CS162
Operating Systems and Systems Programming
Lecture 15

Page Allocation and Replacement

March 19, 2008

Prof. Anthony D. Josep
http://inst.eecs.berkeley.edu/~cs162

Review: Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Review: Software-Loaded TLB

- MIPS/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without hardware TLB fill?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a TLB Fault
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Review: Implementing LRU

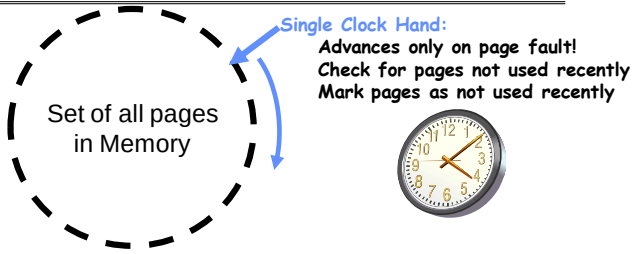
- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page
- Details:
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO

Goals for Today

- Page Replacement Policies
  - Clock Algorithm, Nth chance algorithm, 2nd-Chance-List Algorithm
- Page Allocation Policies
- Working Set/Thrashing
- Distributed Problems
  - Brief History
  - Parallel vs. Distributed Computing
  - Parallelization and Synchronization
  - Prelude to MapReduce

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne; and from slides licensed under the Creative Commons Attribution 2.5 License by the University of Washington (2007). Many slides generated from my lecture notes by Kubiatowicz.

Clock Algorithm: Not Recently Used



- What if hand moving slowly?
  - Good sign or bad sign?
    - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

Nth Chance version of Clock Algorithm

- Nth chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

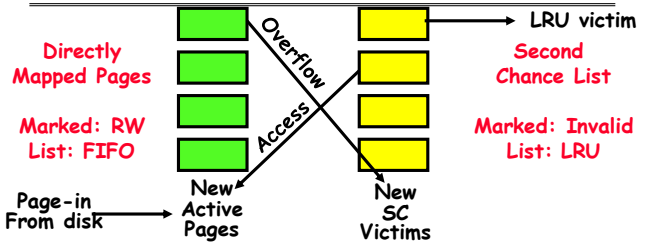
Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - » For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - » Initially, mark all pages as read-only, even data pages
    - » On write, trap to OS. OS sets software "modified" bit, and marks page as read-write.
    - » Whenever page comes back in from disk, mark read-only

Clock Algorithms Details (continued)

- Do we really need a hardware-supported "use" bit?
  - No. Can emulate it similar to above:
    - » Mark all pages as invalid, even if in memory
    - » On read to invalid page, trap to OS
    - » OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous:
    - » On write, trap to OS (either invalid or read-only)
    - » Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)



- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ LRU, but page fault on every page reference
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn't VAX include "use" bit?
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did OK anyway

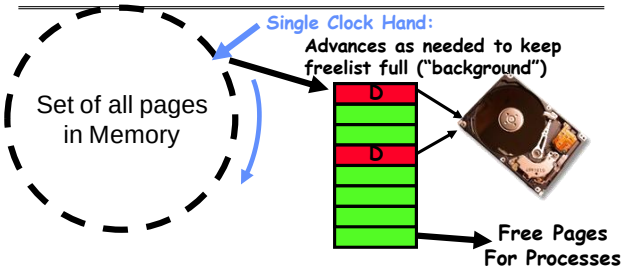
Administrivia

- All slides except last lecture are (finally) online!
  - Reader should be available Friday
- Project 2 code due Thursday 3/20 at 11:59pm

Aside: Powers of 10 and 2

- Strict powers of 10:
  - yotta: 10^{24}
  - exa: 10^{18}
  - peta: 10^{15}
  - tera: 10^{12}
  - giga: 10^{9}
  - mega: 10^{6}
  - kilo: 10^{3}
  - milli(m): 10^{-3}
  - micro (μ): 10^{-6}
  - nano(n): 10^{-9}
  - pico: 10^{-12}
  - femto: 10^{-15}
  - atto: 10^{-18}
  - yocto: 10^{-24}
- Strict powers of 2:
  - yotta: $2^{80} \cong 10^{24}$
  - exa: $2^{60} \cong 10^{18}$
  - peta: $2^{50} \cong 10^{15}$
  - tera: $2^{40} \cong 10^{12}$
  - giga: $2^{30} = 1{,}073{,}741{,}824 \cong 10^{9}$
  - mega: $2^{20} = 1{,}048{,}576 \cong 10^{6}$
  - kilo: $2^{10} = 1024 \cong 10^{3}$
- When to use one or the other?
  - Powers of 2
    - » Memory sizes
  - Powers of 10
    - » Time
    - » Bandwidth

Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit? Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - » Can't push page out to disk without invalidating all PTEs

Allocation of Page Frames (Memory Pages)

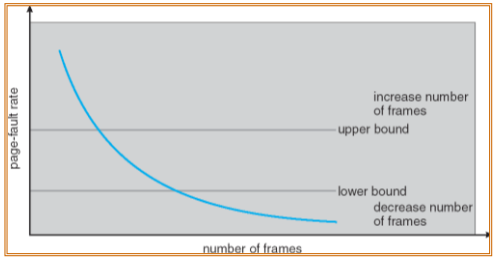
- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from 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

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes⇒process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:
    s_i = size of process p_i and $S = \Sigma s_i$
    m = total number of frames
    a_i = allocation for $p_i = \frac{s_i}{S} \times m$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

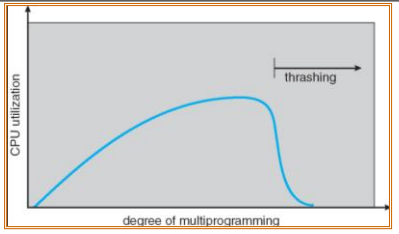
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

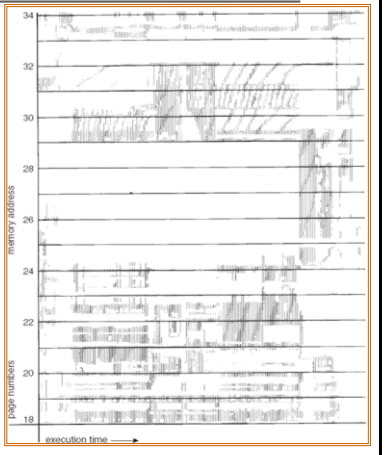
Thrashing



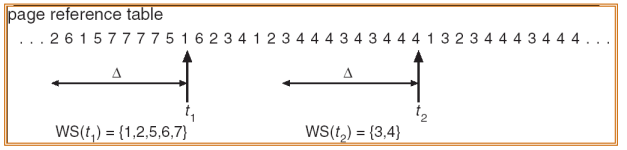
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model



- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma |WS_i|$ ≡ total demand frames
- if $D > m$ ⇒ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

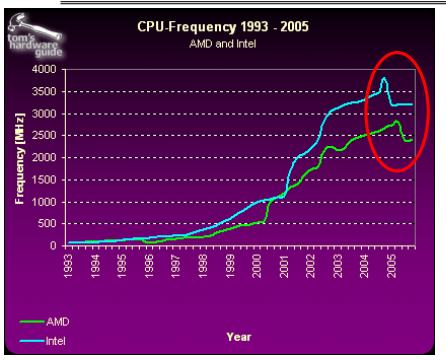
- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages "around" the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

BREAK

Computer Speedup



Moore's Law: "*The density of transistors on a chip doubles every 18 months, for the same cost*" (1965)

- What can you do with 1 computer?
- What can you do with 100 computers?
- What can you do with an entire data center?

Image: Tom's Hardware

Some Distributed Problems

- Rendering multiple frames of high-quality animation

- Simulating several hundred or thousand characters

Happy Feet © Kingdom Feature Productions; *Lord of the Rings* © New Line Cinema; *Shrek* © DreamWorks Animation

More Distributed problems

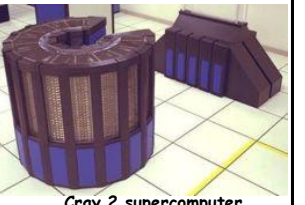
- Indexing the web (Google)
- Simulating an Internet-sized network for networking experiments (PlanetLab, DETER)
- Speeding up content delivery (Akamai)

- *What is the key attribute that all these examples have in common?*

- Distributed computing:
  - Multiple CPUs/cores across many computers (MIMD)
- Parallel computing:
  - Vector processing of data (SIMD)
  - Multiple CPUs/cores in a single computer (MIMD)

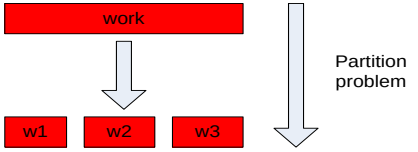
A Brief History… 1975-85

- 1975-85:
  - Primarily vector-based parallel computing in early years
  - Gradually more thread-based parallelism introduced
- 1985-95:
  - "Massively parallel architectures" rise in prominence
  - Message Passing Interface (MPI) and other libs developed
  - Bandwidth was a big problem
- 1995-today:
  - Berkeley Network of Workstations
    - » COTS tech instead of special node machines
  - Cluster/grid architecture increasing
  - Web-wide cluster software
    - » Microsoft, Google, Amazon take this to the extreme (thousands of nodes/cluster)


Cray 2 supercomputer (Wikipedia)

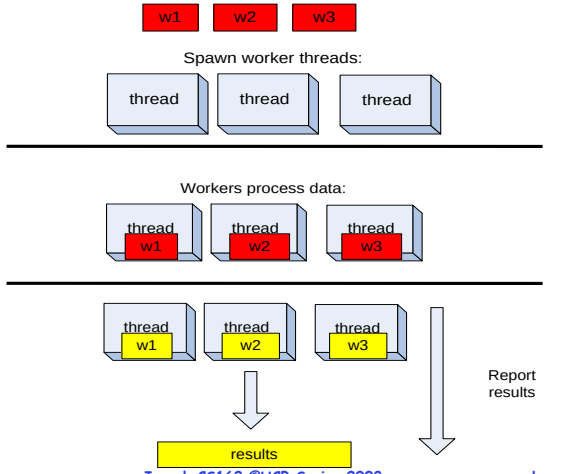
Parallelization Idea

- Parallelization is "easy" if processing can be cleanly split into n units:



- In a parallel computation, we would like to have as many threads as we have processors
  - Ex. 4-CPU computer would be able to run four threads at the same time

Parallelization Idea (2)



Parallelization Pitfalls

- This model is too simple!
  - How do we assign work units to worker threads?
  - What if we have more work units than threads?
  - How do we aggregate the results at the end?
  - How do we know all the workers have finished?
  - What if the work cannot be divided into completely separate tasks?
- Multiple threads must communicate with one another, or access a shared resource
  - We need a *synchronization system!*

Prelude to MapReduce

- Explicit parallelism/synchronization is really hard!!
  - Must consider all possible shared state, keep locks organized, and use them consistently and correctly
- Knowing there are bugs may be tricky; fixing them can be even worse!
- Solution: *Minimize shared state to reduce total system complexity*
- But, synchronization doesn't address distributed computing questions (e.g., moving data around)
- Fortunately, MapReduce handles this for us
  - Google-designed paradigm for making large subset of distributed problems easier to code
  - Automates data distribution & result aggregation
  - Restricts the ways data can interact to eliminate locks (no shared state = no locks!)