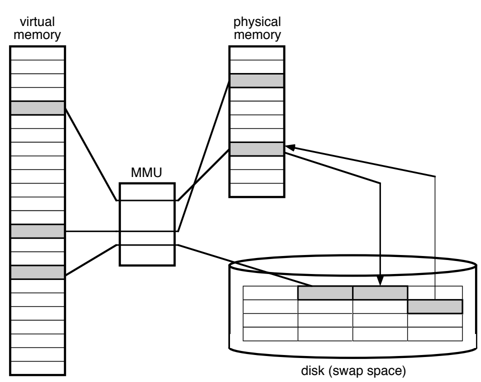
Topics in Memory Management

- Uniprogrammed operating systems
  - Assembling, linking, loading
  - Static memory allocation
  - Dynamic memory allocation
    - Stacks, heaps
- Multiprogrammed operating systems
  - Includes most of the above topics
  - Static relocation
  - Dynamic relocation
    - Logical vs. physical address
    - Partitioning
    - Segmentation
    - Paging
  - Swapping
  - Virtual memory / demand paging

Memory Management So Far

- An application's view of memory is its logical address space
- The OS's view of memory is the physical address space
- A MMU (hardware) is used to implement segmentation, paging, or a combination of the two, by translating addresses for the CPU
- Limitation until now — ***all*** segments / pages of a process must be in main (physical) memory for it to run
- Insight — at a given time, we probably only need to access some small subset of process's logical memory
  - Load pages / segments on demand

Demand Paging (Virtual Memory)



- At a given time, a virtual memory page will be stored either:
  - In a frame in physical memory
  - On disk (*backing store*, or *swap space*)
- A process can run with only part of its virtual address space in main memory
  - Provide illusion of almost infinite memory

Loading a New Process

- Processes are started with 0 or more of their virtual pages in physical memory, and the rest on the disk
- *Page selection* — ***when*** are new pages brought into physical memory?
  - Prepaging — pre-load enough to get started: code, static data, one stack page (DEC ULTRIX)
  - Demand paging — start with 0 pages, load each page on demand (when a page fault occurs) (most common approach)
    - Disadvantage: many (slow) page faults when program starts running
- Demand paging works due to the principle of *locality of reference*
  - Knuth estimated that 90% of a program's time is spent in 10% of the code

Page Faults

- An attempts to access a page that's not in physical memory causes a *page fault*
  - Page table must include a *present* bit (sometimes called *valid* bit) for each page
  - An attempt to access a page without the present bit set results in a *page fault*, an *exception* which causes a *trap* to the OS
  - When a page fault occurs:
    - OS must *page in* the page — bring it from disk into a free frame in physical memory
    - OS must update page table & present bit
    - Faulting process continues execution
- Unlike interrupts, a page fault can occur any time there's a memory reference
  - Even in the middle of an instruction! (how? and why not with interrupts??)
  - However, handling the page fault must be invisible to the process that caused it

Handling Page Faults

- The page fault handler must be able to recover enough of the machine state (at the time of the fault) to continue executing the program
- The PC is usually incremented at the beginning of the instruction cycle
  - If OS / hardware doesn't do anything special, faulting process will execute the next instruction (skipping faulting one)
- With hardware support:
  - Test for faults before executing instruction (IBM 370)
  - Instruction completion — continue where you left off (Intel 386…)
  - Restart instruction, undoing (if necessary) whatever the instruction has already done (PDP-11, MIPS R3000, DEC Alpha, most modern architectures)

Performance of Demand Paging

- Effective access time for demand-paged memory can be computed as:

  eacc = (1–p)(macc) + (p)(pfault)

  where:

  p = probability that page fault will occur

  macc = memory access time

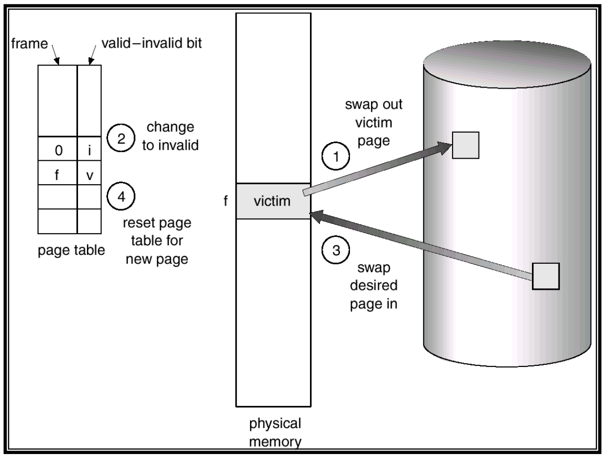
  pfault = time needed to service page fault
- With typical numbers:

  eacc = (1–p)(100) + (p)(25,000,000)
  = 100 + (p)(24,999,900)
  - If p is 1 in 1000, eacc = 25,099.9 ns (250 times slower!)
  - To keep overhead under 10%, 110 > 100 + (p)(24,999,900)
    - p must be less than 0.0000004
    - Less than 1 in 2,5000,000 memory accesses must page fault!

Page Replacement

- When the OS needs a frame to allocate to a process, and all frames are busy, it must evict (copy to backing store) a page from its frame to make room in memory
  - Reduce overhead by having CPU set a *modified / dirty* bit to indicate that a page has been modified
    - Only copy data back to disk for dirty pages
    - For non-dirty pages, just update the page table to refer to copy on disk
- Which page to we choose to replace? Some page replacement policies:
  - Random
    - Pick any page to evict
  - FIFO
    - Evict the page that has been in memory the longest (use a queue to keep track)
    - Idea is to give all pages "fair" (equal) use of memory

Page Replacement



Page Replacement Policy

- When OS needs a frame to use, and all are busy, which page does it evict?
  - Random
    - Pick any page to evict
  - FIFO
    - Evict the page that has been in memory the longest (use a queue to keep track)
  - Optimal (Minimal)
    - Evict the page that will be referenced the farthest into the future
      - Requires knowledge of future
    - Cannot really be implemented
      - Useful for evaluating other policies
  - Least-Recently-Used (LRU)
    - Use the past to predict the future
    - Evict the page that has been unreferenced for the longest period of time

Page Reference Example

- Assumptions: 4 pages, 3 frames
  Page references: ABCABDADBCB

FIFO	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Optimal	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

LRU	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Implementing LRU

- A perfect implementation would be something like this:
  - Associate a clock register with every page in physical memory
  - Update the clock value at every access
  - During replacement, scan through all the pages and find the one with the lowest value in its clock register
  - What's wrong with all this?
- Simple approximations:
  - FIFO
  - Not-recently-used (NRU)
    - Use an R (reference) bit, and set it whenever a page is referenced
    - Clear the R bit periodically, such as every clock interrupt
    - Choose any page with a clear R bit to evict

Implementing LRU (cont.)

- Clock / Second Chance Algorithm
  - Use an R (reference) bit as before
  - On a page fault, circle around the "clock" of all pages in the user memory pool
    - Start after the page examined last time
    - If the R bit for the page is set, clear it
    - If the R bit for the page is clear, replace that page and set the bit
  - Questions:
    - Can it loop forever?
    - What does it mean if the "hand" is moving slowly? …if the hand is moving quickly?
- Least Frequently Used (LFU) / N-th Chance Algorithm
  - Don't evict a page until hand has swept by N times
  - Use an R bit and a counter
  - How is N chosen? Large or small?

Frame Allocation

- How many frames does each process get (M frames, N processes)?
  - At least 2 frames (one for instruction, one for memory operand), maybe more…
  - Maximum is number in physical memory
- Allocation algorithms:
  - Equal allocation
    - Each gets M / N frames
  - Proportional allocation
    - Number depends on size and priority
- Which pool of frames is used for replacement?
  - Local replacement
    - Process can only reuse its own frames
  - Global replacement
    - Process can reuse any frame (even if used by another process)

Thrashing

- Consider what happens when memory gets overcommitted:
  - After each process runs, before it gets a chance to run again, all of its pages may get paged out
  - The next time that process runs, the OS will spend a ***lot*** of time page faulting, and bringing the pages back in
    - All the time it's spending on paging is time that it's not getting useful work done
    - With demand paging, we wanted a very large virtual memory that would be as fast as physical memory, but instead we're getting one that's as slow as the disk!
- This wasted activity due to frequent paging is called *thrashing*
  - Analogy — student taking too many courses, with too much work due

Working Sets

- Thrashing occurs when the sum of all processes' requirement is greater than physical memory
    - Solution — use local page frame replacement, don't let processes compete
      - Doesn't help, as an individual process can still thrash
    - Solution — only give a process the number of frames that it "needs"
      - Change number of frames allocated to each process over time
      - If total need is too high, pick a process and suspend it
- *Working set* (Denning, 1968) — the collection of pages that a process is working with, and which must be resident in main memory, to avoid thrashing
  - Always keep working set in memory
  - Other pages can be discarded as necessary