Paging: inside the OS

CS 241
February 8, 2012

Paging

- Solve the external fragmentation problem by using **fixed-size chunks** of virtual and physical memory
  - Virtual memory unit called a **page**
  - Physical memory unit called a **frame** (or sometimes **page frame**)

Diagram:

```
  virtual memory  physical memory
            (for one process)
     page 0       frame 0
     page 1       frame 1
     page 2       frame 2
     page 3       frame Y
     ...          ...  
     page X       frame Y
```
Application Perspective

- Application believes it has a single, contiguous address space ranging from 0 to $2^P - 1$ bytes
  - Where $P$ is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!

Lots of separate processes
- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science
Enabling data structure

- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM

```
<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory (DRAM)

- PP 0
  - VP 1
  - VP 2
  - VP 7
  - VP 4

- PP 3
  - VP 1
  - VP 2

Disk

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7`
Page hit: reference to VM word that is in physical memory (DRAM cache hit)
Page fault: reference to VM word that is not in physical memory (DRAM cache miss)
Handling page fault

- Page miss causes page fault (an exception)
Handling page fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Loads new frame into freed slot
Handling page fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Loads new frame into freed slot
- Offending instruction is restarted: page hit!
**Page Table Entry**

- **Typical PTE format (depends on CPU architecture!)**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>R</td>
<td>V</td>
<td>prot</td>
<td>physical page (frame) number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

- **Why is this 20 bits wide in the above example?**

- **Various bits accessed by MMU on each page access:**
  - **Modify bit:** Indicates whether a page is “dirty” (modified)
  - **Reference bit:** Indicates whether a page has been accessed (read or written)
  - **Valid bit:** Whether the PTE represents a real memory mapping
  - **Protection bits:** Specify if page is readable, writable, or executable
  - **Physical page number:** Physical location of page in RAM

- Modify bit
- Reference bit
- Valid bit
- Protection bits
- Physical page number
Address translation with a P.T.

Virtual address

Page table address for process

Valid bit = 0: page not in memory (page fault)

Physical address

Virtual page number (VPN)

Virtual page offset (VPO)

Physical page number (PPN)

Physical page offset (PPO)
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Question 1

- Isn’t it slow to have to go to memory twice every time?

- Yes, it would be… so, real MMUs don’t
Page table entries (PTEs) are cached in L1 like any other memory word
- PTEs may be evicted by other data references
- PTE hit still requires a small L1 delay

Solution: **Translation Lookaside Buffer (TLB)**
- Small, dedicated, super-fast hardware cache of PTEs in MMU
- Contains complete page table entries for small number of pages
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Question 2

- Isn’t the page table huge? How can it be stored in RAM?

- Yes, it would be... so, real page tables aren’t simple arrays
Multi-Level Page Tables

Suppose:
- 4KB ($2^{12}$) page size, 64-bit address space, 8-byte PTE

Problem:
- Would need a 32,000 TB page table!
- $2^{64} \times 2^{-12} \times 2^3 = 2^{55}$ bytes

Common solution:
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)
2-level page table hierarchy

Level 1 page table

Level 2 page tables

Physical memory

32 bit addresses, 4KB pages, 4-byte PTEs

2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack
Addr. translation with k-level PT

VIRTUAL ADDRESS

VPN 1  VPN 2  ...  VPN k  VPO

Level 1 page table

Level 2 page table

...  ...

Level k page table

PPN

PHYSICAL ADDRESS

PPN  PPO
Multilevel Page Tables

With two levels of page tables, how big is each table?
- Say we allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
- Primary page table is then $2^{10} \times 4$ bytes per PTE == 4 KB
- Secondary page table is also 4 KB
- Hey ... that's exactly the size of a page on most systems ... cool

What happens on a page fault?
- MMU looks up index in primary page table to get secondary page table
- MMU tries to access secondary page table
  - May result in another page fault to load the secondary table!
- MMU looks up index in secondary page table to get physical frame #
- CPU can then access physical memory address

Issues
- Page translation has very high overhead
  - Up to three memory accesses plus two disk I/Os!!
- TLB usage is clearly very important
Problem (from Tanenbaum)

Suppose:
- 32-bit address
- Two-level page table
- Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset

Question: How large are the pages and how many are there in the address space?
Problem (from Tanenbaum)

- Suppose:
  - 32-bit address
  - Two-level page table
  - Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset

- Question: How large are the pages and how many are there in the address space?
  - Offset is 12 bits
  - Page size $2^{12}$ bytes = 4KB
  - # Virtual pages = $(2^{32} / 2^{12}) = 2^{20}$
  - Note: driven by number of bits in offset
    - Independent of size of top and 2nd level
Question 3

- Is there any other super slick stuff can I do with page tables?

- Yes!
Paging as a tool for protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

---

### Process i:

<table>
<thead>
<tr>
<th></th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

### Process j:

<table>
<thead>
<tr>
<th></th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

---

### Physical Address Space

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
VM as a tool for sharing

- Process 1 maps the shared object.
VM as a tool for sharing

- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.
Protection + sharing example

- `fork()` creates exact copy of a process
  - Lots more on this next week…

- When we fork a new process, does it make sense to make a copy of all of its memory?
  - Why or why not?

- What if the child process doesn't end up touching most of the memory the parent was using?
  - `exec()` replaces a process with a new one
  - Extreme example *and common case*: What happens if a process does an `exec()` immediately after `fork()`?
Copy-on-write

- Idea: Give the child process access to the same memory, but don't let it write to any of the pages directly!
  - 1) Parent forks a child process
  - 2) Child gets a copy of the parent's page tables
    - They point to the same physical frames!!!
Copy-on-write

- All pages (both parent and child) marked read-only
  - Why?

<table>
<thead>
<tr>
<th>Parent</th>
<th>Parent's page tbl</th>
<th>Child's page tbl</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Reserved for OS)</td>
<td><img src="image" alt="Stack" /></td>
<td><img src="image" alt="Heap" /></td>
<td><img src="image" alt="Heap" /></td>
</tr>
<tr>
<td><img src="image" alt="Initialized vars" /></td>
<td><img src="image" alt="Uninitialized vars" /></td>
<td><img src="image" alt="Initialized vars" /></td>
<td><img src="image" alt="Uninitialized vars" /></td>
</tr>
<tr>
<td><img src="image" alt="Code" /></td>
<td></td>
<td><img src="image" alt="Code" /></td>
<td></td>
</tr>
</tbody>
</table>
Copy-on-write

What happens when the child reads the page?
- Just accesses same memory as parent .... niiiiice

What happens when the child writes the page?
- Protection fault occurs (page is read-only!)
- OS copies the page and maps it R/W into the child's addr space

```
Parent
(Reserved for OS)  
<table>
<thead>
<tr>
<th>Stack</th>
<th></th>
<th>Parent's page tbl</th>
<th>Child's page tbl</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uninitialized vars</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialized vars</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(Reserved for OS)  
| Stack    |          |                  |                  |       |
| Heap     |          |                  |                  |       |
| Uninitialized vars |         |                  |                  |       |
| Initialized vars   |         |                  |                  |       |
| Code              |         |                  |                  |       |
```
Copy-on-write

- What happens when the child **reads** the page?
  - Just accesses same memory as parent .... niiiiiice

- What happens when the child **writes** the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space
Copy-on-write

What happens when the child **reads** the page?
- Just accesses same memory as parent .... niiiiice

What happens when the child **writes** the page?
- Protection fault occurs (page is read-only!)
- OS copies the page and maps it R/W into the child's addr space
Another sharing example

- Can also share code segment

Shell #1

- (Reserved for OS)
- Heap
- Uninitialized vars
- Initialized vars
- Code

Shell #2

- (Reserved for OS)
- Stack
- Heap
- Uninitialized vars
- Initialized vars
- Code

Physical Memory

- Code for shell

Same page table mapping!
Benefits of sharing pages

How much memory savings do we get from sharing pages across identical processes?

- A lot! Use the “top” command...

```
Process: 68 total, 2 running, 1 stuck, 65 sleeping... 246 threads 13:17:30
Load Avg: 0.75, 0.58, 0.52   CPU usage:  7.7% user, 17.9% sys, 74.4% idle
SharedLibs: num = 223, resident = 33.3M code, 4.61M data, 4.80M LinkEdit
MemRegions: num = 17413, resident = 208M + 11.0M private, 546M shared
PhysMem: 618M wired, 261M active, 130M inactive, 1010M used, 13.9M free
VM: 9.79G + 150M   635052(61) pageins, 455424(8) pageouts
```

<table>
<thead>
<tr>
<th>PID</th>
<th>COMMAND</th>
<th>%CPU</th>
<th>TIME</th>
<th>#TH</th>
<th>#RTS</th>
<th>#REGS</th>
<th>PR/VT</th>
<th>RS/HD</th>
<th>RSIZE</th>
<th>VSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3784</td>
<td>Grab</td>
<td>5.0%</td>
<td>0:00:51</td>
<td>3</td>
<td>126</td>
<td>159</td>
<td>2.23M+</td>
<td>7.25M+</td>
<td>16.8M+</td>
<td>216M+</td>
</tr>
<tr>
<td>3781</td>
<td>less</td>
<td>0.0%</td>
<td>0:00:02</td>
<td>1</td>
<td>13</td>
<td>17</td>
<td>148K</td>
<td>304K</td>
<td>404K</td>
<td>26.7M</td>
</tr>
<tr>
<td>3776</td>
<td>sh</td>
<td>0.0%</td>
<td>0:00:00</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>88.8K</td>
<td>608K</td>
<td>364K</td>
<td>27.1M</td>
</tr>
<tr>
<td>3777</td>
<td>sh</td>
<td>0.0%</td>
<td>0:00:00</td>
<td>1</td>
<td>13</td>
<td>16</td>
<td>68.8K</td>
<td>608K</td>
<td>544K</td>
<td>27.1M</td>
</tr>
<tr>
<td>3776</td>
<td>man</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>13</td>
<td>16</td>
<td>184K</td>
<td>264K</td>
<td>460K</td>
<td>26.7M</td>
</tr>
<tr>
<td>3752</td>
<td>bash</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>14</td>
<td>16</td>
<td>228K</td>
<td>696K</td>
<td>816K</td>
<td>27.1M</td>
</tr>
<tr>
<td>3751</td>
<td>login</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>16</td>
<td>40</td>
<td>172K</td>
<td>300K</td>
<td>636K</td>
<td>26.9M</td>
</tr>
<tr>
<td>3748</td>
<td>top</td>
<td>12.8%</td>
<td>0:23:16</td>
<td>1</td>
<td>25</td>
<td>20</td>
<td>704K</td>
<td>300K</td>
<td>1.14M</td>
<td>27.0M</td>
</tr>
<tr>
<td>3725</td>
<td>bash</td>
<td>0.0%</td>
<td>0:00:02</td>
<td>1</td>
<td>14</td>
<td>16</td>
<td>228K</td>
<td>696K</td>
<td>812K</td>
<td>27.1M</td>
</tr>
<tr>
<td>3724</td>
<td>login</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>16</td>
<td>40</td>
<td>172K</td>
<td>300K</td>
<td>636K</td>
<td>26.9M</td>
</tr>
<tr>
<td>3722</td>
<td>Terminal</td>
<td>0.2%</td>
<td>0:02:31</td>
<td>6</td>
<td>92</td>
<td>140</td>
<td>2.25M</td>
<td>11.1M</td>
<td>10.3M</td>
<td>216M</td>
</tr>
<tr>
<td>3719</td>
<td>WinAppHelp</td>
<td>0.0%</td>
<td>0:00:05</td>
<td>1</td>
<td>57</td>
<td>95</td>
<td>716K</td>
<td>4.10M</td>
<td>3.00M</td>
<td>198M</td>
</tr>
<tr>
<td>3713</td>
<td>mdenp</td>
<td>0.0%</td>
<td>0:00:00</td>
<td>4</td>
<td>68</td>
<td>119</td>
<td>1.59M</td>
<td>3.16M</td>
<td>4.64M</td>
<td>57.8M</td>
</tr>
<tr>
<td>3675</td>
<td>iTunes</td>
<td>3.5%</td>
<td>0:51:76</td>
<td>9</td>
<td>193</td>
<td>370</td>
<td>7.12M</td>
<td>12.1M+</td>
<td>10.2M</td>
<td>263M</td>
</tr>
<tr>
<td>3670</td>
<td>AddressBook</td>
<td>0.0%</td>
<td>0:02:58</td>
<td>1</td>
<td>92</td>
<td>179</td>
<td>2.21M</td>
<td>5.56M</td>
<td>15.2M</td>
<td>216M</td>
</tr>
<tr>
<td>3659</td>
<td>Mail</td>
<td>0.0%</td>
<td>0:59:65</td>
<td>8</td>
<td>172</td>
<td>415</td>
<td>25.3M</td>
<td>10.9M+</td>
<td>27.2M</td>
<td>258M</td>
</tr>
<tr>
<td>3084</td>
<td>Skype</td>
<td>0.7%</td>
<td>17:20:32</td>
<td>18</td>
<td>240</td>
<td>452</td>
<td>23.9M</td>
<td>8.65M+</td>
<td>20.8M</td>
<td>304M</td>
</tr>
<tr>
<td>655</td>
<td>vfstool</td>
<td>0.0%</td>
<td>0:00:07</td>
<td>2</td>
<td>14</td>
<td>29</td>
<td>120K</td>
<td>308K</td>
<td>256K</td>
<td>32.1M</td>
</tr>
</tbody>
</table>
Summary

- Paging implementation
  - Basics: get page off disk if necessary (*page fault*) and then map virtual to physical address
  - Problem: Mapping requires extra memory access (solution?)
  - Problem: Page table can get huge (solution?)

- Paging enables flexible use of memory
  - Protection
  - Sharing (e.g., copy-on-write defers writes as long as possible)
  - Caching
    - Q: How do I choose which page to evict when swapping?