10장. 가상 메모리 (1)

운영체제

목차

학습목표

- 가상메모리를 정의하고 그 이점을 설명한다.
- 요구 페이징을 사용하여 페이지가 메모리에 적재되는 방법을 설명한다.
- FIFO, 최적 및 LRU 페이지 교체 알고리즘을 적용한다.
- 프로세스의 작업집합을 설명하고, 프로그램 지역성과 어떤 관련이 있는지 설명한다.
- Linux, Windows 10 및 Solaris 가상메모리를 관리하는 방법을 설명한다.

10.1 BACKGROUND

10. Virtual Memory

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual memory

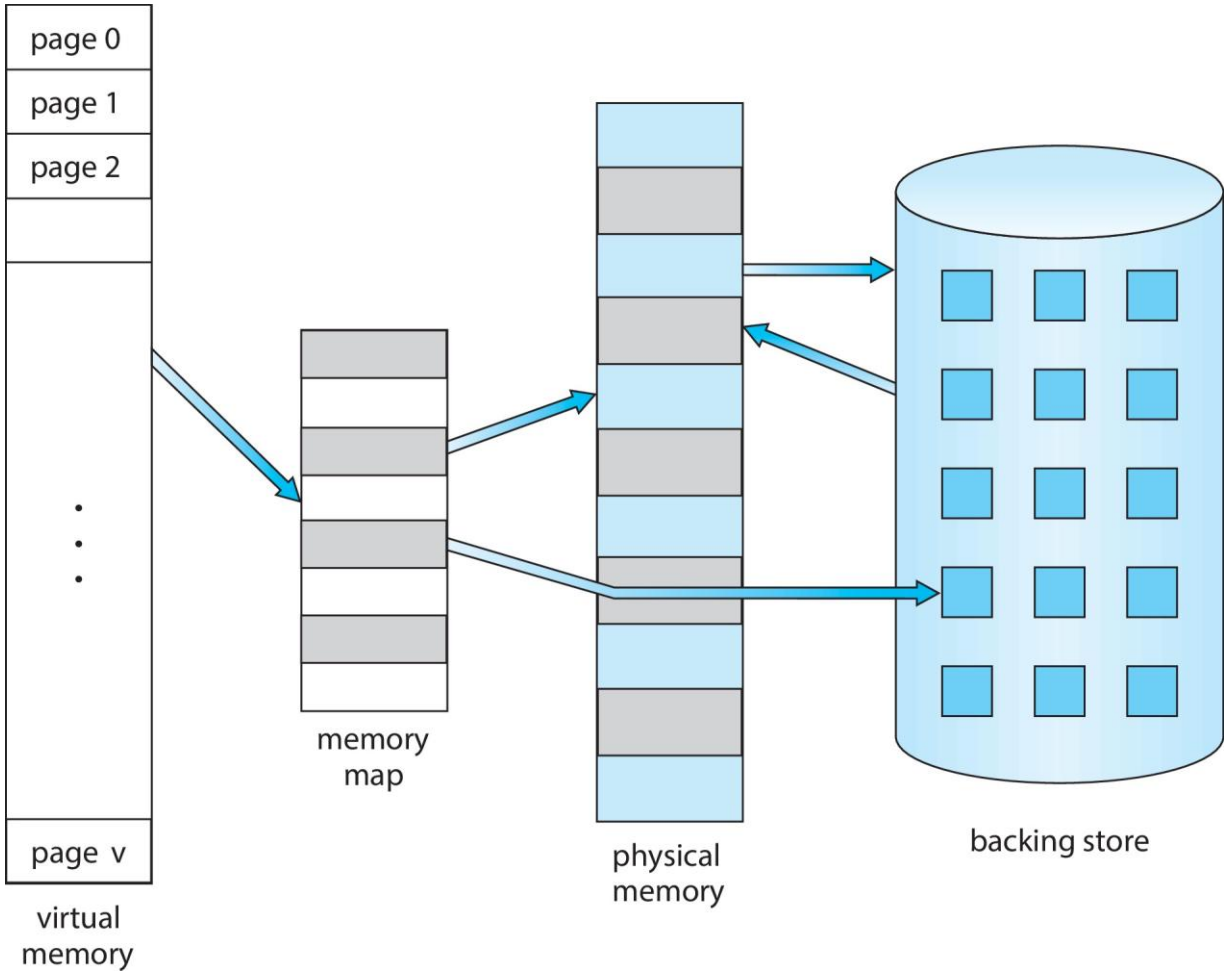
- **Virtual memory** – separation of user logical memory from physical memory
  - **Only part** of the program **needs** to be in memory for execution
  - Logical address space can therefore be much **larger** than physical address space
  - Allows address spaces to be **shared** by several processes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

Virtual memory (Cont.)

- **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

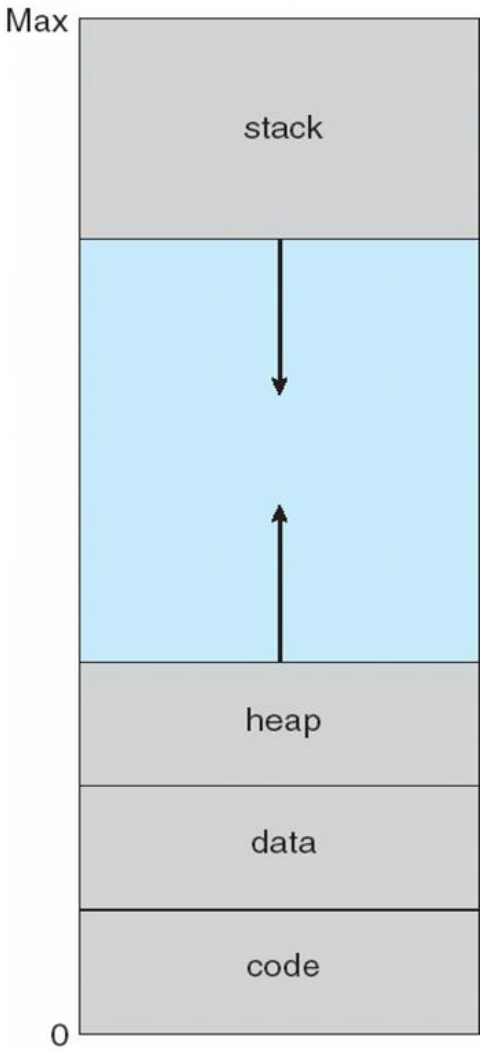
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

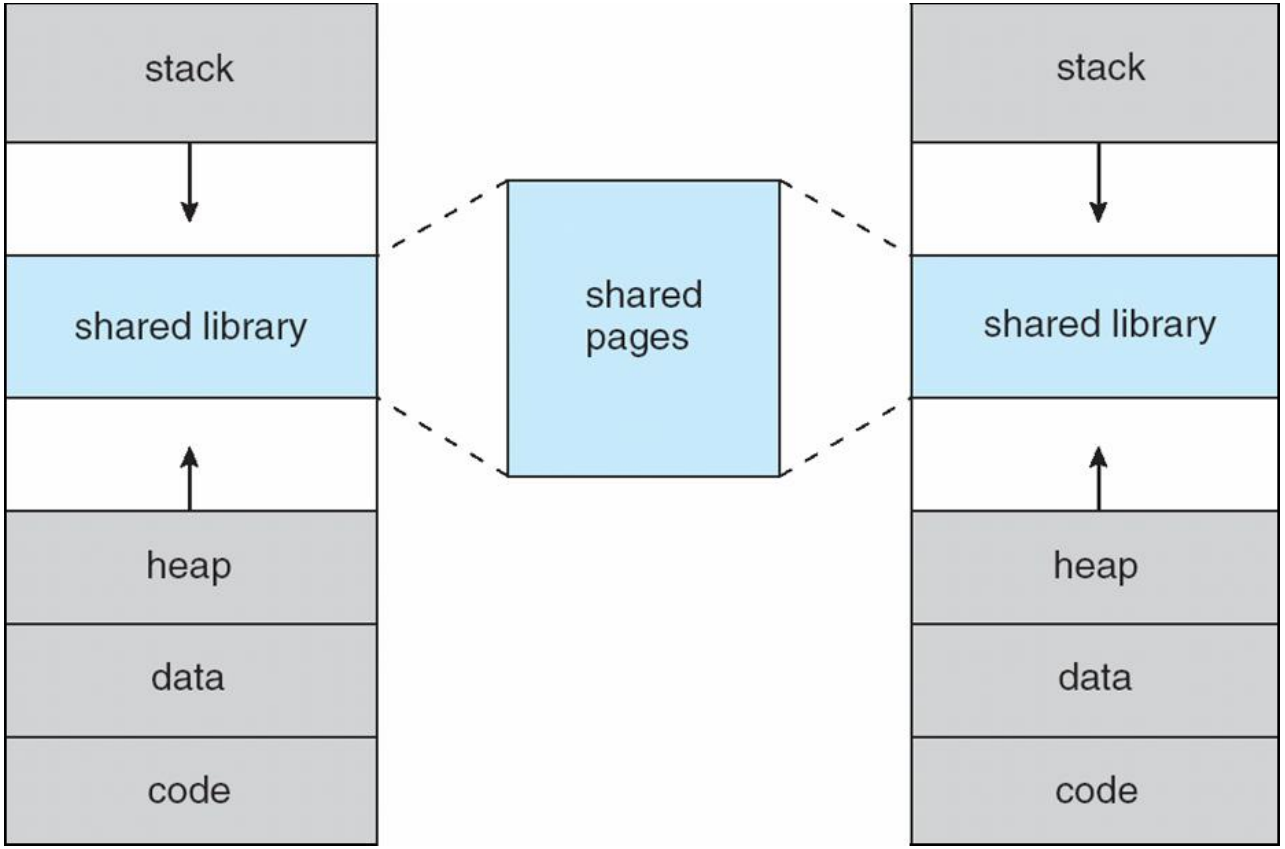


Virtual-address Space

- Usually design logical address space for the 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
- 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
- Pages can be shared during fork(), speeding process creation



Shared Library Using Virtual Memory

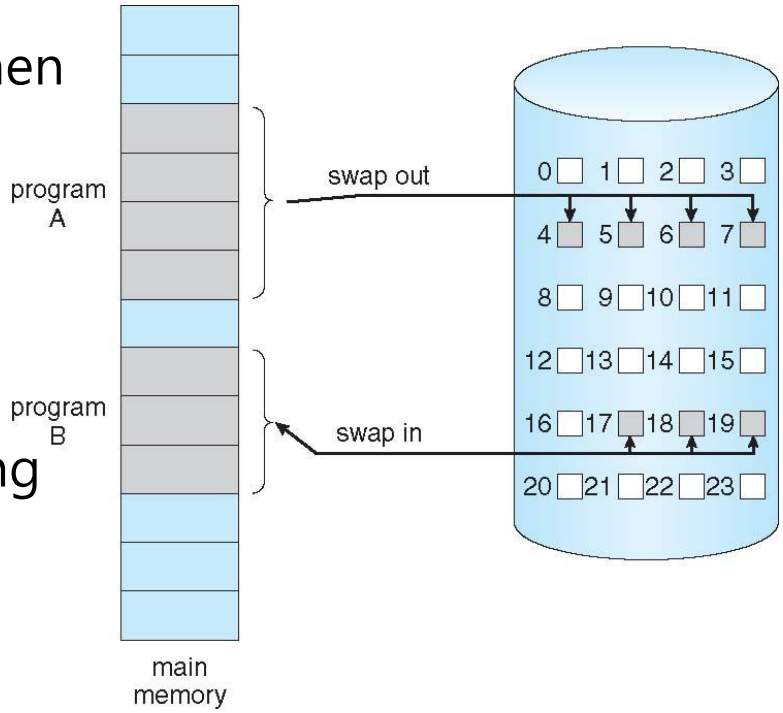


10.2 DEMAND PAGING

10. 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
- Similar to paging system with swapping (diagram on right)
- 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

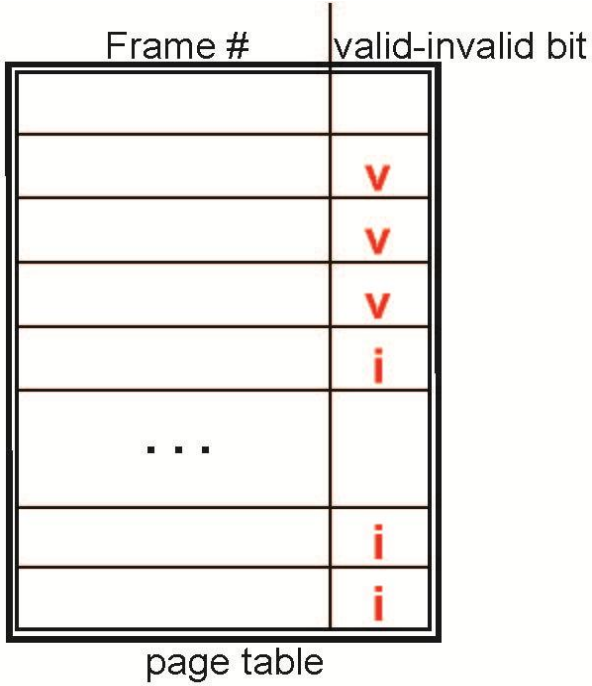


10.2.1 Basic Concepts

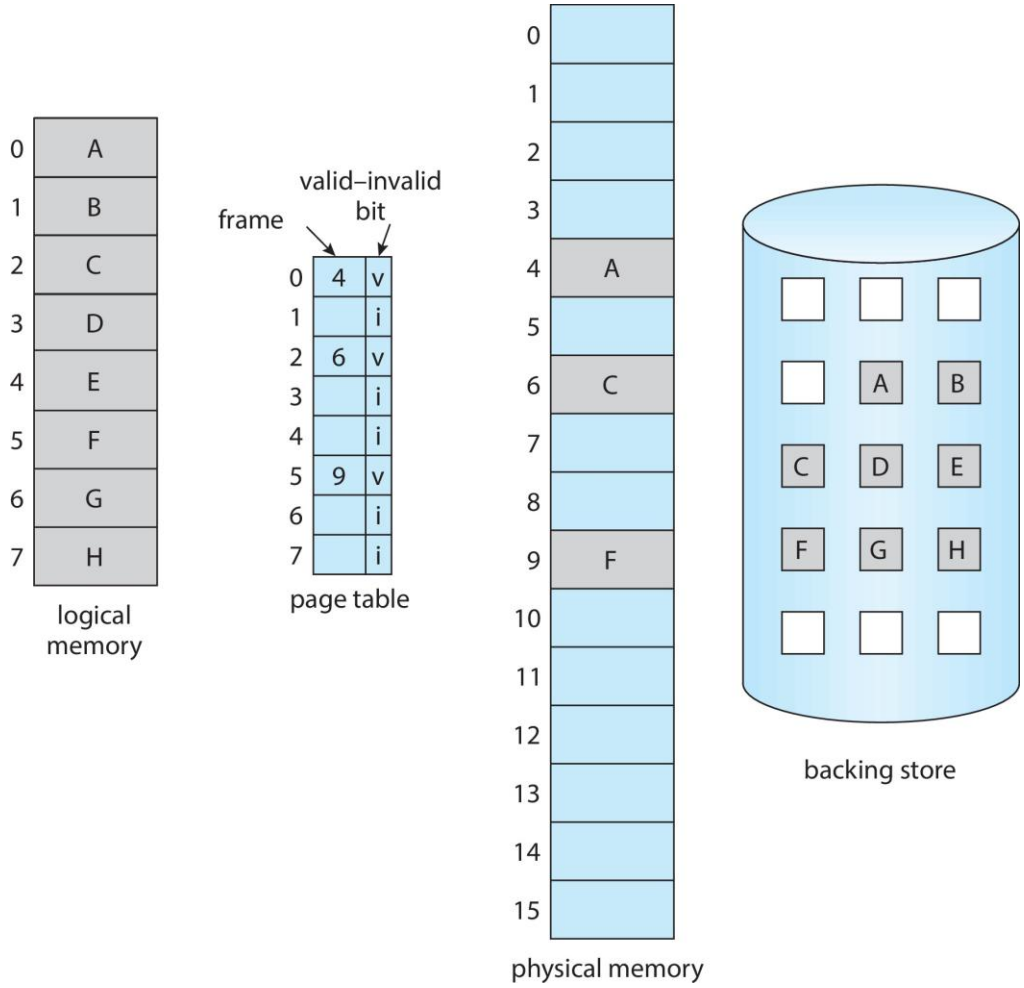
- With swapping, the pager **guesses** which pages will be used before swapping them out again
- How to determine that **set of pages**?
- Need new **MMU** functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non demand-paging
- If page needed and **not** memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code
- Use page table with **valid-invalid** bit (see chapter 9)

Page table with Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, 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 the page table entry is i ⇒ page fault



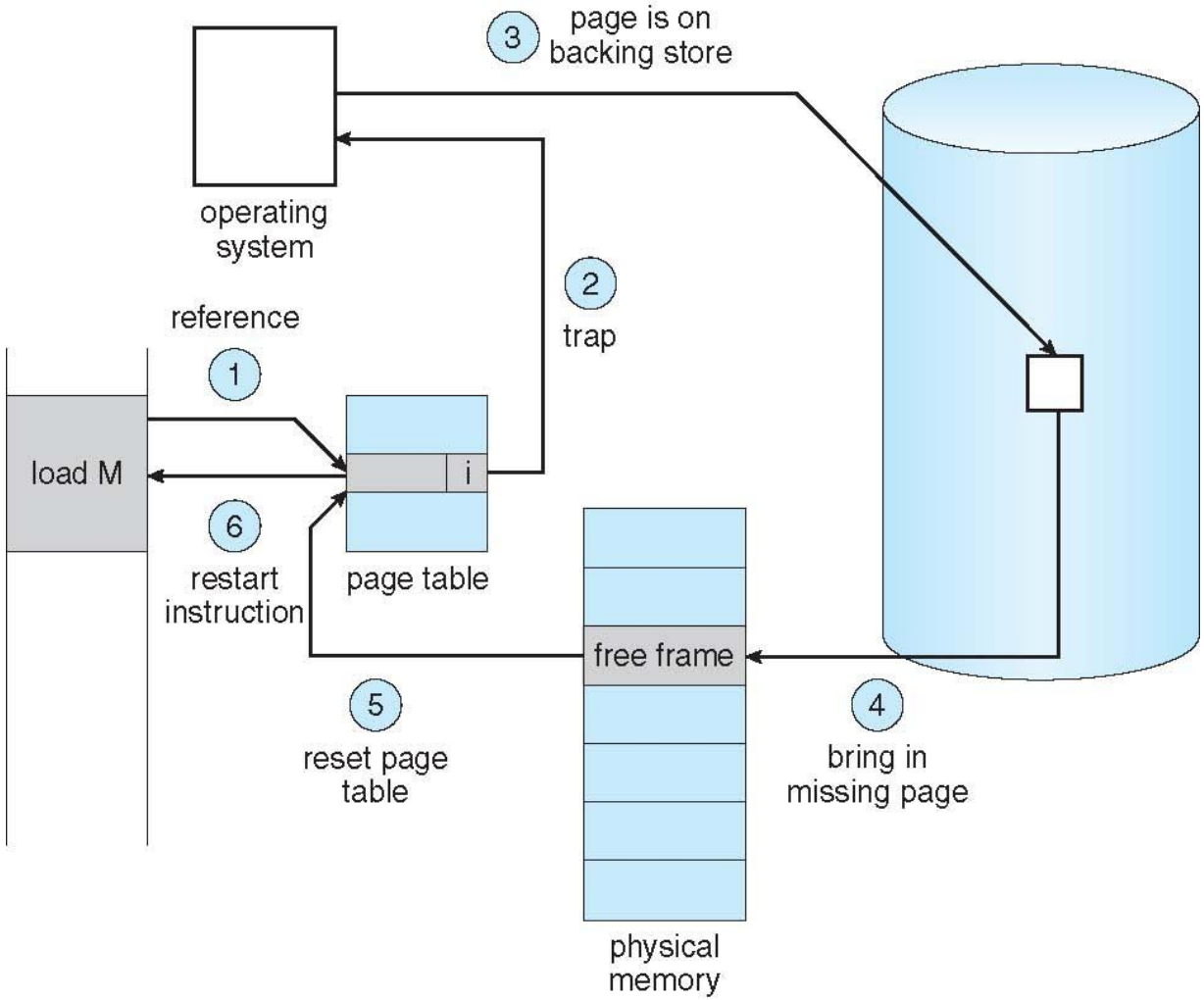
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = v
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)

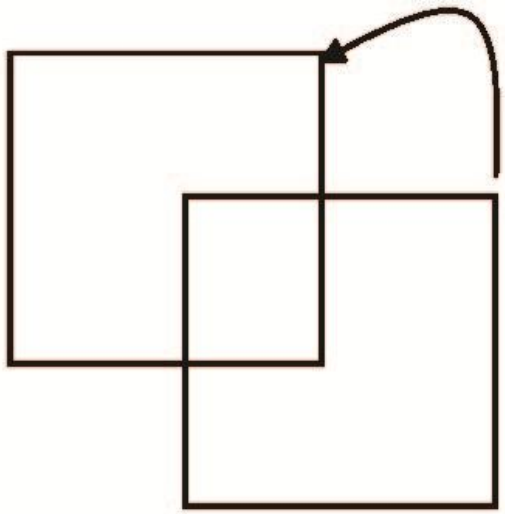


Aspects of Demand Paging

- **Pure demand paging:** 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
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

10.2.2 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** -- a pool of free frames for satisfying such requests.

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

10.2.3 Performance of Demand Paging

- 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)

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

Stages in Demand Paging

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

Stages in Demand Paging (cont.)

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

Performance of Demand Paging

- Three major activities
  1. Service the interrupt – careful coding means just several hundred instructions needed
  2. Input the page from disk – lots of time
  3. Restart the process – again just a small amount of time
- 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)

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

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p x (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 = 8,200 nanoseconds
  - 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

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks; less management needed than file system
- 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 Paging Optimizations (cont.)

- Demand page in from **program binary** on disk, but **discard** rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages **modified in memory** but not yet written back to the file system
- Mobile systems
  - Typically, don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

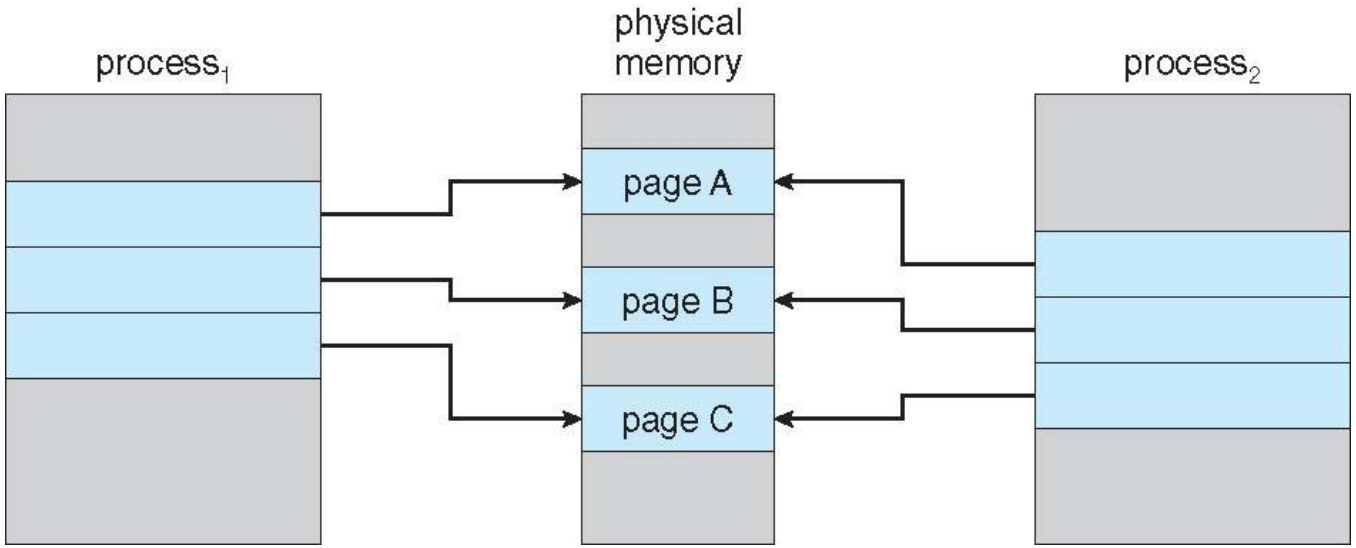
10.3 COPY-ON-WRITE

10. Virtual Memory

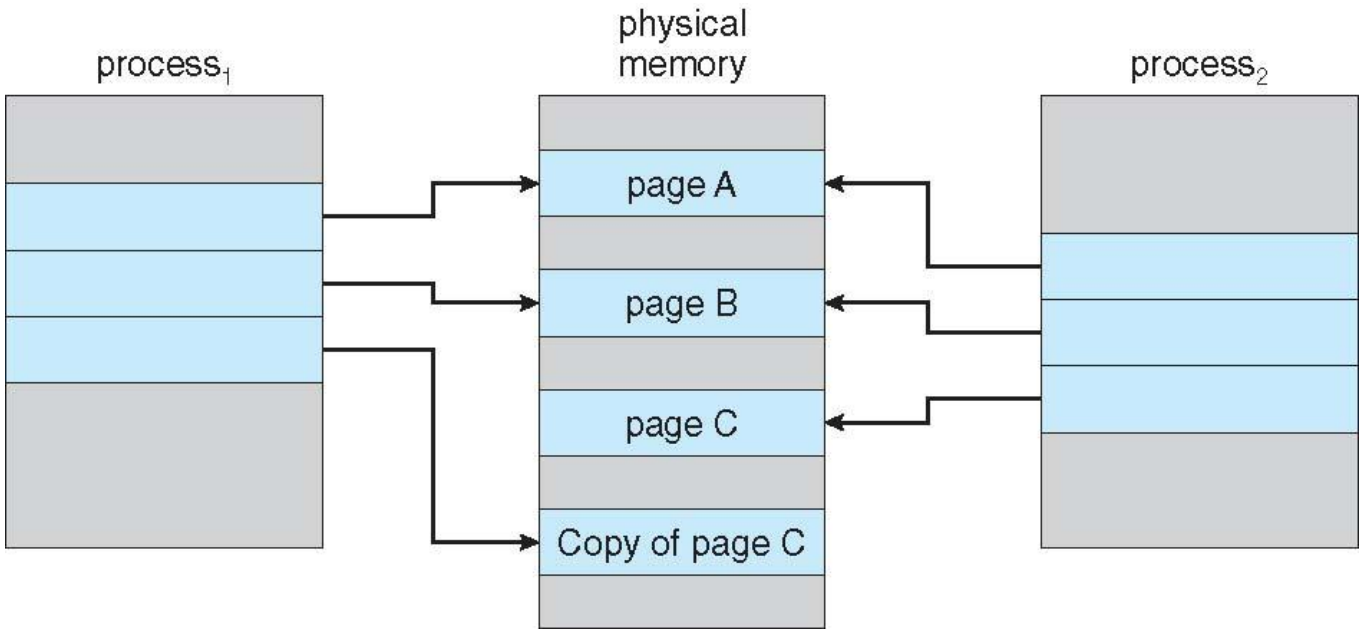
Copy-on-Write

- **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
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

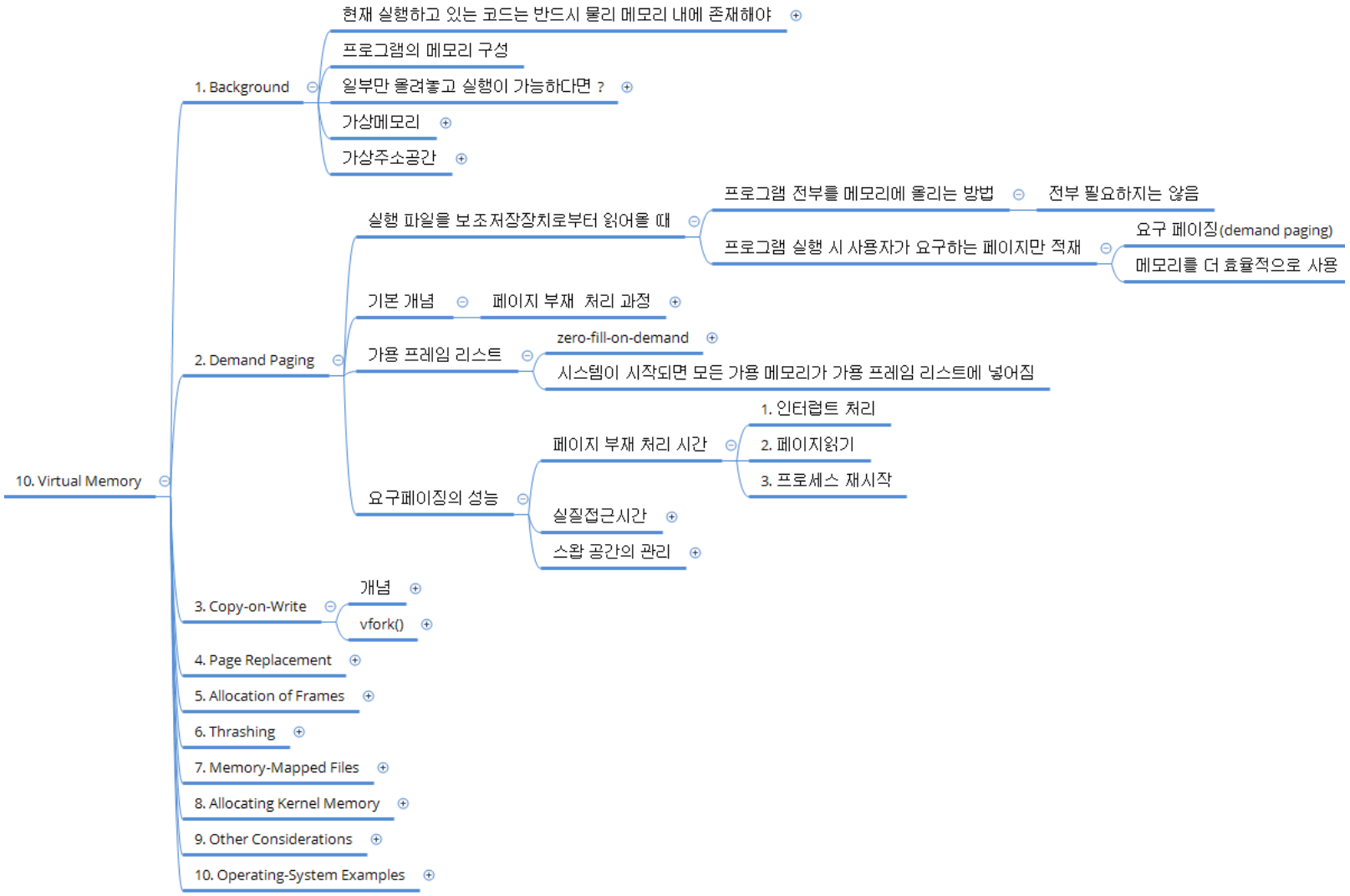
Before Process 1 Modifies Page C



After Process 1 Modifies Page C



학습한 내용


10. Virtual Memory
1. Background
현재 실행하고 있는 코드는 반드시 물리 메모리 내에 존재해야
프로그램의 메모리 구성
일부만 올려놓고 실행이 가능하다면 ?
가상메모리
가상주소공간
2. Demand Paging
실행 파일을 보조저장장치로부터 읽어올 때
프로그램 전부를 메모리에 올리는 방법
전부 필요하지는 않음
프로그램 실행 시 사용자가 요구하는 페이지만 적재
요구 페이징(demand paging)
메모리를 더 효율적으로 사용
기본 개념
페이지 부재 처리 과정
가용 프레임 리스트
zero-fill-on-demand
시스템이 시작되면 모든 가용 메모리가 가용 프레임 리스트에 넣어짐
요구페이징의 성능
페이지 부재 처리 시간
1. 인터럽트 처리
2. 페이지읽기
3. 프로세스 재시작
실질접근시간
스왑 공간의 관리
3. Copy-on-Write
개념
vfork()
4. Page Replacement
5. Allocation of Frames
6. Thrashing
7. Memory-Mapped Files
8. Allocating Kernel Memory
9. Other Considerations
10. Operating-System Examples