EECS 482
Introduction to Operating Systems

Winter 2019

Manos Kapritsos

Thanks to Harsha Madhyastha and Peter Chen for the slides and notes
Recap: Paging

- Both address spaces and physical memory broken up into fixed size pages
if (virtual page is invalid or non-resident or protected) {
    trap to OS fault handler; retry
} else {
    physical page # = pageTable[virtual page #].physPageNum
}
Paging

● Pros?
  ◆ Flexible memory allocation/growing address space
  ◆ Virtual memory
  ◆ No external fragmentation
  ◆ Flexible sharing

● Cons?
  ◆ Large page tables

● How to modify paging to reduce space needed for translation data?
Multi-level Paging

- Standard page table is a simple array
- Multi-level paging generalizes this into a tree

Example: Two-level page table with 4KB pages
- Index into level 1 page table: virtual address bits 31-22
- Index into level 2 page table: virtual address bits 21-12
- Page offset: bits 11-0
Multi-level Paging

How does this let translation data take less space?
Sparse Address Space

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>283</td>
</tr>
<tr>
<td>3</td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
<tr>
<td>1048572</td>
<td>invalid</td>
</tr>
<tr>
<td>1048573</td>
<td>1078</td>
</tr>
<tr>
<td>1048574</td>
<td>48136</td>
</tr>
<tr>
<td>1048575</td>
<td>60</td>
</tr>
</tbody>
</table>
Sparse Address Space

<table>
<thead>
<tr>
<th>Bits 31-22</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>389</td>
</tr>
<tr>
<td>1</td>
<td>invalid</td>
</tr>
<tr>
<td>2</td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
<tr>
<td>1021</td>
<td>invalid</td>
</tr>
<tr>
<td>1022</td>
<td>invalid</td>
</tr>
<tr>
<td>1023</td>
<td>7046</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 21-12</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>283</td>
</tr>
<tr>
<td>3</td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 21-12</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
<tr>
<td>1020</td>
<td>invalid</td>
</tr>
<tr>
<td>1021</td>
<td>1078</td>
</tr>
<tr>
<td>1022</td>
<td>48136</td>
</tr>
<tr>
<td>1023</td>
<td>60</td>
</tr>
</tbody>
</table>
Multi-level paging

- How to share memory between address spaces?
- What must be changed on a context switch?

- Pros:
  - Easy memory allocation
  - Flexible sharing
  - Space efficient for sparse address spaces

- Cons?
  - Two extra lookups per memory reference
Translation lookaside buffer

- TLB caches virtual page # to PTE mapping
  - Cache hit → Skip all the translation steps
  - Cache miss → Get PTE, store in TLB, restart instruction

- Does this change what happens on a context switch?
Page Replacement

- Not all valid pages may fit in physical memory
  - Some pages are swapped out to disk

- To read in a page from disk, some resident page must be swapped out to disk

- Which page to evict when you need a free page?
  - Goal: minimize page faults
Replacement policies

- Random
- FIFO
  - Replace page brought into memory longest time ago
- Optimal?
Replacement policies

- LRU (least recently used)
  - Approximates OPT by using past reference pattern
    - If page hasn’t been used for a while, it probably won’t be used for a long time in the future

- LRU is hard to implement exactly
  - Can we simplify LRU by approximating it?
The “referenced” bit

- Most MMUs maintain a “referenced” bit for each resident page
  - Set by MMU when page is read or written
  - Can be cleared by OS

- How to use reference bit to identify old pages?
- Why maintain reference bit in hardware?
Clock replacement algorithm

- Arrange resident pages around a clock

Algorithm to select page for eviction:
- Consider page pointed to by clock hand
- If not referenced, page hasn’t been accessed since last sweep → Evict
- If referenced, page has been referenced since last sweep
  » What to do?

- What if all pages have been referenced since last sweep?
- What about new pages?
A nice feature of clock is that it only adds overhead when you need to evict a page.
Administrivia

- Project 2 due today!
- Midterm on Friday
- Traveling this week
  - Office hours canceled
  - Wednesday lecture by Baris Kasikci

**You can start working on Project 3**
- Covered most of the material you need to know to do the project
Page eviction

- Where to evict page to?
- When do you NOT need to write page to disk?

- Why not write to disk on every store?
- The page that is brought from disk must wait while some page is evicted
  - How could you avoid this (sometimes)?
- DON’T use these optimizations in Project 3!
if (virtual page is non-resident or protected) {
    trap to OS fault handler
    retry access
} else {
    physical page # = pageTable[virtual page #].physPageNum
    pageTable[virtual page #].referenced = true
    if (access is write) {
        pageTable[virtual page #].dirty = true
    }
    access physical memory
}
### Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Resident</th>
<th>Read/Write enabled</th>
<th>Dirty</th>
<th>Referenced</th>
</tr>
</thead>
</table>

- Why no valid bit in PTE?
- For valid non-resident pages, does PTE contain disk block?
- Can we make do without resident bit?
## Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Read/Write enabled</th>
<th>Dirty</th>
<th>Referenced</th>
</tr>
</thead>
</table>

- Can we make do without dirty bit?
## Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Read/Write enabled</th>
<th>Referenced</th>
</tr>
</thead>
</table>

- Can we make do without referenced bit?
- Application too may want to control protection
  - Not in project 3
## Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>read_enabled</th>
<th>write_enabled</th>
</tr>
</thead>
</table>

Address Space Management

- How to manage a process’s accesses to its address space?
  - Kernel sets up page table per process and manages which pages are resident
  - MMU looks up page table to translate any virtual address to a physical memory address

- What about kernel’s address space?
- How does MMU handle kernel’s loads and stores?
Storing Page Tables

- Two options:
  - In physical memory
  - In kernel’s virtual address space
- Difference: Is PTBR a physical or virtual addr?
- Pros and cons

- Project 3 uses second option
  - Kernel’s address space managed by infrastructure
Kernel vs. user address spaces

- Can you evict the kernel’s virtual pages?

- How can kernel access specific physical memory addresses (e.g., to refer to translation data)?
  - Kernel can issue untranslated address (bypass MMU)
  - Kernel can map physical memory into a portion of its address space (e.g., vm_physmem in Project 3)