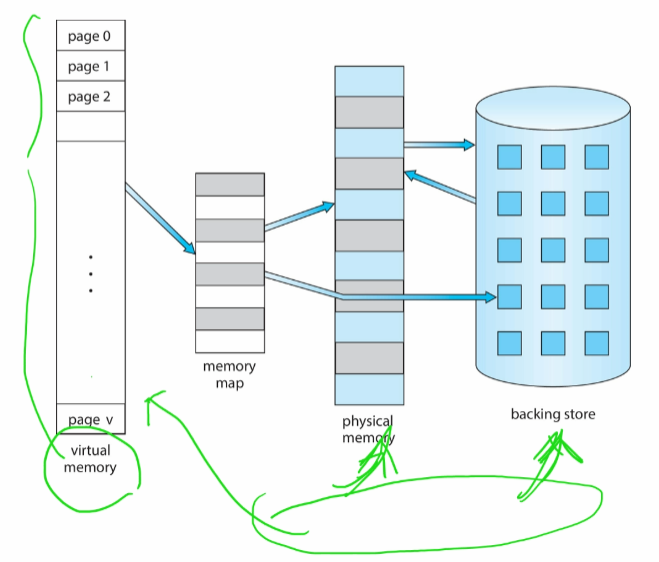
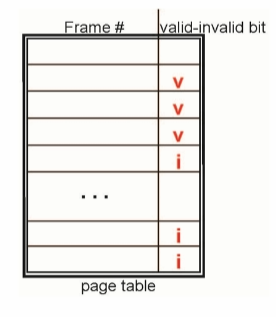
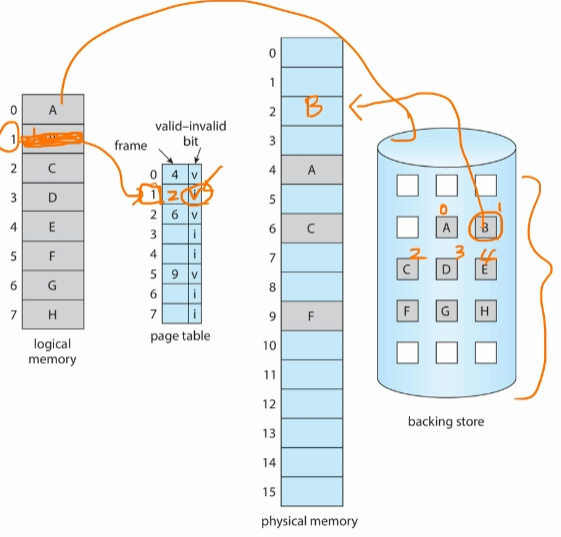
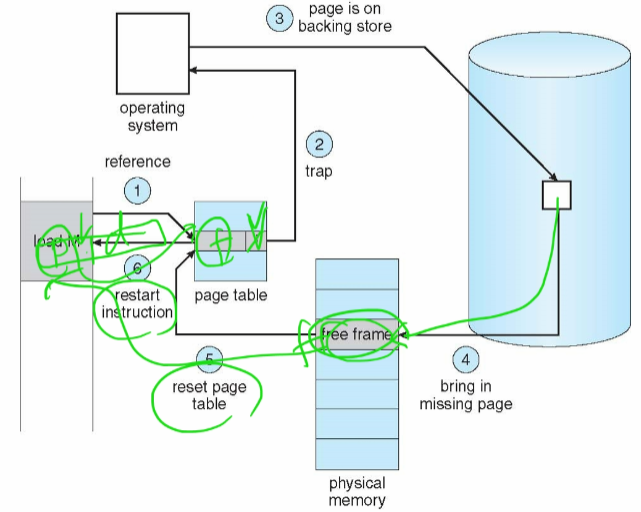
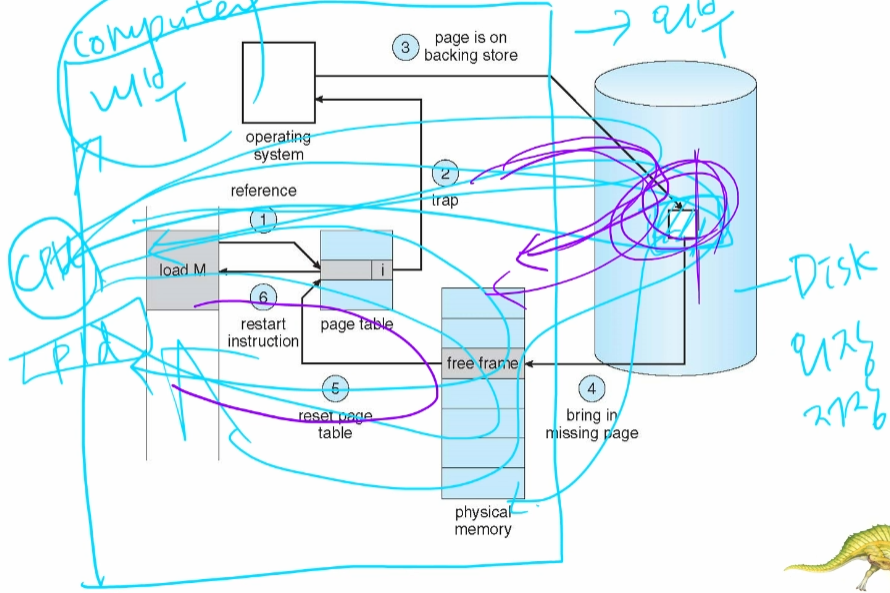
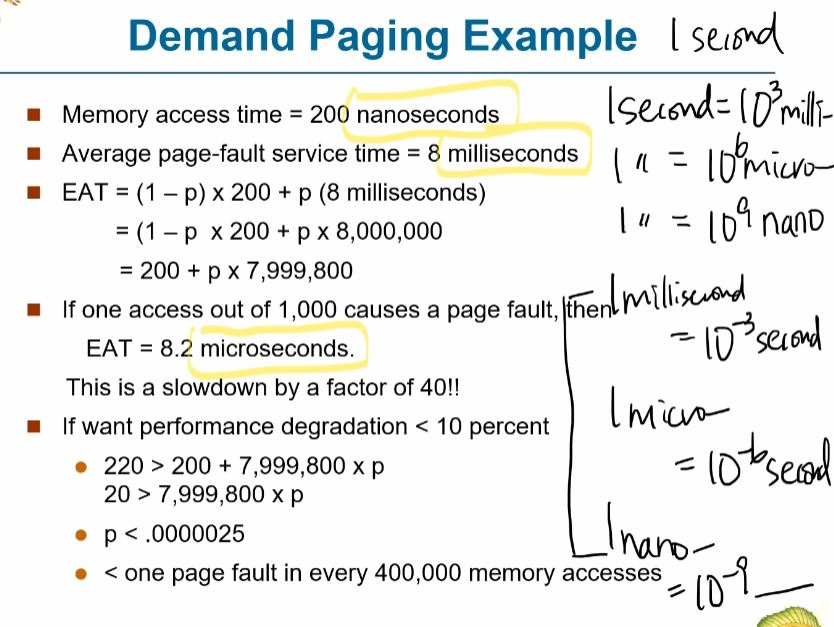
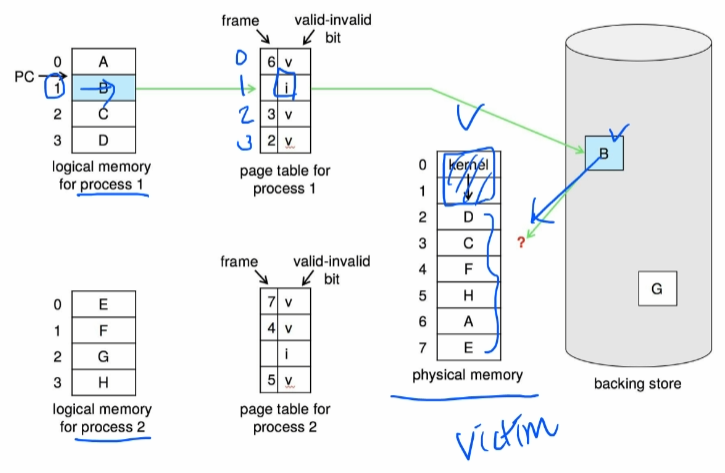
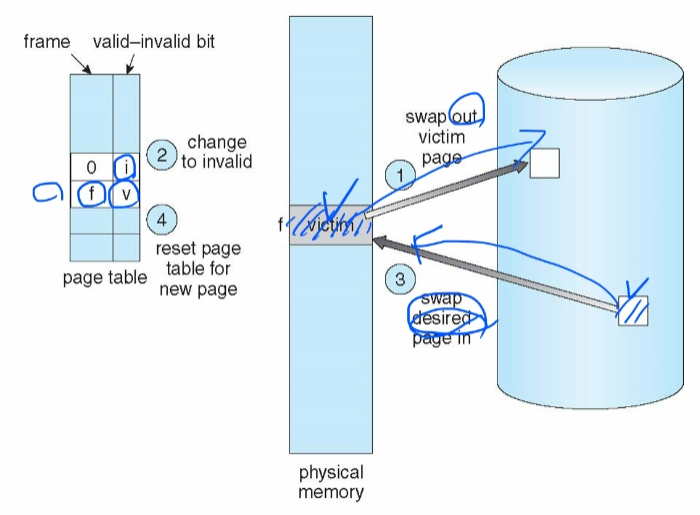
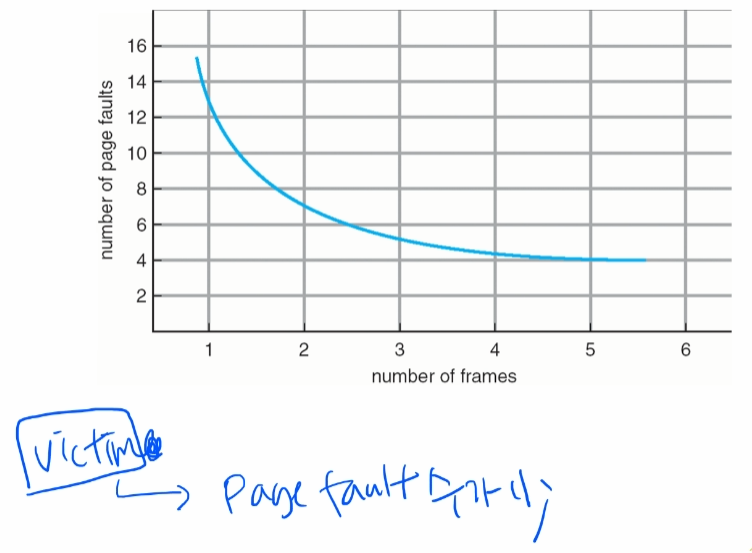
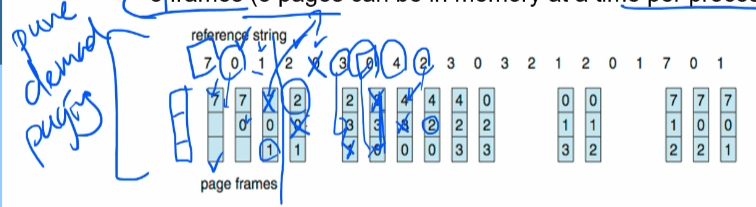
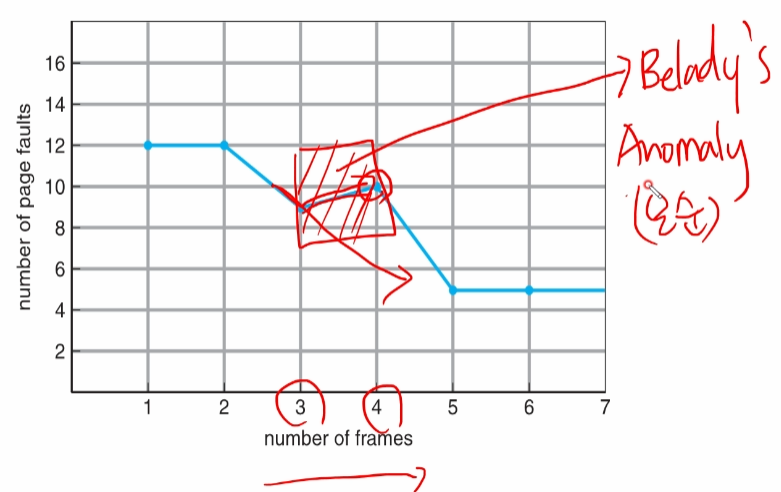
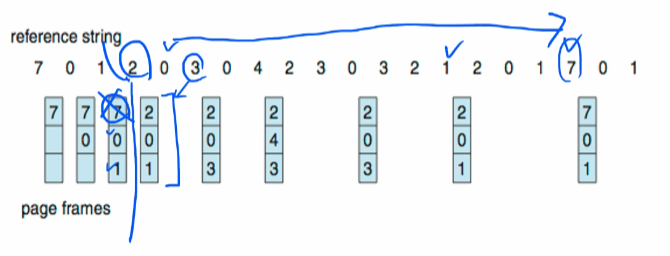
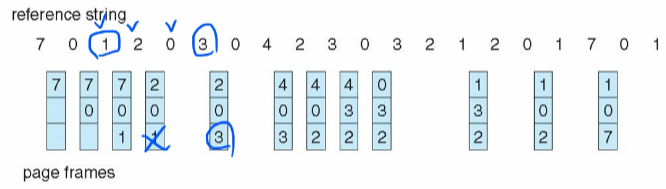
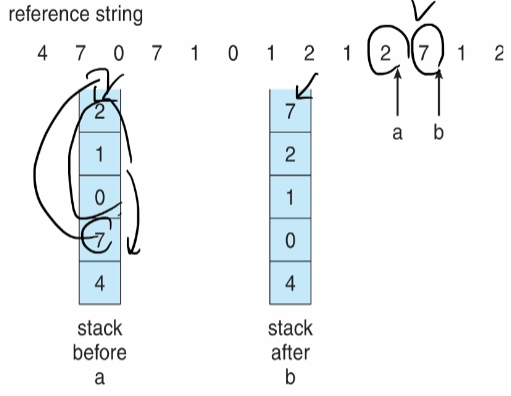
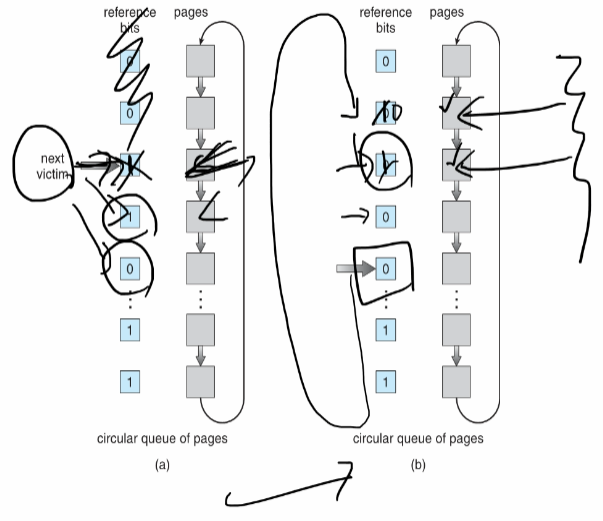
Chapter 10: Virtual Memory; 가상 메모리

1. Background; 배경
   * 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 That is Larger Than Physical Memory; 물리 메모리보다 큰 가상 메모리를 보여주는 다이어그램  
     
2. Demand Paging; 요구 페이징
   * 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
   * Valid-Invalid Bit
     + With each page table entry a valid-invalid bit is associated (v => in-memory – memory resident, i => not-in-memory)
     + Initially valid-invalid bit is set to I on all entries
     + Example of a page table snapshot:  
       
     + During MMU address translation, if valid-invalid bit in page table entry is i => page fault
   * Page Table When Some Pages Are Not in Main Memory; 일부 페이지가 메인 메모리에 없을 때의 페이지 테이블  
     
   * Steps in Handling Page Fault; 페이징 폴트를 처리하는 과정
     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
     3. Find free frame
     4. Swap page into frame via scheduled disk operation
     5. Reset tables to indicate page now in memory  
        Set validation bit
     6. Restart the instruction that caused the page fault  
        
   * Aspects of Demand Paging
     1. Extreme case – start process with no pages in memory
        1. OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
        2. And for every other process pages on first access
        3. Pure demand paging  
           순수 요구 페이징
     2. Actually, a given instruction could access multiple pages -> multiple page faults
        1. Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
        2. Pain decreased because of locality of reference
   1. Free-Frame List; 가용 프레임 리스트
      * 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.  
        
   2. Performance of Demand Paging; 요구 페이징의 성능
      * Three major activities
        + Service the interrupt – careful coding means just several hundred instructions needed
        + Read the page – lots of time
        + Restart the process – again just a small amount of time
      * Page Fault Rate 0 <= p <= 1
        + if p = 0 no page faults
        + if p = 1, every reference is a fault
      * Effective Access Time (EAT)  
        EAT = (1 – p x memory access  
        + p (page fault over head  
        + swap page out  
        + swap page in)  
        
3. Page Replacement; 페이지 교체
   * Need for Page Replacement; 페이지 교체의 필요성  
     
   * Page Replacement; 페이지 교체  
     
   * Page and Frame Replacement Algorithm
     + Frame-allocation algorithm determines
       - How many frames to give each process
       - Which frames to replace
     + Page-replacement algorithm
       - Want lowest page-fault rate on both first access and re-access
     + Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
       - String is just page numbers, not full addresses
       - Repeated access to the same page does not cause a page fault
       - Results depend on number of frames available
     + In all our examples, the reference string of referenced page numbers is  
       7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
   * Graph of Page Faults Versus The Number of Frames; 페이지 폴트 횟수 대 프레임 개수의 그래프  
     
   1. FIFO Page Replacement; FIFO 페이지 교체
      * Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
      * 3 frames (3 pages can be in memory at a time per process)  
          
        15 page faults
      * Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
        + Adding more frames can cause more page faults!
          1. Belady’s Anomaly
      * How to track ages of pages?
        + Just use a FIFO queue
      * FIFO Illustrating Belady’s Anomaly(모순)  
        
        + FIFO를 쓰면 일반적이지 않은 상황이 발생할 수 있음
   2. Optimal Page Replacement; 최적 페이지 교체
      * Replace page that will not be used for longest period of time
        + 9 is optimal for the example
      * How do you know this?
        + Can’t read the future
      * Used for measuring how well your algorithm performs  
        
   3. Least Recently Used (LRU) Algorithm; LRU 페이지 교체
      * Use past knowledge rather than future
      * Replace page that has not been used in the most amount of time
      * Associate time of last use with each page  
        
      * 12 faults – better than FIFO but worse than OPT
      * Generally good algorithm and frequently used
      * But how to implement?
      * Counter implementation (P.448)
        + Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
        + When a page needs to be changed, look at the counters to find smallest value
          1. Search through table needed
      * Stack implementation
        + Keep a stack of page numbers in a double link form:
        + Page referenced:
          1. move it to the top
          2. requires 6 pointers to be changed
        + But each update more expensive
        + No search for replacement
      * LRU and OPT are cases of stack algorithms that don’t have Belady’s Anomaly
      * Use Of A Stack to Record Most Recent Page References; 가장 최근의 페이지 참조를 기록하기 위한 스택의 사용  
        
   4. LRU Approximation Page Replacement; LRU 근사 페이지 교체
      * LRU needs special hardware and still slow
      * Reference bit
        + With each page associate a bit, initially = 0
        + When page is referenced bit set to 1
        + Replace any with reference bit = 0 (if one exists)
          1. We do not know the order, however
      * Second-chance algorithm
        + Generally FIFO, plus hardware-provided reference bit
        + Clock replacement
        + If page to be replaced has
          1. Reference bit = 0 -> replace it
          2. reference bit = 1 then:

set reference bit 0, leave page in memory

replace next page, subject to same rules

* + - Second-Chance (clock) Page-Replacement Algorithm; 이차 기회(클록) 페이지 교체 알고리즘  
      
  1. Counting Algorithms
     + Keep a counter of the number of references that have been made to each page
       - Not common
     + Lease Frequently Used (LFU) Algorithm: replaces page with smallest count
     + Most Frequently Used (MFU) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used