Page Allocation and Replacement

March 11, 2010

Ion Stoica

http://inst.eecs.berkeley.edu/~cs162

Review: Paging & Address Translation

Physical Address:
Offset
Physical Page #

Virtual Address:
Offset
Virtual P2 index
Virtual P1 index
PageTablePtr

Page Table
(1st level)

Page Table
(2nd level)

Physical Memory:

Page Table
(2nd level)

PageTablePtr

Page Table
(1st level)

Physical Address:
Offset
Physical Page #

Virtual Address:
Offset
Virtual P2 index
Virtual P1 index
PageTablePtr

Page Table
(1st level)

Page Table
(2nd level)

Physical Memory:

Page Table
(2nd level)

PageTablePtr

Page Table
(1st level)

Physical Address:
Offset
Physical Page #

Virtual Address:
Offset
Virtual P2 index
Virtual P1 index
PageTablePtr

Page Table
(1st level)

Page Table
(2nd level)

Physical Memory:

Page Table
(2nd level)

PageTablePtr

Page Table
(1st level)

Physical Address:
Offset
Physical Page #

Virtual Address:
Offset
Virtual P2 index
Virtual P1 index
PageTablePtr

Page Table
(1st level)

Page Table
(2nd level)

Physical Memory:

Page Table
(2nd level)

PageTablePtr

Page Table
(1st level)
**Review: Demand Paging Mechanisms**

- **PTE helps us implement demand paging**
  - Valid $\Rightarrow$ Page in memory, PTE points at physical page
  - Not Valid $\Rightarrow$ 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

**Goals for Today**

- Precise Exceptions
- Page Replacement Policies
  - Clock Algorithm
  - Nth chance algorithm
  - Second-Chance-List Algorithm
- Page Allocation Policies
- Working Set/Thrashing

**Consider weird things that can happen**

- What if an instruction has side-effects?
  - Options:
    - Unwind side-effects (easy to restart)
    - Finish off side-effects (messy!)
  - Example 1: mov (sp)+,10
    - What if page fault occurs when write to stack pointer?
    - Did sp get incremented before or after the page fault?
  - Example 2: strcpy (r1), (r2)
    - Source and destination overlap: can’t unwind in principle!
    - IBM S/370 and VAX solution: execute twice – once read-only
- What about “RISC” processors?
  - For instance delayed branches?
    - Example: bne elsewhere
      - ld r1, (sp)
    - Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    - Example: div r1, r2, r3
      - ld r1, (sp)
    - What if takes many cycles to discover divide by zero, but load has already caused page fault?
Precise Exceptions

- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position

- Imprecise ⇒ system software has to figure out what is where and put it all back together

- Performance goals often lead designers to forsake precise interrupts
  - System software developers, user, markets etc. usually wish they had not done this

Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Load<br>
4. Load page table<br>
5. Load page table<br>
6. Load frame<br>
7. Perform in missing page

What Factors Lead to Misses?

- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
      - Need to predict future somehow! More later.

- Capacity Misses:
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!

- Conflict Misses:
  - Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache

- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Demand Paging Example

- Since Demand Paging like caching, can compute average access time (“Effective Access Time”)
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:
    - EAT = (1 − p) x 200ns + p x 8ms
    - = (1 − p) x 200ns + p x 8,000,000ns
    - = 200ns + p x 7,999,800ns

- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!

- What if want slowdown by less than 10%?
  - 200ns x 1.1 < EAT ⇒ p < 2.5 x 10^-6
  - This is about 1 page fault in 400,000!
Administrivia

- Midterm exam statistics:
  - Mean score: 80.4
  - Median score: 82.5
  - Standard Dev.: 11.1
  - Max: 101
- By tomorrow
  - Solutions will be up
  - Grades in glookup
- Regrading policy: you have one week (until 3/18)
  - One page explaining
    » Which problem you want to be regraded and why (look at solution first)
  - We will regrade your entire exam, so there is a chance your grade may decrease!
- All in all: Excellent grades! Congratulations!

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    » The cost of being wrong is high: must go to disk
    » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair - let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future...
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable - makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!
  - Head
    Page 6 → Page 7 → Page 1 → Page 2
  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list...
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away
Suppose we have the same reference stream:
- A B C A B D A B C B

Consider MIN Page replacement:

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

Example: MIN

When will LRU perform badly?

Consider the following: A B C D A B C D A B C D

LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- Every reference is a page fault!

MIN Does much better:

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

Graph of Page Faults Versus The Number of Frames

- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: BeLady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called BeLady's anomaly)

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page
Implementing LRU & Second Chance

- **Perfect:**
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons

- **Second Chance Algorithm:**
  - Approximate LRU
    - Replace an old page, not the oldest page
  - FIFO with “use” bit

Details
- A “use” bit per physical page
- On page fault check page at head of queue
  - If use bit=1 ➔ clear bit, and move page at tail (give the page second chance)
  - If use bit=0 ➔ replace page
- Moving pages to tail still complex

Clock Algorithm

- **Clock Algorithm:** more efficient implementation of second chance algorithm
- Arrange physical pages in circle with single clock hand

Details:
- 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

- 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

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives

First loaded page

B u:0

Last loaded page

A u:1

D u:0

C u:0

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives

First loaded page

B u:0

Last loaded page

A u:1

D u:0

C u:0

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives

first loaded page

last loaded page

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives

first loaded page

last loaded page
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
Clock Replacement Illustration

- Max page table size 4

- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives

Clock Replacement Illustration

- Max page table size 4

- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives

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=clock 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 using “invalid” bit:
    » 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
  - When clock hand passes by, reset use bit 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

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
Summary

• Precise Exception specifies a single instruction for which:
  – All previous instructions have completed (committed state)
  – No following instructions nor actual instruction have started

• 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

• Nth-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