20 Lecture: Virtual Memory

Outline:
- Announcements
- The CSU
- From last time: Dynamic segmentation
- Memory allocation strategies
- Dealing with memory size
- Virtual Memory (Fotheringham, 1961)
  - What it’s all about
- Translation Logistics
  - Where to keep the page table?
  - Possible approaches to dealing with size
- Translation Lookaside Buffers
  - What about really big memories?
- Limited memory size
- Inverted page tables

20.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Final Redemption Plan
- Common Final: Saturday, June 8th, 10:10-1:00
- Banquets
- be aware of which version of minix you’re running
- Minix booting info: be specific:
  ```
  c0d0p0s0> image=/boot/image/testimage
  c0d0p0s0> boot
  ```
  Also: `make hdboot`
- RTFM/RTFB/RTFA
- Remember, your SecretKeepers shouldn’t interact with stdin or stdout (stderr, is ok for error messages). They only interact with the filesystem through messages.
- Can you make the system busy?
20.2 From last time: Dynamic segmentation

More complicated (no such thing as a free lunch)

Problems:

- **fragmentation** (Leads to the same partitioning problems as before. Jobs can’t find spaces big enough to run.)

  Fragmentation occurs when the manner in which a subdividable resource (typically storage) is allocated leads to unusable portions.

  **internal** fragmentation occurs when the allocated block is larger than can be used by the process. The remaining unused space inside the block is lost to the system.

  **external** fragmentation occurs when the spaces between allocated blocks become too small to be usefully allocated to another process, even though sufficient total quantities of the resource are available.

What about fragmentation?

  - Compaction?
    
    We can move processes, but this is computationally expensive and we may have to re-link (w/o base registers)?
    
    What about pointers in this environment?

- **process growth**

  **padding** We can pad processes—leave extra space around them “just in case”.

    - Inward-facing (neat, bounded)
    - Outward facing (allows for swapping)

  **swapping** bump one process to disk and restart later.

  **divide** divide the process into stack and heap parts that each grow in different directions.

20.3 Memory allocation strategies

- **bitmaps** Simple manipulation (look for an appropriate block)
  
  but painfully slow

- **lists** keep a list of adjacent (process/hole) blocks. Merge holes as possible

  Either way, options for picking a location:

  - first fit
  - next fit
  - best fit oddly enough, worse than first fit. (leaves little holes)
  - worst fit
20.4 Dealing with memory size

And now we return to the original problem: Too little memory.
But we don’t really use all of memory at once, do we?
All of these solutions are based on what would come to be known as the principle of locality of reference and of a working set (Denning, 1968, 1980).

- overlays
- virtual memory

20.5 Virtual Memory (Fotheringham, 1961)

transparent It’s there, but you can’t see it.

virtual You can see it, but it isn’t there.

20.5.1 What it’s all about

The way it works

- Programs exist in a virtual address space
- Virtual memory is divided into pages
- Physical memory is divided into frames
- Non-resident pages are stored in backing store (swap space) and brought in as needed.
- Virtual addresses are translated into physical addresses by the MMU (Memory Management Unit) using page tables

- Page tables contain:
  - virtual page number (index)
  - cache enabled (C)
  - modified bit (M)
  - referenced bit (R)
  - protection bits (rwx?)
  - present bit (P)
  - physical frame number

- Attempting to reference an unmapped page causes a page fault.

VM translation is illustrated in Figure 32
What it provides

1. expanded memory
2. relocation
3. isolation

The last one, isolation is important to consider: no process can refer to another process’s memory. So how can processes communicate?

Only through the kernel because the kernel has access to physical memory. (At least part of it—the pager—must have access. (How does it do that? It’s using VM hardware, too.))

---

20.6 Translation Logistics

Consider: page tables are large and must be accessed quickly:

For 4KB Pages, 27 bits per entry:

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| C  | M  | R  | r  | w  | x  | P  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

At one word (32 bits) per Page table entry:

| Virtual Memory size: | 4Gb \((2^{32})\) |
| Page Size:           | 4k \((2^{12})\)   |
| No. Pages:           | 1048576 \((2^{20})\) |
| P.T. Size            | 4Mb               |

Figure 32: Virtual Memory translation in the MMU (4k pages)
Remember, that’s per process because every process has its own virtual address space.

20.6.1 Where to keep the page table?

- In the MMU? (fast translation, slow loading)
  With a 400 MHz memory bus, to move 4Mb would take 0.0026s, and that’s just to move the data ignoring any instruction overhead. Of course, you’d have to move the old table out, too, so, that’d be 5.2 msec per context switch. Given that a quantum is 100msec, that’s a minimum of 5% overhead!

- in Memory (slower translation, possibly faster loading)

20.6.2 Possible approaches to dealing with size

- restrict virtual memory
- page the page tables (multi-level page tables?)
- Even so, many references per instruction:
  TLB: a small, associative page-table cache.

20.6.3 Translation Lookaside Buffers

- Small (32–64 entry) associative cache of page table entries
- Management (hardware vs. software.)

  **hardware management** A TLB miss causes an ordinary page lookup and the MMU replaces a TLB entry.

  **software management** A TLB miss causes a trap. This has to be handled efficiently, but with reasonably large TLBs the miss rate can be kept low.
  Why do this? This leads to a much simpler (i.e. faster) TLB.

  Idea: Keep a cache of recently evicted TLB entries around and check them first. (Consider the case where a multi-level page table must be searched.)