CSCC 69H3

Operating Systems
Winter 2013
Professor Bianca Schroeder
U of T

Paging

- Partition memory into equal, fixed-size chunks
  - These are called page frames or simply frames
- Divide processes’ memory into chunks of the same size
  - These are called pages
- Possible page frame sizes are restricted to powers of 2 to simplify translation

Example of Paging

Suppose a new process, D, arrives needing 3 frames of memory

We can fit Process D into memory, even though we don’t have 3 contiguous frames available!

Address translation

- Source code
- Program binary
- Memory
- Process

Uses symbolic addresses (variable names)
Uses logical addresses (relative to start of stack frame)

How does address translation work for
- Static/dynamic partitioning?
- Paging?
Address translation: Partitioning schemes

- All memory used by process is contiguous in these methods
- Basic idea: add relative address to process starting (base) address to form real, or physical, address
- 2 registers, “base” and “limit”

Hardware for Relocation

- Basic idea: add relative address to process starting (base) address to form real, or physical, address
- 2 registers, “base” and “limit”
  - When process is assigned to CPU (i.e., set to “Running” state), load base register with starting address of process
  - Load limit register with last address of process

Address translation for Paging

- Need more than base & limit registers now
- Operating system maintains a page table for each process

What does an address specify now?
Support for Paging
- Operating system maintains page table for each process
- Page table records which physical frame holds each page
- Virtual addresses are now page number + page offset
- Page number = vaddr / page_size
- Page offset = vaddr % page_size
- Simple to calculate if page size is power-of-2

Example Address Translation
- Suppose addresses are 16 bits, pages are 4K (4096 bytes)
  - How many bits of the address do we need for offset?
  - 12 bits (2^12 = 4096)
  - What is the maximum number of pages for a process?
  - 2^4

Example Address Translation
- To translate virtual address: 0x3468
  - Extract page number (high-order 4 bits)
    - p = vaddr >> 12 == 3
  - Get frame number from page table
  - Combine frame number with page offset
    - offset = vaddr % 4096
    - addr = frame * 4096 + offset

Support for Paging
- Operating system maintains page table for each process
- Page table records which physical frame holds each page
- Virtual addresses are now page number + page offset
- Page number = vaddr / page_size
- Page offset = vaddr % page_size
- Simple to calculate if page size is power-of-2
- On each memory reference, processor translates page number to frame number and adds offset to generate a physical address
- Keep a “page table base register” to quickly locate the page table for the running process
Page Table Entries (PTE)

- Page table entries (PTEs) control mapping
  - Modify bit (M) says whether or not page has been written
    - Set when a write to a page occurs
  - Reference bit (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - Valid bit (V) says whether PTE can be used
    - Checked on each use of virtual address
  - Protection bits specify what operations are allowed on page
    - Read/write/execute
  - Page frame number (PFN) determines physical page
  - Not all bits are provided by all architectures

Page Lookups Overview

- What's wrong with this approach?
  - Need 2 references for address lookup (first page table, then actual memory)
- Idea: Use hardware cache of page table entries
  - Translation Lookaside Buffer (TLB)
  - Small hardware cache of recently used translations

TLBs

- Translate virtual page #s into PTEs (not physical addr)
  - Can be done in a single machine cycle
- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
TLBs

- TLBs are small (64 – 1024 entries)
- Still, address translations for most instructions are handled using the TLB
  - >99% of translations, but there are misses (TLB miss)...
- TLBs exploit locality
  - Processes only use a handful of pages at a time
  - Only need those pages to be "mapped"
  - Hit rates are therefore very important

Managing TLBs

- Who places translations into the TLB (loads the TLB)?
  - Hardware (Memory Management Unit)
  - Software loaded TLB (OS)

Managing TLBs (2)

- OS ensures that TLB and page tables are consistent
  - When the bits of a PTE change, OS needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple
What happens if not all pages of all processes fit into physical memory?

Page Table Entries (PTE)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>R</td>
<td>V</td>
<td>Prot</td>
<td>Page Frame Number</td>
</tr>
</tbody>
</table>

- Page table entries (PTEs) control mapping
  - Modify bit (M) says whether or not page has been written
    - Set when a write to a page occurs
  - Reference bit (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - Valid bit (V) says whether PTE can be used
    - Checked on each use of virtual address
  - Protection bits specify what operations are allowed on page
    - Read/write/execute
  - Page frame number (PFN) determines physical page
  - Not all bits are provided by all architectures

Page allocation & eviction

- What happens when new page is allocated?
  - Initially, pages are allocated from memory
  - When memory fills up:
    - Some other page needs to be evicted from memory
    - This is why physical memory pages are called “frames”
    - Evicted pages go to disk (the swap file)
  - When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When a process accesses the page, the invalid PTE will cause a trap (page fault)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process

Policy Decisions

- Page tables, MMU, TLB, etc. are mechanisms that make virtual memory possible
- Next, we'll look at policies for virtual memory management:
  - Fetch Policy – when to fetch a page
  - Placement Policy – where to put the page
  - Replacement Policy – what page to evict to make room?

Demand Paging

- Timing: Disk read is initiated when the process needs the page
- Request size: Process can only page fault on one page at a time, disk sees single page-sized read
- What alternative do we have?

Prepaging (aka Prefetching)

- Predict future page use at time of current fault
  - On what should we base the prediction? What if it's wrong?
Policy Decisions

- Page tables, MMU, TLB, etc. are mechanisms that make virtual memory possible
- Next, we'll look at policies for virtual memory management:
  - Fetch Policy – when to fetch a page
  - Demand paging vs. Prepaging
  - Placement Policy – where to put the page
  - Are some physical pages preferable to others?
  - Replacement Policy – what page to evict to make room?
  - Lots and lots of possible algorithms!

Evicting the best page

- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
- Replacement algorithms are evaluated on a reference string by counting the number of page faults
  - Let's start by cheating a little bit …
    - Assume we know the reference string – what is the best replacement policy in this case?

Placement Policy

- In paging systems, memory management hardware can translate any virtual-to-physical mapping equally well
- Why would we prefer some mappings over others?
  - NUMA (non-uniform memory access) multiprocessors
  - any processor can access entire memory, but local memory is faster
  - Cache performance
  - Choose physical pages to minimize cache conflicts
- These are active research areas!
Evicting the best page

- Reference string: 2,3,2,1,5,4,5,3,5,3,2

<table>
<thead>
<tr>
<th>Cold misses: first access to a page (unavoidable)</th>
<th>Capacity misses: caused by replacement due to limited size of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 2 2 2</td>
<td>X 5</td>
</tr>
</tbody>
</table>

- Lesson 1:
  - The best page to evict is the one never used again
  - Will never fault on it

Evicting the best page

- Reference string: 2,3,2,1,5,4,5,3,5,3,2

<table>
<thead>
<tr>
<th>Cold misses: first access to a page (unavoidable)</th>
<th>Capacity misses: caused by replacement due to limited size of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 2 2 2</td>
<td>X 5</td>
</tr>
</tbody>
</table>

- Lesson 2:
  - Never is a long time, so picking the page closest to “never” is the next best thing
  - Evicting the page that won’t be used for the longest period of time minimizes the number of page faults
  - Proved by Belady, 1966

Belady’s Algorithm

- Belady’s algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream (aka OPT or MIN)
  - Idea: Replace the page that will not be used for the longest period of time
  - Problem: Have to know the future perfectly
- Why is Belady’s useful then? Use it as a yardstick
  - Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could use some work
  - Random replacement is often the lower bound

What are possible replacement algorithms?

- First-in-first-out (FIFO)
- Least-recently-used (LRU)
- Least-frequently-used
- Most-frequently-used

• Many of these require book-keeping ...
• Let’s start with algorithms that require only information contained in PTE
Page Table Entries (PTE)

- **Modify (M)**
  - says whether or not page has been written
- **Reference (R)**
  - says whether page has been accessed
  - is cleared periodically (e.g. at clock interrupt)
- **Valid (V)**
  - says whether PTE can be used
- **Protection bits:**
  - what operations are allowed on page

<table>
<thead>
<tr>
<th>M</th>
<th>R</th>
<th>V</th>
<th>Prot</th>
<th>Page Frame Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>26</td>
</tr>
</tbody>
</table>

Not-Recently-Used (NRU)

Divide pages into 4 classes:
- **Class 1**: Not referenced, not modified
- **Class 2**: Not referenced, modified
- **Class 3**: Referenced, not modified
- **Class 4**: Referenced, modified

- Remove page at random from lowest-numbered class that’s not empty

First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Then again, maybe it’s not
  - We don’t have any info to say one way or the other
- FIFO suffers from “Belady’s Anomaly”
  - The fault rate might actually increase when the algorithm is given more memory (very bad)

Example of Belady’s anomaly

- Reference string: 0,1,2,3,0,1,4,0,1,2,3,4

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Youngest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Oldest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Anomaly</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Youngest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Oldest</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 3 frames, 9 faults
- 4 frames, 10 faults
Second-Chance

- Idea:
  - FIFO (First-in-first-out) considers only age
  - NRU (Not recently used) considers only usage
  - Maybe we should combine the two!

- Second chance algorithm:
  - Don’t evict the oldest page if it has been used.
  - Evict the oldest page that has not been used.
  - Pages that are used often enough to keep reference bits set will not be replaced

Modelling Clock

- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

Implementing Second Chance (clock)

- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the ref bit is on, turn it off and go to next page
  - If the ref bit (aka use bit) is off, it hasn’t been used recently
    - What is the minimum “age” if ref bit is off?
- Arm moves quickly when pages are needed
- Low overhead when plenty of memory
- If memory is large, “accuracy” of information degrades