CSE 4/521 Introduction to Operating Systems

Lecture 15 – Virtual Memory I

(Background, Demand Paging)

Summer 2018

Overview

Objective:

- To describe the benefits of a virtual memory system.
- To explain the concept of demand paging.

- Background
- Demand Paging

Recap

- Paging
  - Address Translation Scheme, Paging Hardware, Translation Look Aside Buffer (TLB)
- Structure of Page Table
  - Hierarchical Page Table, Two-level Page-Table, Hashed Page Table, Inverted Page Table

Questions

1. What is paging? How is it different from the concept of swapping you are familiar with? (Easy)

2. In what way does TLB facilitate better paging? (Medium)

3. Describe a mechanism by which one segment could belong to the address space of two different processes? (Easy)

Overview

- Background
- Demand Paging

Overview

- Background
- Demand Paging

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time

Background

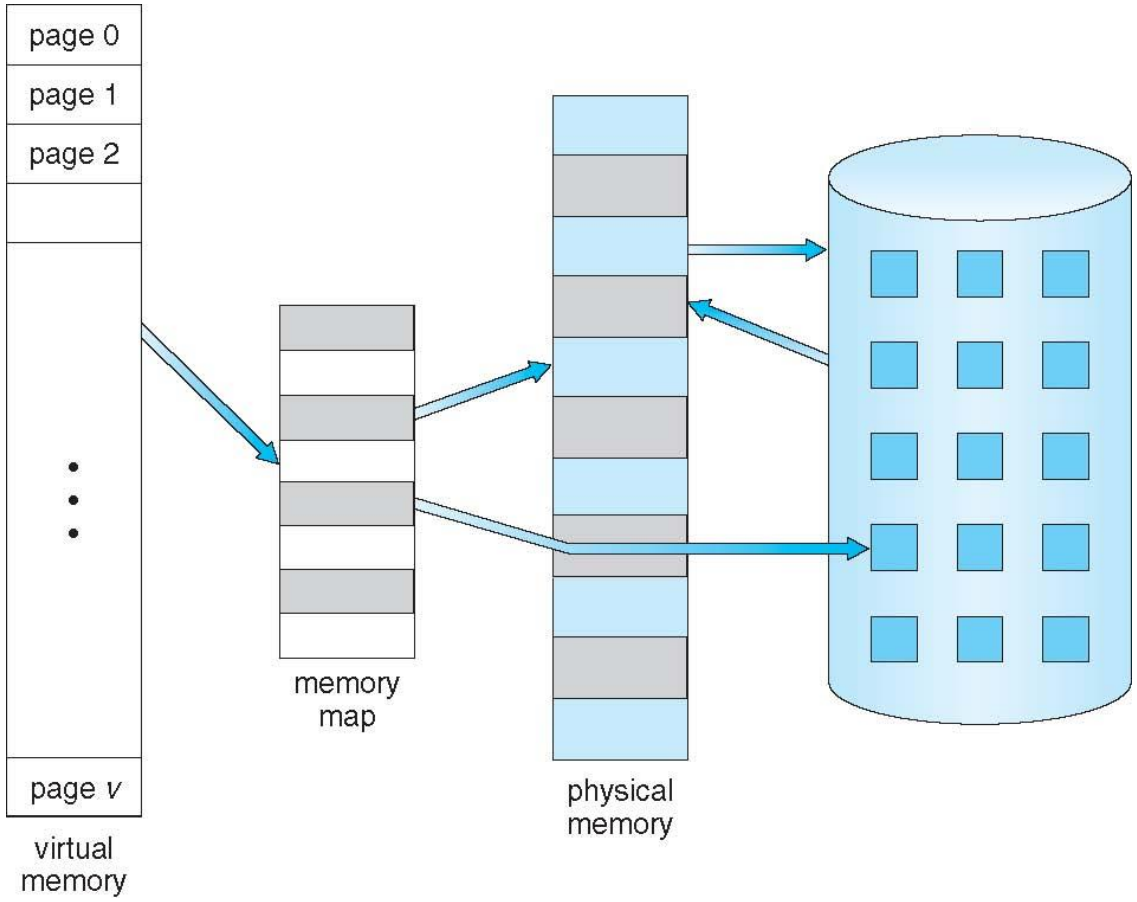
- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - More programs running concurrently

Background

- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

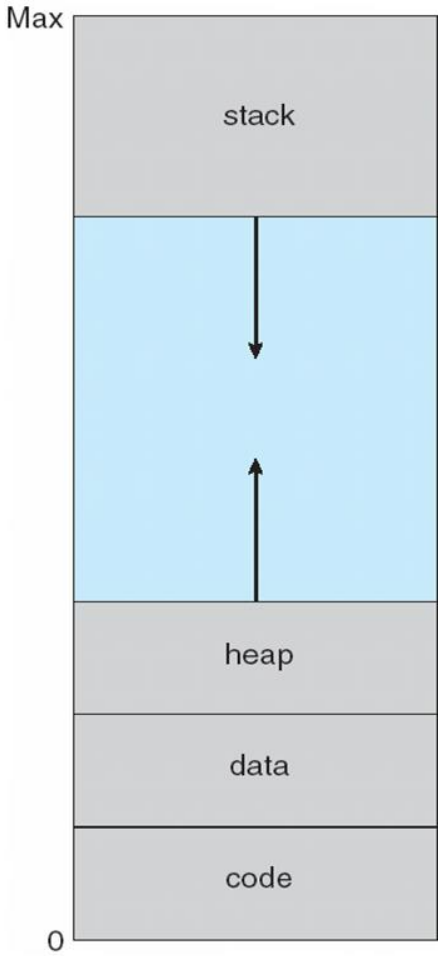
- Virtual memory can be implemented via Demand paging

Background :
Virtual Memory larger than Physical Memory

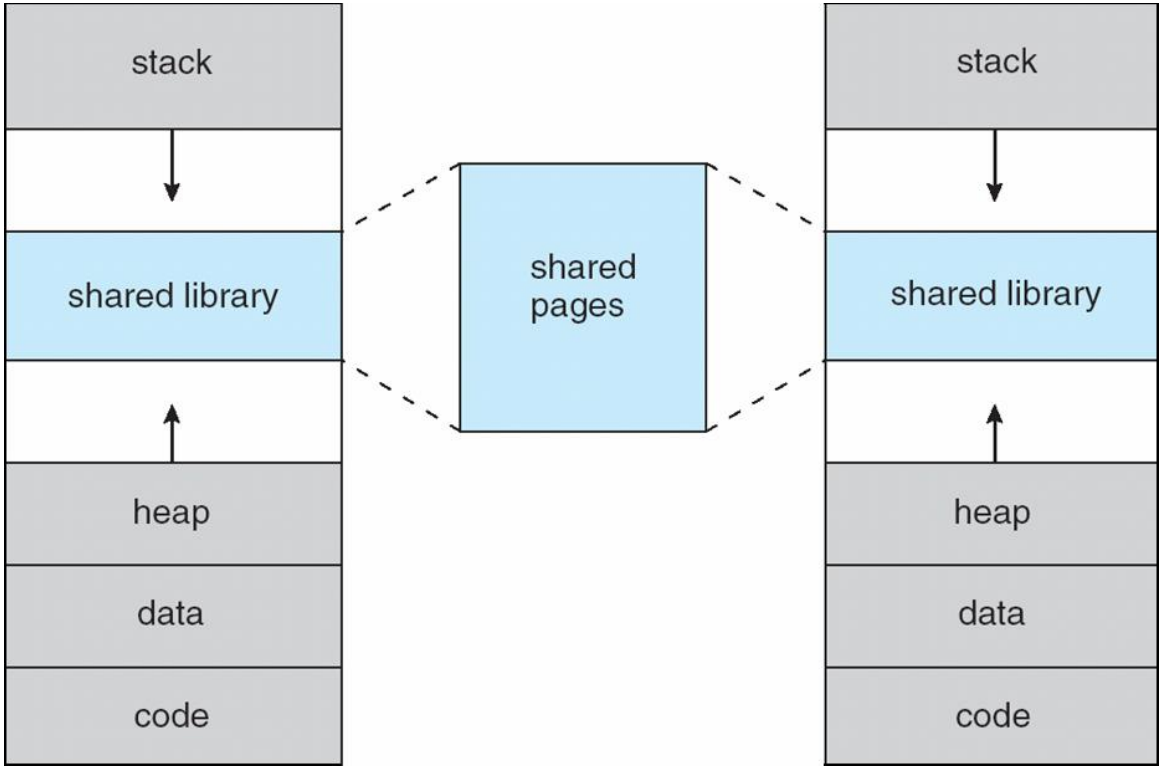


Background: Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Background:
Shared Library Using Virtual Memory

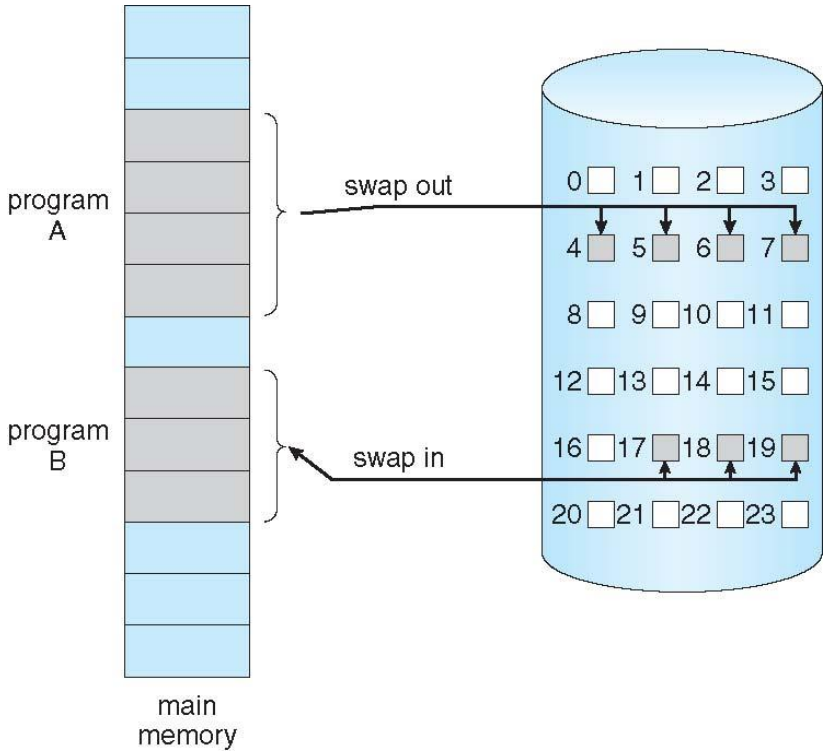


Overview

- Background
- Demand Paging

Demand Paging

- 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
- Similar to paging system with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

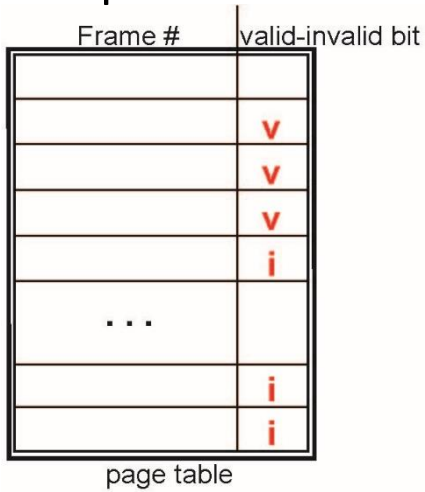


Demand Paging : Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage

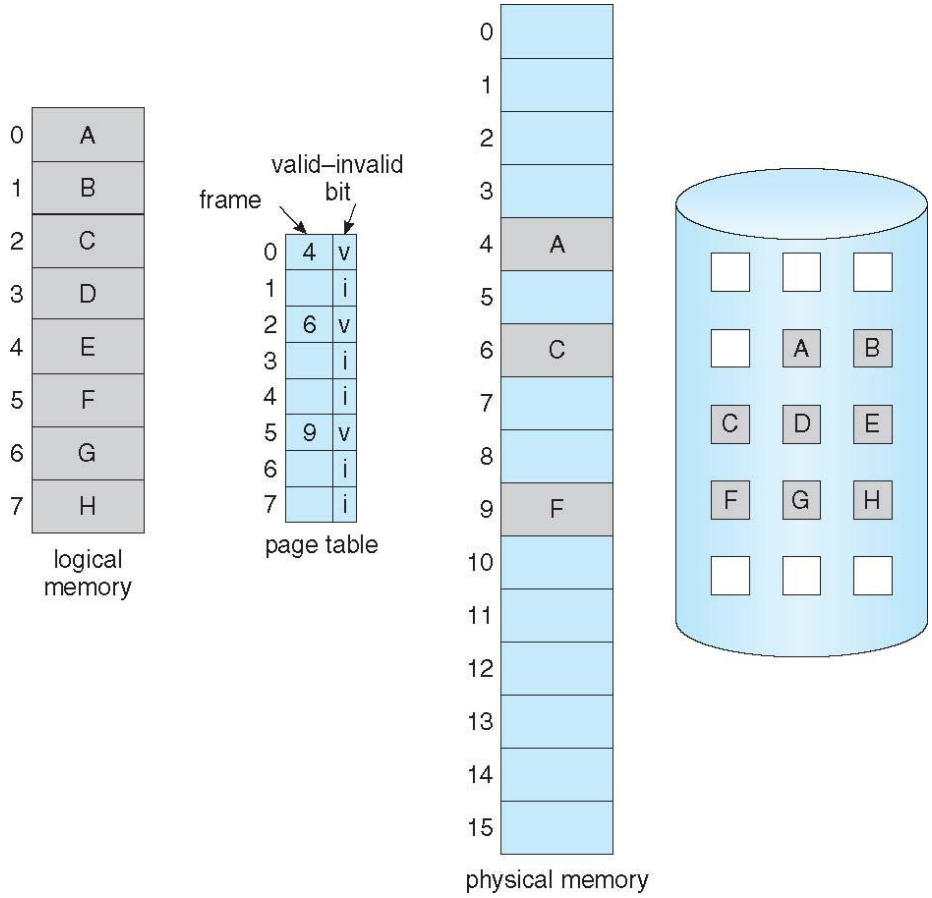
Demand Paging : Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇒ memory resident, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is i ⇒ page fault

Demand Paging : Page Table when Some Pages are not in Main Memory



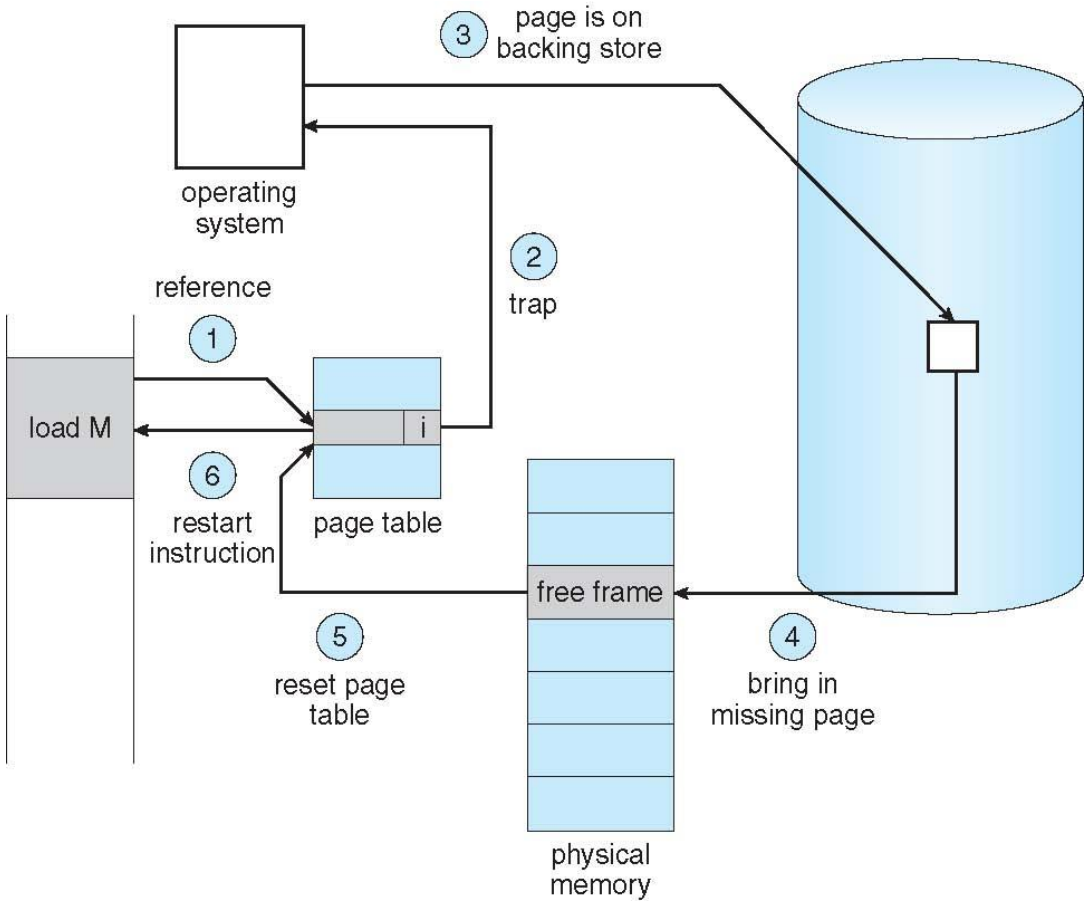
Demand Paging : Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
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

Demand Paging : Steps in Handling a Page Fault



Demand Paging : 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
- In reality, an 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 (swap device with swap space)

Demand Paging : Demand Paging Example

- Three major activities
  - Service the interrupt
  - Read the page
  - Restart the process
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access}$$
$$+ p \text{ (page fault overhead + swap page out + swap page in)}$$

Demand Paging : Demand Paging Example

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

  EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 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 : Performance of Demand Paging

Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Credits for slides

Silberschatz, Galvin and Gagne