Chapter 9 summary

- Code needs to be in memory to execute, but entire program rarely used.
- 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.
    - Less I/O needed to load or swap programs into memory → each user program runs faster.
- **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 address space** – logical view of how process is stored in 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.
    - Like **paging system** with swapping.
    - Page is needed – reference to it:
        - invalid reference (accessing page that is not related to the current program) → 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.
    - Pages that are always needed are **memory resident**.
    - If page needed and not memory resident:
        - Need to detect and load the page into memory from storage.
- **Valid-Invalid Bit**
    - **v** → in-memory – **memory resident**
    - **i** → not-in-memory

  - During MMU address translation, if valid–invalid bit in page table entry is i → **page fault**.
- **Page Fault (steps in handling page fault):**
  - Operating system looks at the process's internal table to decide:
    - Invalid reference → abort
    - Just not in memory
  - Find free frame.
  - Swap page into frame via scheduled disk operation.
  - Reset tables to indicate page now in memory (Set validation bit = v).
  - Restart the instruction that caused the page fault.
- **Aspects of demand paging:**
  - **Pure demand paging** – start process with no pages in memory.
  - **Locality of reference** – Some localities that their references are known and are used continuously.
  - Hardware support needed for demand paging:
    - Page table with <u>valid / invalid</u> bit.
    - <u>Secondary memory</u> (swap device with swap space).
    - Ability to <u>restart</u> any instruction after a page fault.
- **Instruction restart:**
  - Must be able to restart the instruction in **exactly** in the same <u>place</u> and <u>state</u>.
  - Consider an instruction that could access several different locations.
    - Take care if there is an overlap!
- **Performance of demand paging:**
  - 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
  - Effective Access Time (EAT):
    - EAT = (1 – p) x memory access + p (page fault overhead + swap page out + swap page in)
    - Page fault ↑ memory access time ↑ performance ↓
- **Demand paging optimization:**
  - Swap space I/O faster than file system I/O even if on the same device.
  - Demand pages for program binary files – they are never modified.
- **Copy-on-write:**
  - Allow both <u>parent</u> and <u>child</u> processes to initially **share** the same pages in memory.

        - if either process writes to a shared page, a copy of the shared page is created.
    - In general, free frames are allocated from a pool of zero-fill-on-demand pages.
- **Page replacement** – find some page in memory, but not really in use, page it out.
- Same page may be brought into memory several times.
- Page replacement:
    - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
    - Use modify (dirty) bit to reduce overhead of page transfers.
        - Only modified pages are written to disk.
    - Page replacement completes separation between logical memory and physical memory.
- Basic page replacement:
    - Find the location of the desired page on disk.
    - 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 victim frame to disk if dirty.
    - Bring the desired page into the (newly) free frame; update the page and frame tables.
    - Continue the process by restarting the instruction that caused the trap.

    (Note: if no frames are free, two-page transfers are required for page fault – increasing EAT (Effective Access Time))

- Frame-allocation algorithm determines how many frames to give each process.
    - Frames 🡅 page fault 🡇
- Evaluate each 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.
- **First-In-First-Out (FIFO) Algorithm:** **Important example in slide 34**
    - Belady's Anomaly: Adding more frames can cause more page faults!
- **Optimal algorithm:** **Important example in slide 37**
    - Replace page that will not be used for longest period of time.
    - Used for measuring how well your algorithm performs.
- **Least Recently Used (LRU) Algorithm:** **Important example in slide 38**
    - Use past knowledge rather than future.
    - Replace page that has not been used in the most amount of time.

- Better than FIFO but worse than OPT.
- Counter implementation:
  - Every page-table entry has a time-of-use field; whenever a reference to a page is made, the contents of the clock are copied to this field its entry.
- Stack implementation:
  - Keep a stack of page numbers, whenever a page is referenced, it is removed and put on top of the stack.
  - Least recently used page is at bottom of the stack.

- **LRU Approximation Algorithms:**
  - 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)
      - However, we do not know the order.
  - Second-chance algorithm:
    - Generally, FIFO plus hardware-provided reference bit
    - Clock replacement
    - If page to be replaced has:
      - Reference bit = 0 => replace it
      - Reference bit = 1 then:
        - set reference bit 0, leave page in memory.
        - replace next page, subject to same rules.
- **Enhanced Second-Chance Algorithm:**
  - Improve algorithm by using reference bit and modify bit (if available) in concert.
  - Take ordered pair (reference, modify)
    - (0, 0) neither recently used not modified – best page to replace.
    - (0, 1) not recently used but modified – not quite as good, must write 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 need to write out before replacement.
- **Counting Algorithms:**
  - Keep a counter of the number of references that have been made to each page.
  - Lease Frequently Used (LFU) Algorithm: replaces the page with smallest count
    - A problem when a page is used heavily during initial phase of a process but then is never used again.
  - Most Frequently Used (MFU) Algorithm: replaces the page with largest count
- **Page-Buffering Algorithms:**
  - Keep a pool of free frames, always and keep a list of modified pages.

- **Allocation of frames:**
    - Each process needs minimum number of frames.
    - Maximum of course is total frames in the system.
    - Fixed Allocation:
        - Fixed equal allocation:
            - Equal number of frames to each process.
        - Fixed proportional allocation:
            - Allocate according to the size of process.
            - Dynamic as degree of multiprogramming, process sizes change.
    - Priority Allocation:
        - Use a proportional scheme wherein ratio of frames depends on priorities of processes rather than on relative sizes of processes or on a combination of size and priority.
- **Global vs. Local Allocation:**
    - Global replacement – a process selects a replacement frame from the set of all frames; one process can take a frame from another.
    - Local replacement – each process selects from only its own set of allocated frames.
        - More consistent per-process performance.
        - But possibly underutilized memory.
- **Thrashing:**
    - If a process does not have "enough" pages, the page-fault rate is very high.
        - Page fault to get page.
        - Replace existing frame.
        - But quickly need replaced frame back.
        - This leads to:
            - Low CPU utilization.
            - Operating system thinking that it needs to increase the degree of multiprogramming.
            - Another process added to the system.
    - Thrashing → a process is busy swapping pages in and out.