CS 3204
Operating Systems

Lecture 19

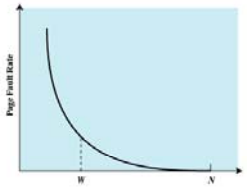
Godmar Back

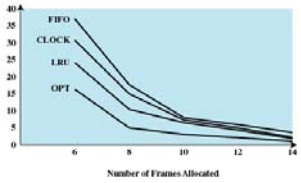
Announcements

- Project 3 due Nov 11, 11:59pm

VM Design Issues & Techniques

of Page Faults vs Frame Allocation





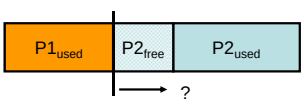
- Desired behavior of paging algorithm: reduce page fault rate below "acceptable level" as number of available frames increases
- Q.: does increasing number of physical frames always reduce page fault rate?
  - A.: usually yes, but for some algorithms not guaranteed ("Belady's anomaly")

Page Buffering

- Select victim (as dictated by page replacement algorithm – works as an add-on to any algorithm we discussed)
- But don't evict victim – put victim on tail of victim queue. Evict head of that queue instead.
- If victim page is touched before it moves to head of victim queue, simply reuse frame
- Further improvement: keep queue of unmodified victims (for quick eviction – aka *free page list*) and separate queue of modified pages (aka *modified list* - allows write-back in batch)
- Related issue: when should you write modified pages to disk?
  - Options: demand cleaning vs pre-cleaning (or pre-flushing)

Local Replacement

- So far, considered global replacement policies
  - Most widely used
- But could also divide memory in pools
  - Per-process or per-user
- On frame allocation, requesting process will evict pages from pool to which it belongs
- Advantage: Isolation
  - No between-process interference
- Disadvantage: Isolation
  - Can't temporarily "borrow" frames from other pools
- Q.: How big should pools be?
  - And when should allocations change?



When Virtual Memory works well

- Locality
  - 80% of accesses are to 20% of pages
  - 80% of accesses are made by 20% of code
- Temporal locality:
  - Page that's accessed will be accessed again in near future
- Spatial locality:
  - Prefetching pays off: if a page is accessed, neighboring page will be accessed
- If VM works well, average access to all memory is about as fast as access to physical memory

VM Access Time & Page Fault Rate

access time = p * memory access time
+ (1-p) * (page fault service time + memory access time)

- Consider expected access time in terms of fraction p of page accesses that don't cause page faults.
- Then 1-p is page fault frequency
- Assume p = 0.99, assume memory is 100ns fast, and page fault servicing takes 10ms – how much slower is your VM system compared to physical memory?
- access time = 99ns + 0.01*(10000100) ns ≈ 100,000ns or 0.1ms
  - Compare to 100ns or 0.0001ms speed ≈ about 1000x slowdown
- Conclusion: even low page fault rates lead to huge slowdown

Thrashing: When Virtual Memory Does Not Work Well

- System accesses a page, evicts another page from its frame, and next access goes to just-evicted page which must be brought in
- Worst case a phenomenon called Thrashing
  - leads to constant swap-out/swap-in
  - 100% disk utilization, but no process makes progress
    - CPU most idle, memory mostly idle

When does Thrashing occur?

- Process does exhibit locality, but is simply too large
  - Here: (assumption of) locality hurts us
- Process doesn't exhibit locality
  - Does not reuse pages
- Processes individually fit & exhibit locally, but in total are too large for the system to accommodate all

What to do about Thrashing?

- Buy more memory
  - ultimately have to do that
  - increasing memory sizes ultimately reason why thrashing is nowadays less of a problem than in the past – still OS must have strategy to avoid worst case
- Ask user to kill process
- Let OS decide to kill processes that are thrashing
  - Linux has an option to do that (see next slide)
- In many cases, still: reboot only time-efficient option
  - But OS should have reasonable strategy to avoid it if it can

An aircraft company discovered that it was cheaper to fly its planes with less fuel on board. The planes would be lighter and use less fuel and money was saved. On rare occasions however the amount of fuel was insufficient, and the plane would crash. This problem was solved by the engineers of the company by the development of a special OOF (out-of-fuel) mechanism. In emergency cases a passenger was selected and thrown out of the plane. (When necessary, the procedure was repeated.) A large body of theory was developed and many publications were devoted to the problem of properly selecting the victim to be ejected. Should the victim be chosen at random? Or should one choose the heaviest person? Or the oldest? Should passengers pay in order not to be ejected, so that the victim would be the poorest on board? And if for example the heaviest person was chosen, should there be a special exception in case that was the pilot? Should first class passengers be exempted? Now that the OOF mechanism existed, it would be activated every now and then, and eject passengers even when there was no fuel shortage. The engineers are still studying precisely how this malfunction is caused.

Source: lkml (Andries Brouwer), 2004

OS Strategies to prevent thrashing

- Or contain its effects
- Define: “working set” (1968, Denning)
- Set of pages that a process accessed during some window/period of length T in the past
  - Hope that it’ll match the set accessed in the future
- Idea: if we can manage to keep working set in physical memory, thrashing will not occur

Working Set

- Suppose we know or can estimate working set – how could we use it?
- Idea 1: give each process as much memory as determined by size of its WS
- Idea 2: preferably evict frames that hold pages that don’t seem to be part of WS
- Idea 3: if WS cannot be allocated, swap out entire process (and exclude from scheduling for a while)
  - “medium term scheduling”, “swap-out scheduling”
  - (Suspended) inactive vs active processes
  - Or don’t admit until there’s enough frames for their WS (“long term scheduling”)

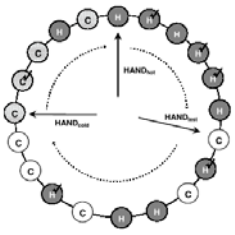
Estimating Working Set

- Compute “idle time” for each page
  - Amount of CPU time process received since last access to page
- On page fault, scan resident pages
  - If referenced, set idle time to 0
  - If not referenced, idle_time += time since last scan
  - If idle_time > T, consider to not be part of working set
- This is known as working set replacement algorithm [Denning 1968]
- Variation is WSClock [Carr 1981]
  - treats working set a circular list like global clock does, and updates “time of last use” (using a process’s CPU use as a measure) – evicting those where T_last < T_current - T

Page Fault Frequency

- Alternative method of working set estimation
  - PFF: # page faults/instructions executed
  - Pure CPU perspective vs memory perspective provided by WSClock
- Below threshold – can take frames away from process
- Above threshold – assign more frames
- Far above threshold – suspect thrashing & swap out
- Potential drawback: can be slow to adopt to periods of transition

Clock-PRO

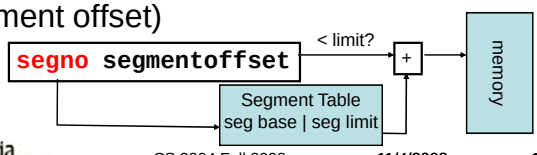


- Clock and algorithms like it try to approximate LRU:
  - LRU does not work well for:
  - Sequential scans, large loops
- Alternative:
  - Reuse distance: should replace page with large reuse distance
- Clock-PRO: Idea – extend our focus by remembering information about pages that were evicted from frames previously
- See [Jiang 2005]

Segmentation

Segmentation

- Historical alternative to paging
- Instead of dividing virtual address space in many small, equal-sized pages, divide into a few, large segments
- Virtual address is then (segment number, segment offset)



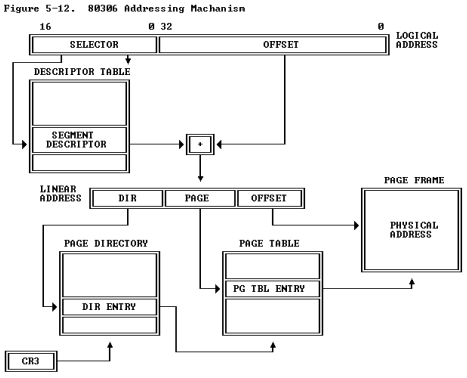
Segmentation (2)

- Advantages:
  - little internal fragmentation "segments can be sized just right"
  - easy sharing – can share entire code segment
  - easy protection – only have to set access privileges for segment
  - small number of segments means small segment table sizes
- Disadvantages:
  - external fragmentation (segments require physically continuous address ranges!)
  - if segment is partially idle, can't swap out

Segmentation (3)

- Pure segmentation is no longer used
  - (Most) RISC architectures don't support segmentation at all
  - Other architectures combine segmentation & paging
- Intel x86 started out with segmentation, then added paging
  - Segment number is carried in special set of registers (GS, ES, FS, SS), point to "selectors" kept in descriptor tables
  - Instruction opcode determines with segment is used
  - Today: segmentation unit is practically unused (in most 32-bit OS, including Pintos): all segments start at 0x00000000 and end at 0xFFFFFFFF (Exception: for thread-local data!)
  - Do not confuse with Pintos's code/data segments, which are linear subregions of virtual addresses spanning multiple virtual pages
- Note: superpages are somewhat of a return to segmentation

Combining Segmentation & Paging



Mem Mgmt Without Virtual Memory

- Problems that occur when VM is lacking motivate need for it, historically
  - But still important for VM-less devices (embedded devices, etc.)
- Imagine if we didn't have VM, it would be hard or impossible to
  - Retain the ability to load a program anywhere in memory
  - Accommodate programs that grow or shrink in size
  - Use idle memory for other programs quickly
  - Move/relocate a running program in memory
- VM *drastically* simplifies systems design