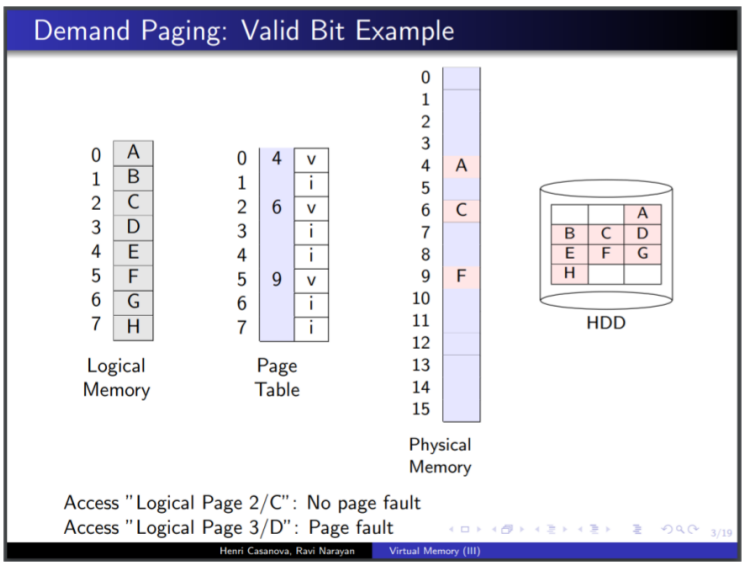
Demand Paging

- way in which OS allocates pages to process
- "Don't load a page before the process references it"
  - Initially just load one page, the one with the first instruction of the program
  - Each time the program issues an address, load the corresponding page if not already loaded
- a lazy scheme as it opposes the eager scheme of loading all page at once
- For each page, the OS keeps track of whether it is in RAM or not
- done using valid bit of page table entries
  - A page is marked as valid if it is legal and in memory
  - A page is marked as invalid if it is illegal or on disk
  - Initially all pages are marked invalid
- During address translation, if the bit is invalid, a trap is generated: a page fault



Page Faults

- When the CPU issues an address, first one determines whether it's legal or not
  - i.e., does it correspond to a page number that's not beyond the number of pages allowed for a process
  - If it is illegal, then the process is aborted with some message
- Lookup the valid bit in the page table entry
- If the valid bit is set, do the address translation as usual

If not

- Find a free frame (from the list of free frames in the kernel)
- Schedule the disk access to load the page into the frame
- Kick the process off the CPU and put it the blocked/waiting state
- Once the disk access is complete, update the process' page table with the new logical/physical memory mapping
- Update the valid bit
- Set the process state to Ready (it should then run soon)
- That process will rerun the instruction that caused the page fault

Rerun the "offending" instruction

- After a pagefault is resolved that OS has loaded the required page in RAM, and one simply rerun the instruction from scratch
  i.e., restart the fetch, decode, execute cycle at that instruction
- This is only possible because our instructions don't modify more than one memory location
  Which avoids a difficult "the instruction did half its work in RAM, but then page faulted, so when you restart it be careful that the first half of the work was already done" situation
- In other terms, load/store ISAs are perfectly designed for page faults

Virtual Memory Performance

What are the limits of this on-demand mechanism? Is it worth using?

- t_m memory access time: 10ns to 200 ns; typically: 70 ns
  i.e., the time to access a byte in memory
- t_p page fault time Typically: 5-50 ms (SSD: 3-10 times faster)
  i.e., the time required to load the page from the disk, place it in memory, and rerun the instruction
- How much faster is the memory compared to the disk?
  Assume that $t_m = 10^{-8}$ s and $t_p = 10^{-2}$ s

$$\frac{t_p}{t_m} = \frac{10^{-2}}{10^{-8}} = 10^6$$

  $\rightarrow$ The memory is 1 million time faster than the disk

Effective Access Time

- total memory access time (T)

$$T = n_0 \times t_m + n_p \times t_p$$

  - n_0 times there is no page faults
  - n_p times there is a page fault
  - n = $n_p + n_0$
- average access time

$$t = \frac{n_0 \times t_m + n_p \times t_p}{n}$$

  =

$$(1 - \frac{n_p}{n}) \times t_m + \frac{n_p}{n} \times t_p$$

- page fault rate (page fault probability)

$$p = \frac{n_p}{n}$$

  s.t $0 \leq p \leq 1$
- average access time

$$t = (1 - p)t_m + pt_p$$

Conclusion The Page fault rate must be kept as small as possible.
What can be done?

- increase the memory size
- limit the size of process address space
- tell programmers to develop program with small address space

Fork() Exec()

- fork() makes a copy of the parent process address space to create an identical child process
- most of the time exec() is used in the child to run another program
- Why is making a copy of the parent's address space wasteful?
  The child address space is immediately overwritten with another

Copy-on-Write

- during fork(), don't copy the address space and initially share all pages
  - Save for some heap and stack pages, that are necessary for any new process
- Whenever the parent or the child modifies a page, then copy it
- This "lazy" scheme is used in all OSes

Page Replacement

- Virtual Memory increases multi-programming and provides the illusion of large address spaces
- **What if we run out of memory?**
  a page fault occurs and free frame list is empty
  $\therefore$ need to kick a page out of RAM
- page replacement
  Evict a victim page from a frame & Put the newly needed page into that frame
  may require two page transfers

Dirty Bit

- No need to write a victim back to disk if that victim has never been modified
- With this reason each page table entry has a dirty bit
  dirty bit is initially set to 0
  - whenever the process writes to the page the dirty bit is set to 1
  - if a page is evicted its written to disk only if its dirty
- Most OSes do opportunistic un-dirtying: If the disk is idle pick a dirty page, write it out and clear its dirty bit

Lesson Summary

- mechanism
  - bring pages in from disk on demand (when page fault)
  - write pages to disk when needed (RAM is full)
  - dirty bit is used to avoid doing redundant writes to disk
- we need policies and the question is
  Which pages do we kick back to disk? and How many frames do we let a process have?