CS162
Operating Systems and Systems Programming
Lecture 15
Page Allocation and Replacement

October 25, 2010
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

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

Goals for Today

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

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

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.
Transparent Exceptions

- How to transparently restart faulting instructions?
  - Could we just skip it?
    » No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    » Need to know which instruction caused fault
    » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    » Save/restore registers, stack, etc
- What if an instruction has side-effects?

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 somewhere
      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 \(\Rightarrow\) 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
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise \(\Rightarrow\) 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
Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - \( \text{EAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \)
  - \( \text{EAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \)

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose \( p = \) Probability of miss, \( 1-p = \) Probability of hit
  - Then, we can compute EAT as follows:
    - \( \text{EAT} = 200\text{ns} + p \times 8 \text{ms} \)
    - \( = 200\text{ns} + p \times 8,000,000\text{ns} \)
  - If one access out of 1,000 causes a page fault, then \( EAT = 8.2 \mu s \):
    - This is a slowdown by a factor of 40!
  - What if want slowdown by less than 10%?
    - \( 200\text{ns} \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6} \)
    - This is about 1 page fault in 400000!

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

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!

  - 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)
**Administrivia**

- **Still Grading Midterms**
  - Hope to hand them out in section on Friday
  - Solutions have been posted
    - Just go to handouts page
- **Would you like an extra 5% for your course grade?**
  - Attend lectures and sections! 5% of grade is participation
- **Second Midterm?**
  - Would have to be either Monday or Wednesday after Thanksgiving (11/29 or 12/01).
  - I am currently seeing if there is a room available.
- **We have an anonymous feedback link on the course homepage**
  - Please use to give feedback on course
  - Soon: We will have a survey to fill out
- **Project II**
  - Due by Tomorrow at midnight
  - Good Luck!

---

**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>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

---

**Example: MIN**

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

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

---

**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>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td>D</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Every reference is a page fault!
- **MIN Does much better:**

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

- 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

Clock Algorithm: Not Recently Used

- Single Clock Hand:
  Advances only on page fault!
  Check for pages not used recently
  Mark pages as not used recently

- 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 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

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

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.