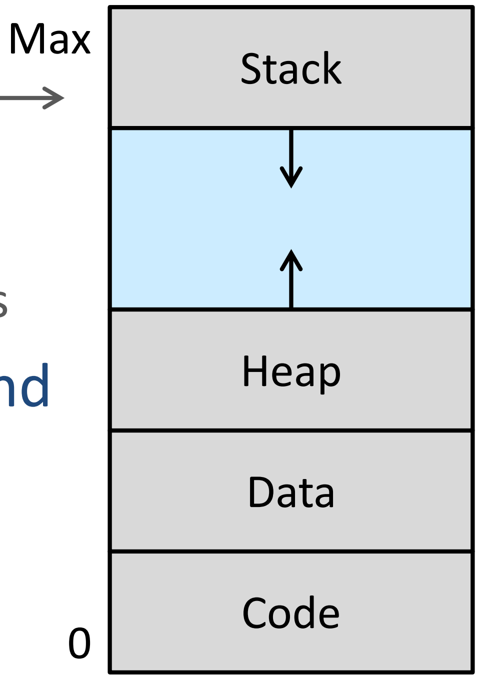
Virtual Memory

- 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

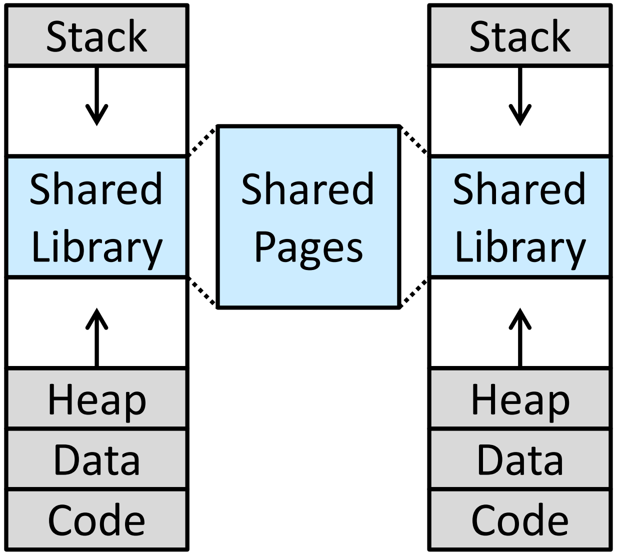
- Logical (or virtual) view of how a process is stored in memory
  - A process begins at a certain logical address and exists in contiguous memory
  - The physical page frames assigned to the process may not be contiguous
  - The MMU maps logical pages to physical page frames



- 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

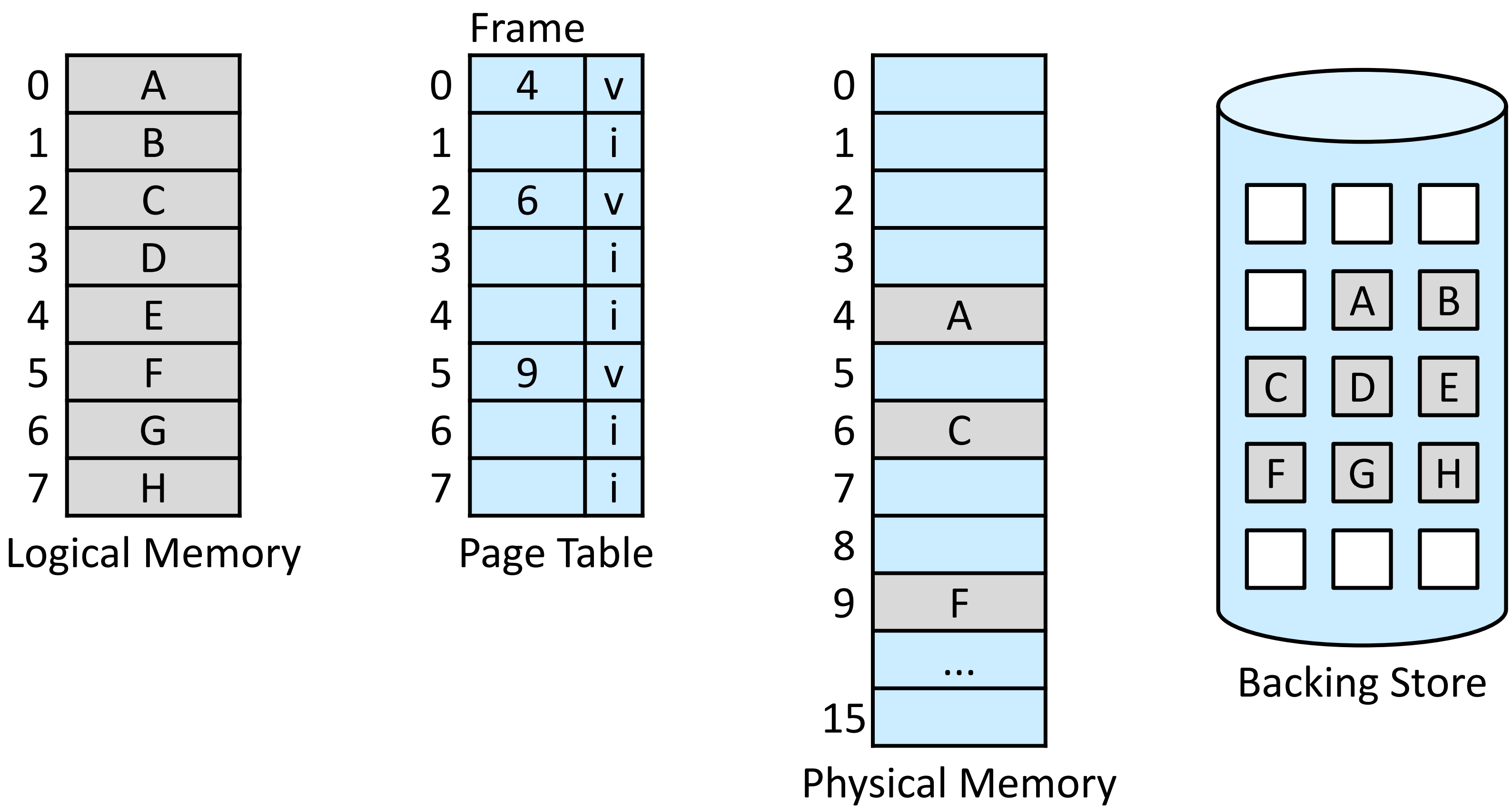
- One approach
  - Program starts with a list of available options that the user is to select
- Another approach: **demand paging**
  - Pages are loaded only when they are **demanded** during execution
  - Pages that are never accessed are never loaded into physical memory
  - It is commonly used in 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

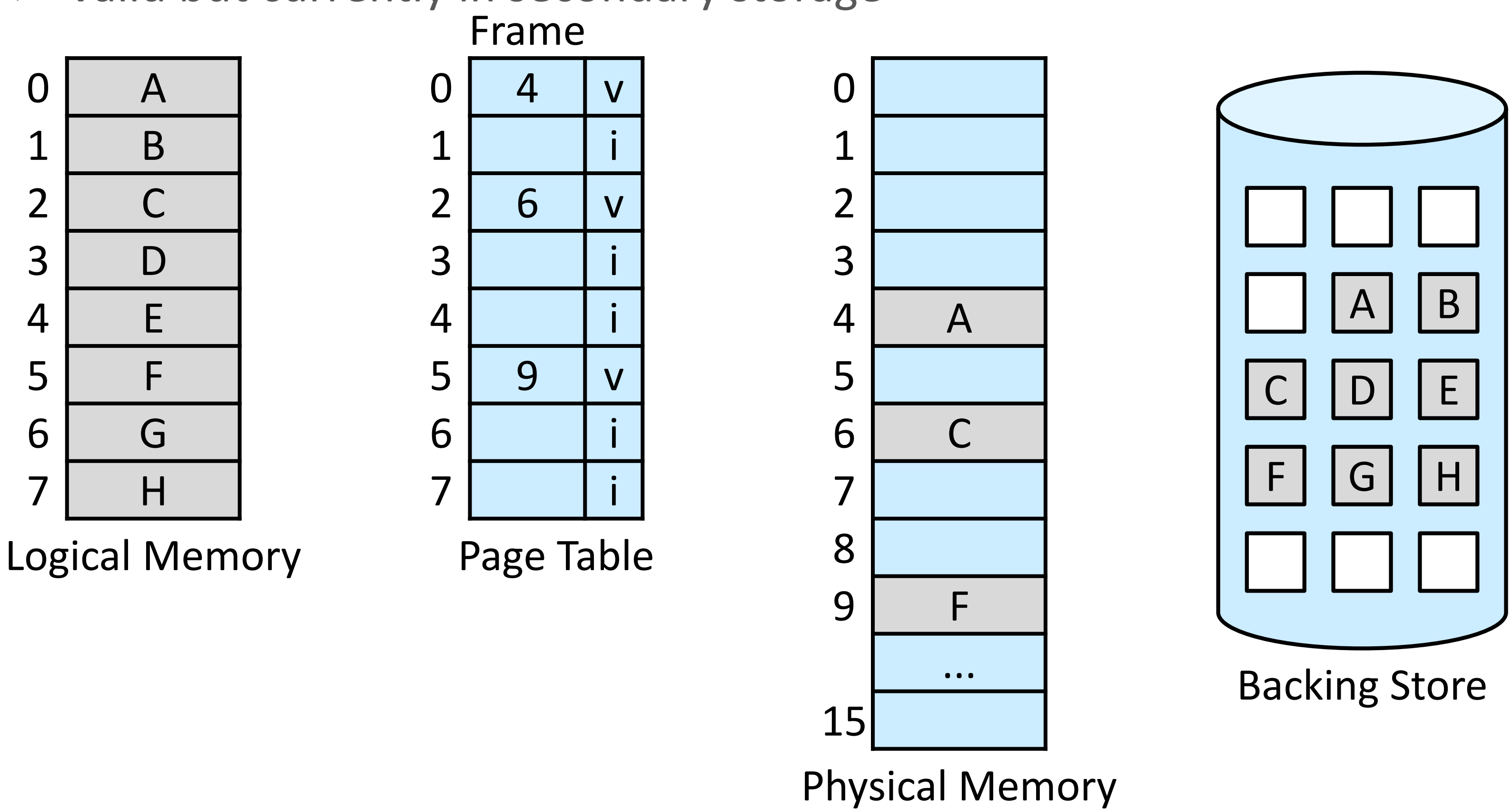
Basic Concepts (1/2)

- 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



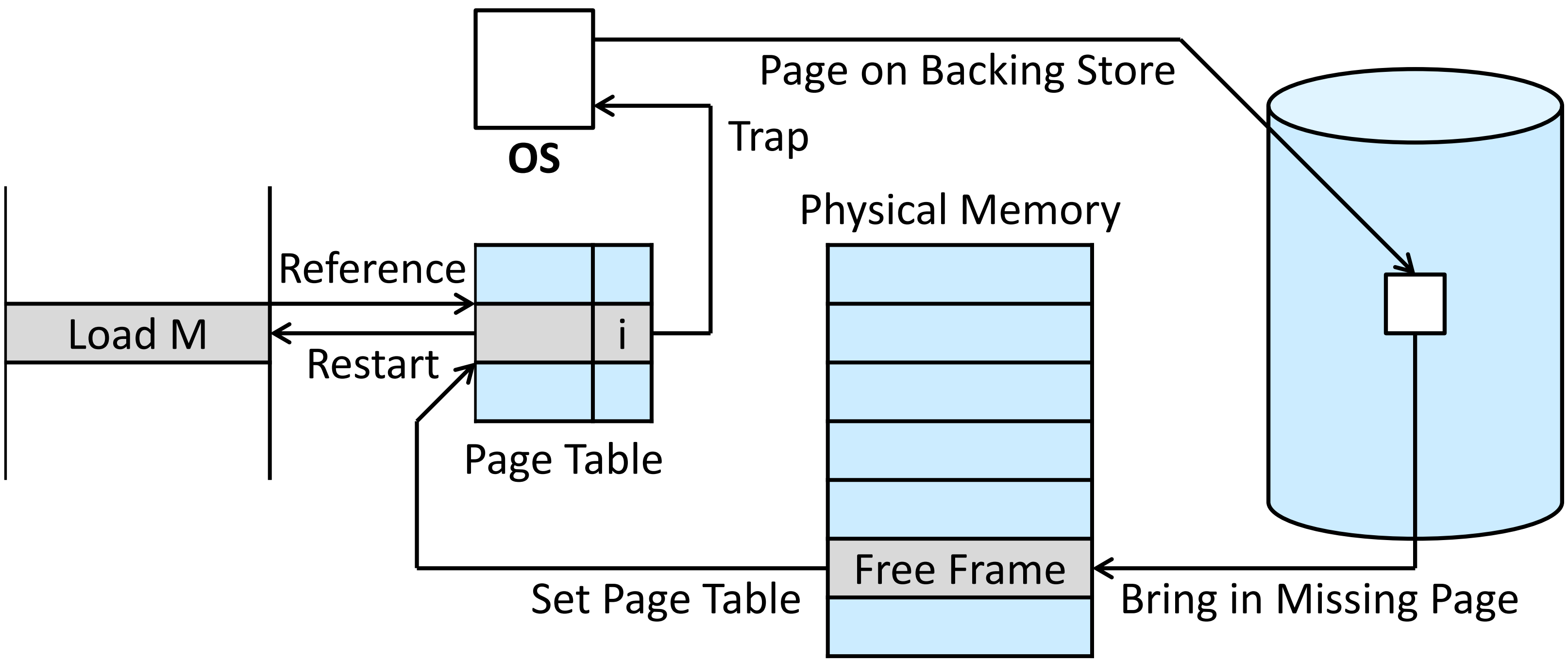
Basic Concepts (2/2)

- 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



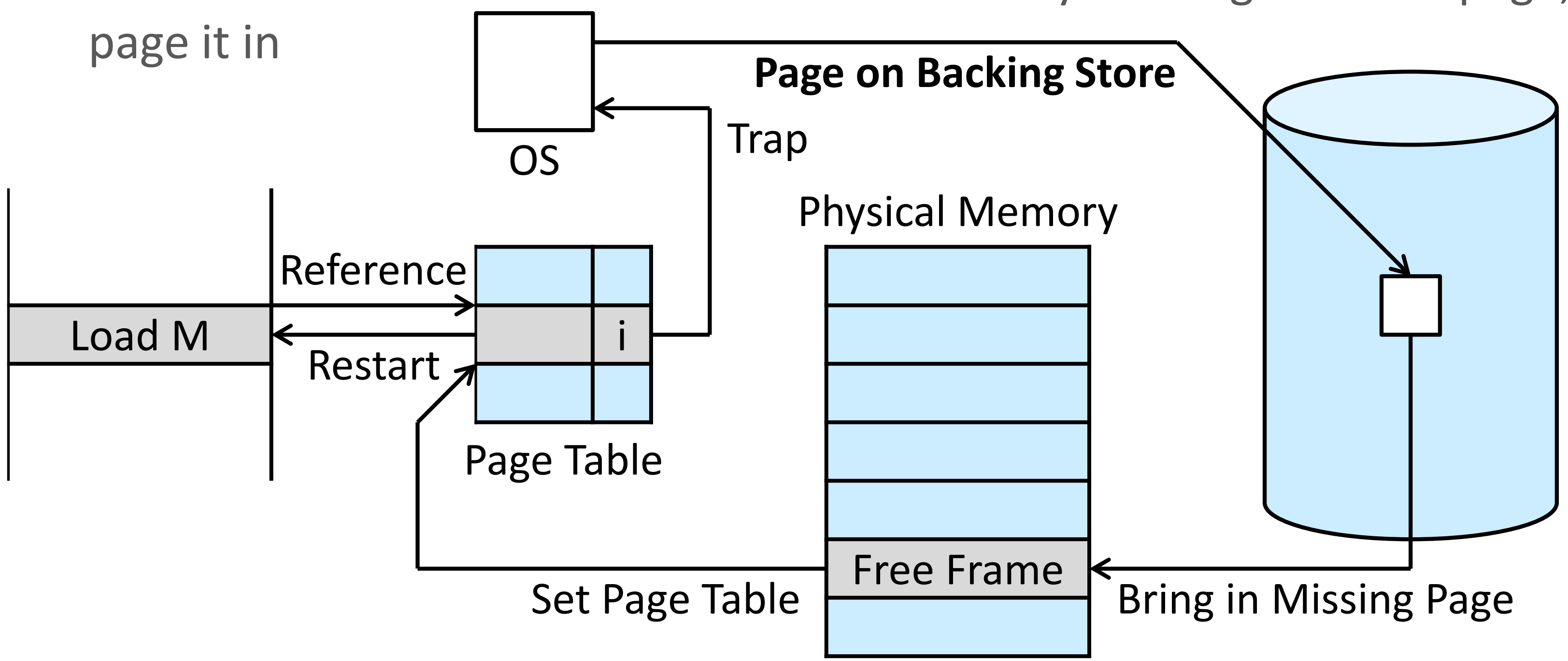
Page Fault Handling

- 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



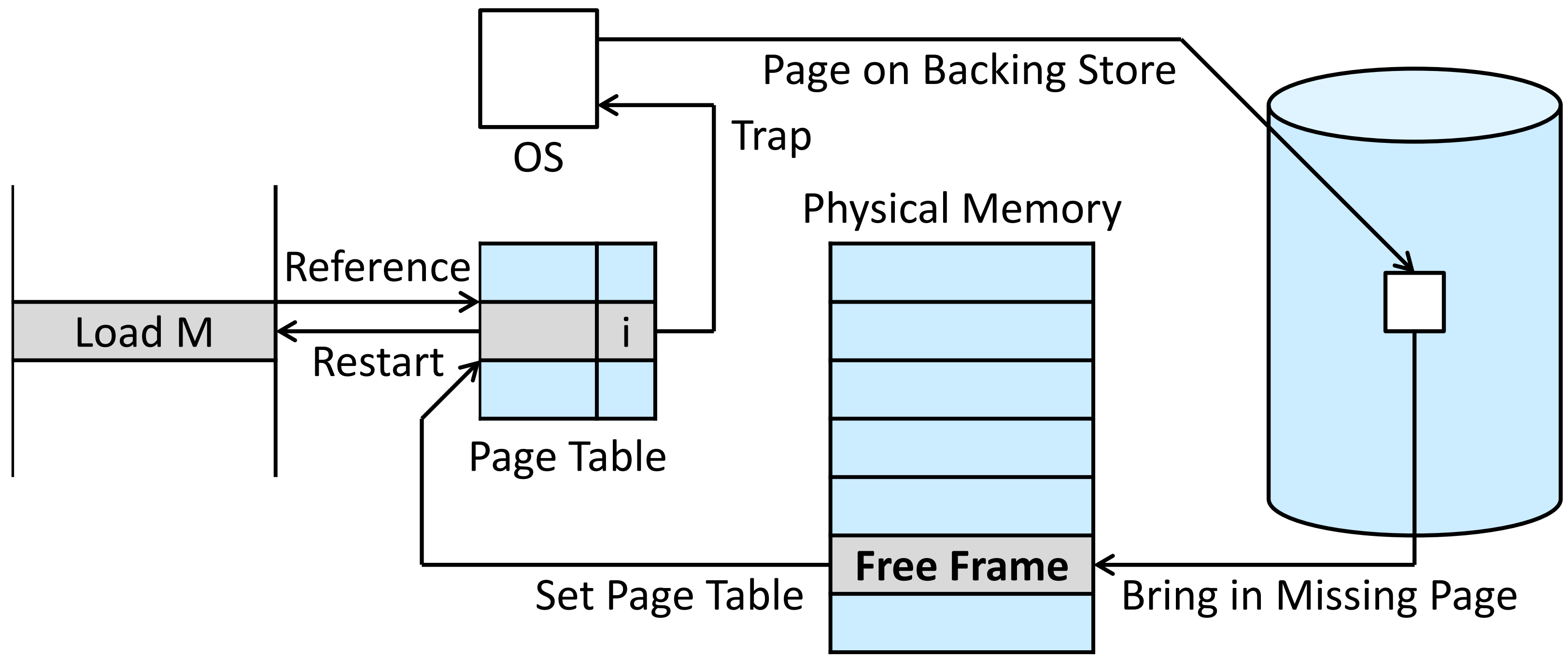
Page Fault Handling

- 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



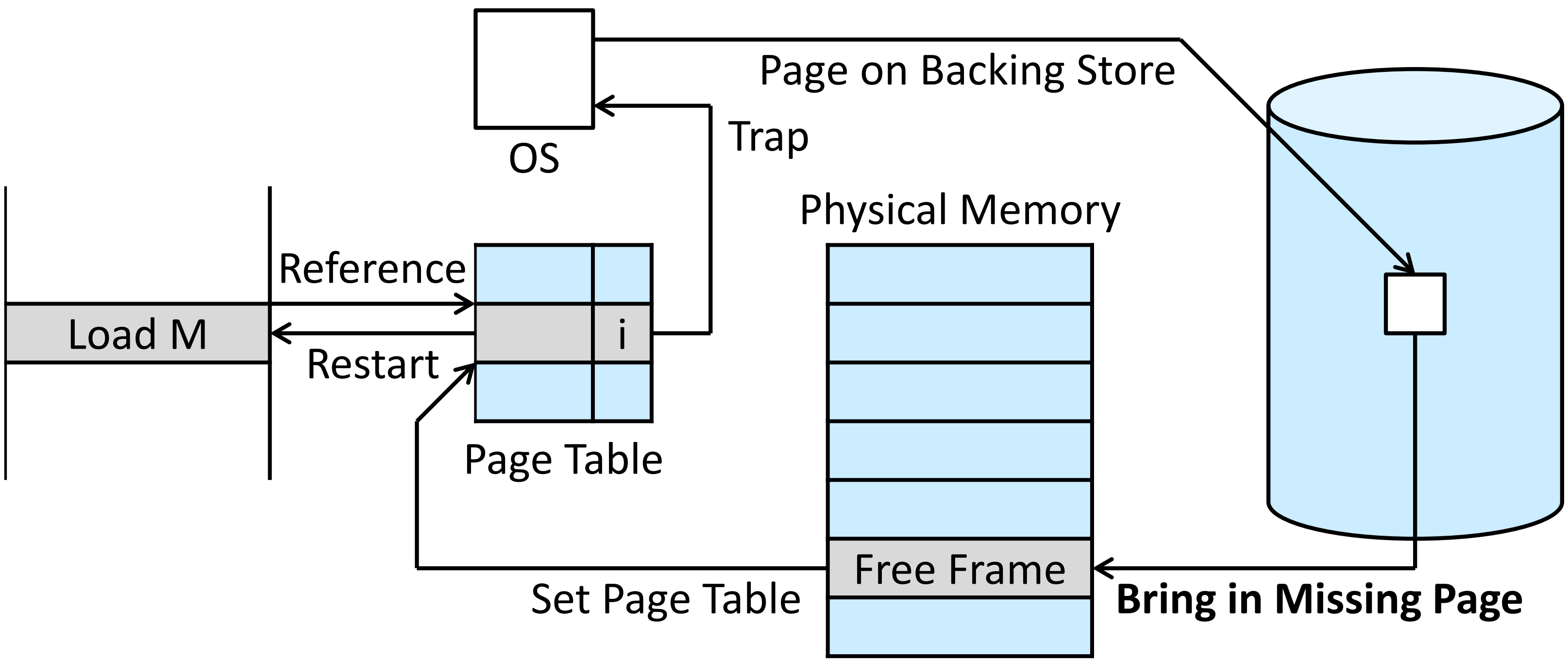
Page Fault Handling

- 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



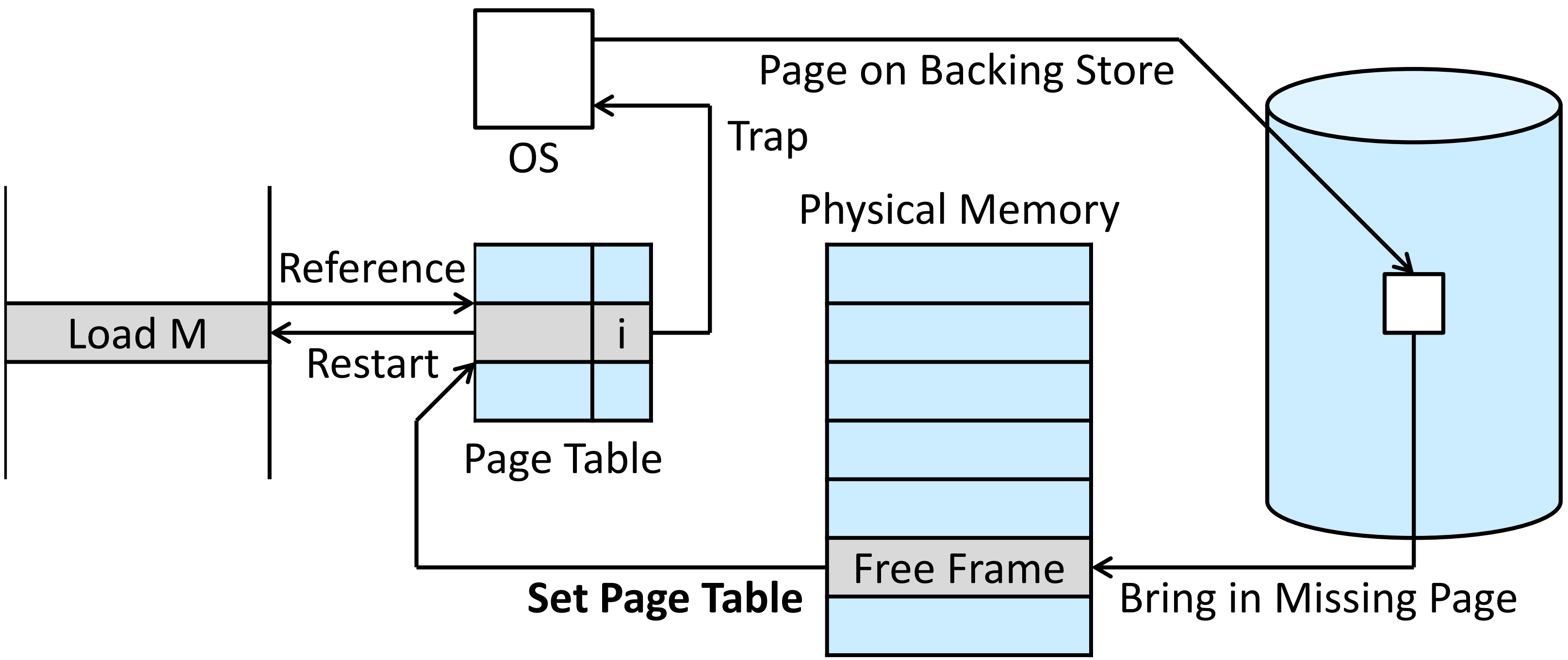
Page Fault Handling

- 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



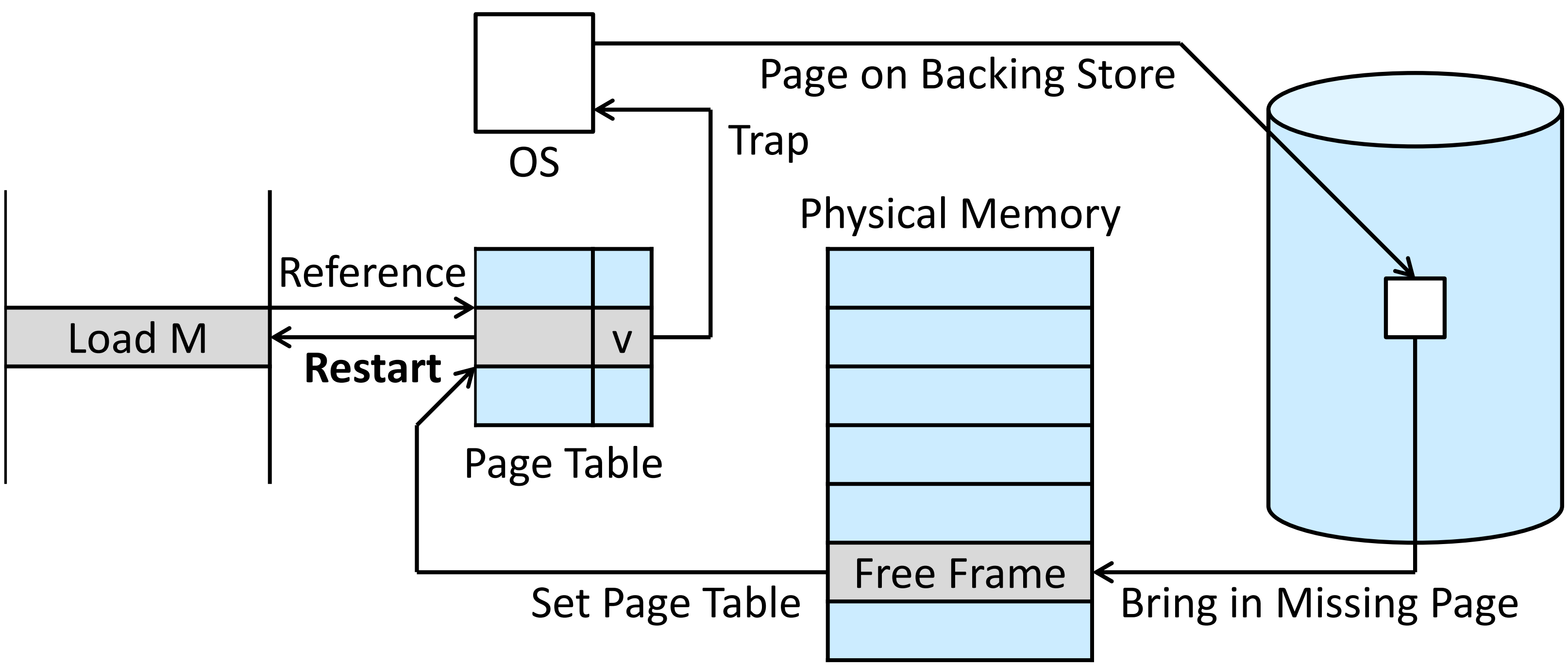
Page Fault Handling

- 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

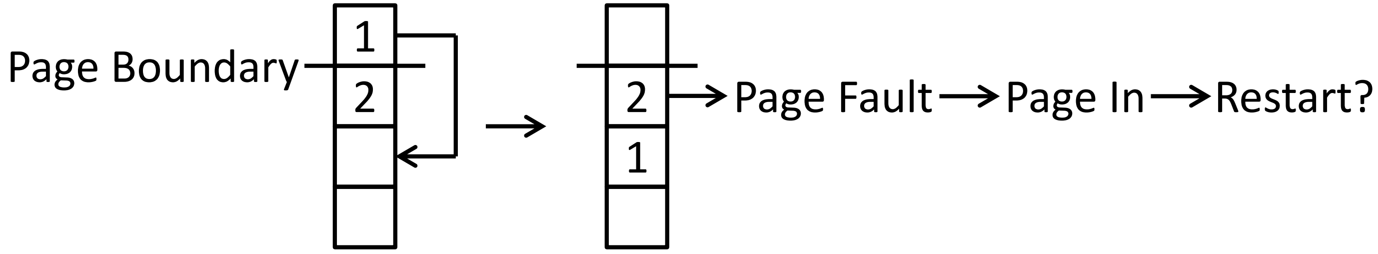
- **Pure demand paging**
  - Never bring a page into memory until it is required
  - Start executing a process with no page in memory
  - Set the instruction pointer to the first instruction (fault immediately)
- **Locality of reference**
  - Theoretically, some programs access several new pages of memory 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

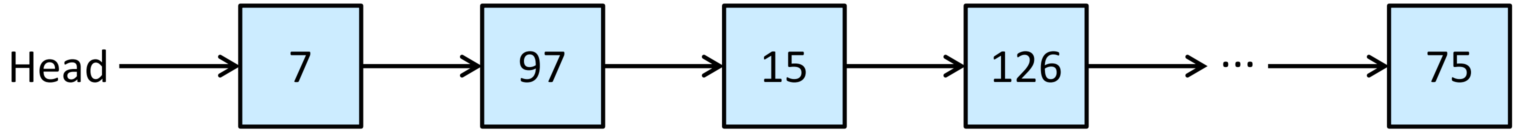


Outline

- 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
  - Probably of a page fault = p
  - EAT = (1 − p) · 200 + p · 8,000,000 = 200 + p · 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)

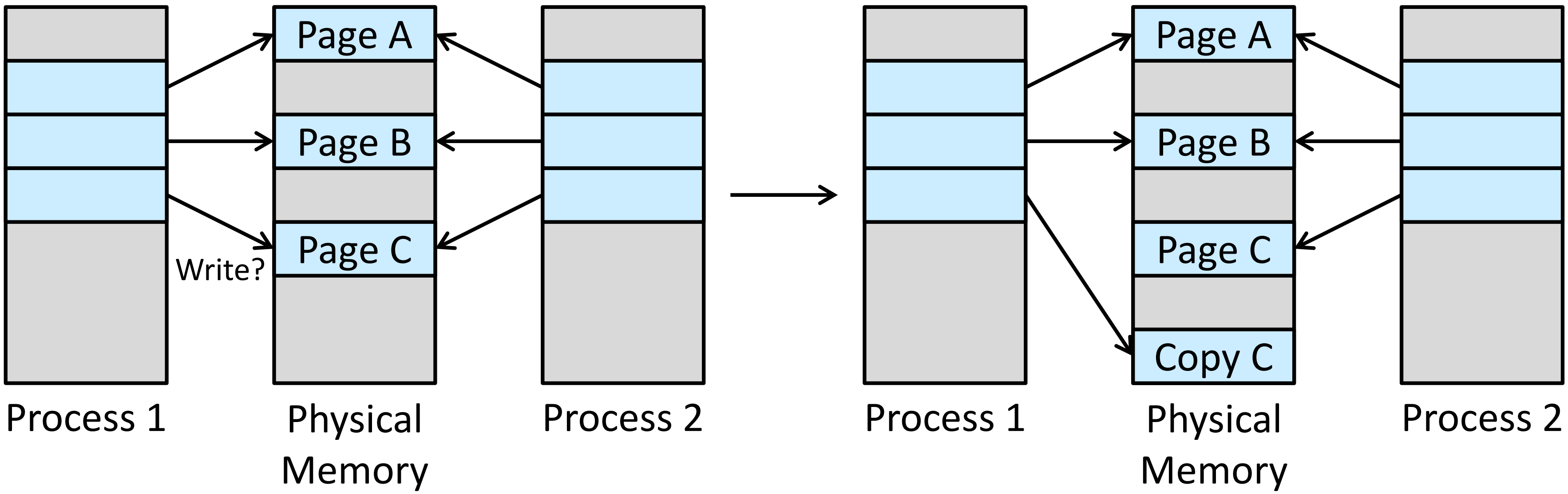
- Some systems attempt to limit the amount of swap space used through demand paging of binary executable files
  - When page replacement is called for
    - These frames can simply be overwritten (because they are never modified)
    - The pages can be read in from the file system again if needed
  - 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

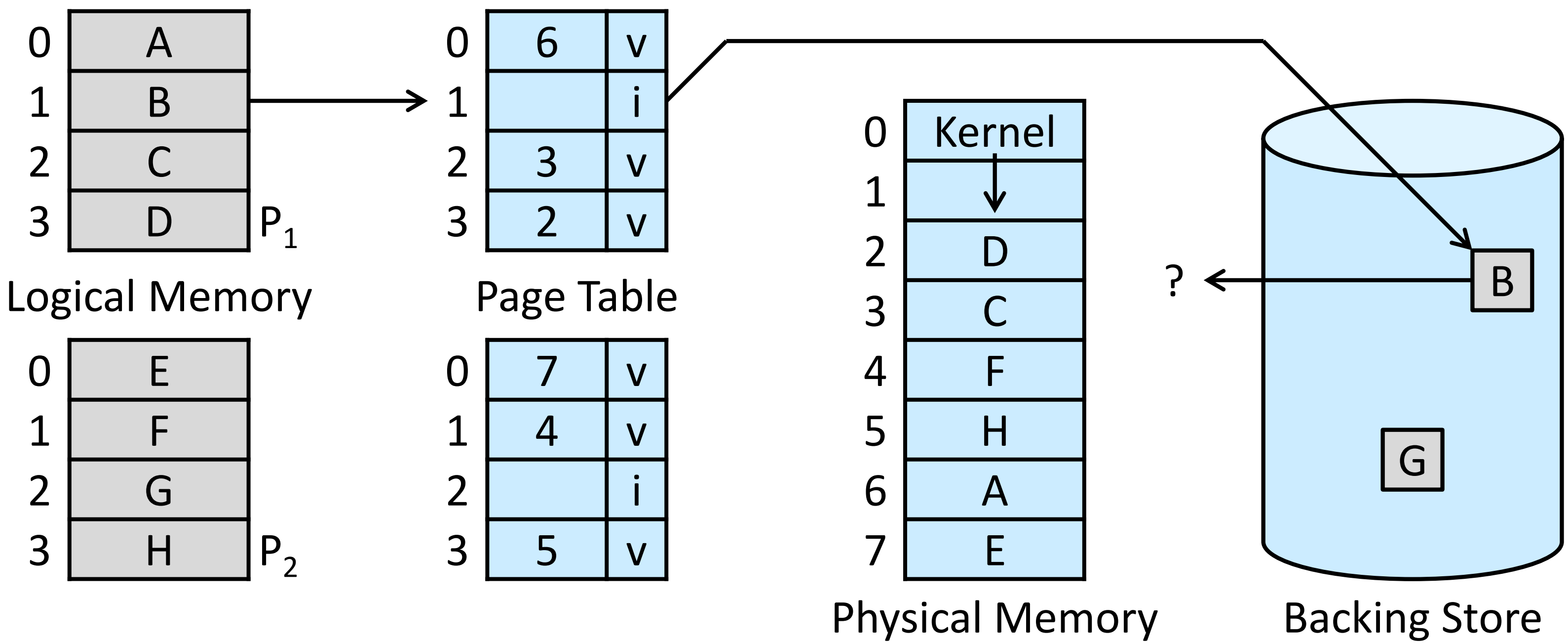
- **`vfork()`**
  - The parent process is suspended
  - The child process uses the address space of the parent
- **`vfork()`** does **<u>not</u>** use copy-on-write
  - If the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes
  - 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**

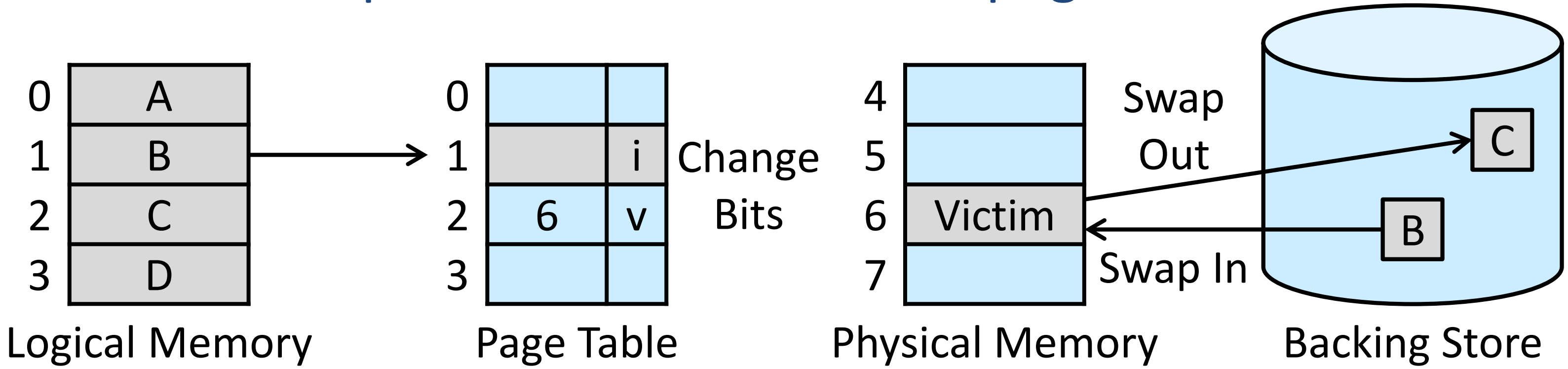


Outline

- 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

Q&A