Memory Management

Ken Birman

How to create a process?

- On Unix systems, executable read by loader

  ![Diagram showing the process of creating a process]

  - Compiler: generates one object file per source file
  - Linker: combines all object files into one executable
  - Loader: loads executable in memory

What does process look like?

- Process divided into segments, each has:
  - Code, data, heap (dynamic data) and stack (procedure calls)

  ![Diagram showing process segments]

Processes in Memory

- We want many of them to coexist

  ![Diagram showing memory allocation]

  - Issues:
    - PowerPoint needs more memory than there is on machine?
    - What is visual studio tries to access 0x0005?
    - If PowerPoint is not using enough memory?
    - If OS needs to expand?

Issues

- Protection: Errors in process should not affect others
- Transparency: Should run despite memory size/location

![Diagram showing issues]

Scheme 1: Load-time Linking

- Link as usual, but keep list of references
- At load time: determine the new base address
  - Accordingly adjust all references (addition)

  ![Diagram showing load-time linking]

  - Issues: handling multiple segments, moving in memory
Scheme 2: Execution-time Linking
- Use hardware (base + limit reg) to solve the problem
- Done for every memory access
- Relocation: physical address = logical (virtual) address + base
- Protection: is virtual address < limit?

- When process runs, base register = 0x3000, bounds register = 0x2000. Jump addr = 0x2000 + 0x3000 = 0x5000

**Logical and Physical Addresses**

**Logical Address View**
- 0x3000
- 0x1000
- 0

**Physical Address View**
- MMU
- OS

**Segmentation**
- Processes have multiple base + limit registers
- Processes address space has multiple segments
- Each segment has its own base + limit registers
- Add protection bits to every segment

**Types of Segments**
- Each “file” of a program (or library) typically results in
  - An associated code segment (instructions)
  - The “initialized” data segment for that code
  - The zeroed part of the data segment (“bss”)
- Programs also have
  - One heap segment – grows “upward”
  - A stack segment for each active thread (grows down; all but first have a size limit)
**Lots of Segments**
- Bottom line: a single process might have a whole lot of active segments!
- And it can add them while it runs, for example by linking to a DLL or forking a thread
- Address space is a big range with chunks of data separated by gaps
- It isn’t uncommon for a process on a modern platform to have hundreds of segments in memory...

**Mapping Segments**
- Segment Table
  - An entry for each segment
  - Is a tuple \(\text{base}, \text{limit}, \text{protection}\)
- Each memory reference indicates segment and offset

**Segmentation Example**
- If first two bits are for segments, next 12 for offset

<table>
<thead>
<tr>
<th>Seg</th>
<th>base</th>
<th>bounds</th>
<th>rw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0x6ff</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x0000</td>
<td>00</td>
</tr>
</tbody>
</table>

- where is 0x0240?
- 0x1108?
- 0x265c?
- 0x3002?
- 0x1700?

**Segmentation: Discussion**
- Advantages:
  - Allows multiple segments per process
  - Easy to allow sharing of code
  - Do not need to load entire process in memory
- Disadvantages:
  - Extra translation overhead:
    - Memory & speed
  - An entire segment needs to reside contiguously in memory! 
    \(\Rightarrow\) Fragmentation

**Fragmentation**
- “Free memory is scattered into tiny, useless pieces”
- External Fragmentation:
  - Variable sized pieces \(\Rightarrow\) many small holes over time
- Internal Fragmentation:
  - Fixed sized pieces \(\Rightarrow\) internal waste if entire piece is not used

**Paging**
- Divide memory into fixed size pieces
- Called “frames” or “pages”
- Pros: easy, no external fragmentation
Mapping Pages

- If $2^m$ virtual address space, $2^n$ page size
  - $(m-n)$ bits to denote page number, $n$ for offset within page

Translation done using a Page Table

![Mapping Pages Diagram]

Paging: Hardware Support

- Entire page table (PT) in registers
  - PT can be huge ~ 1 million entries
- Store PT in main memory
  - Have PTBR point to start of PT
- Solution: a special cache (next slide)

Each time we context switch, must switch the current PT too, so that the process we execute will be able to see its memory region

Paging: Discussion

- Advantages:
  - No external fragmentation
  - Easy to allocate
  - Easy to swap, since page size usually same as disk block size
- Disadvantages:
  - Space and speed
    - One PT entry for every page, vs. one entry for contiguous memory for segmentation

Paging + Segmentation

- Paged segmentation
  - Handles very long segments
  - The segments are paged
- Segmented Paging
  - When the page table is very big
    - Segment the page table
    - Let's consider System 370 (24-bit address space)

Size of the page

- Small page size:
  - High overhead:
    - What is size of PT if page size is 512 bytes, and 32-bit addr space?
- Large page size:
  - High internal fragmentation

![Size of the page Diagram]
Segmented Paging Example

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>0x14</td>
<td>R</td>
</tr>
<tr>
<td>0x0000</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>0x0d</td>
<td>RW</td>
</tr>
</tbody>
</table>

Table entry addresses

- 0x001f
- 0x0011
- 0x2020
- 0x0013
- 0x000c
- 0x000f
- 0x009f
- 0x006
- 0x1000

Segment table

- 0x2000: read?
- 0x2004: read?
- 0x2008: read?
- 0x200c: write?

Handling PT size

- Segmented Paging removes need for contiguous allocation
- OS maintains one copy of PT for every process
- Other approaches:
  - Hierarchical Paging: Page the page table
  - Hashed Page Table: Each entry maps to linked list of pages
  - Inverted Page Table: Map from Frame to (VA, process) instead of VA to frame per process

Virtual Memory

- Each process has illusion of large address space
  - $2^{32}$ for 32-bit addressing
- However, physical memory is much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: Some addresses reside in disk

What is virtual memory?

- Virtual Memory: Some addresses reside in disk

What does VM work?

- Modify Page Tables with another bit ("is present")
  - If page in memory, is_present = 1, else is_present = 0
  - If page is in memory, translation works as before
  - If page is not in memory, translation causes a page fault

Virtual Memory

- Load entire process in memory (swapping), run it, exit
  - Is slow (for big processes)
  - Wasteful (might not require everything)
- Solutions: partial residency
  - Paging: only bring in pages, not all pages of process
  - Demand paging: bring only pages that are required
  - Where to fetch page from?
    - Have a contiguous space in disk: swap file (pagefile.sys)
Page Faults

- On a page fault:
  - OS finds a free frame, or evicts one from memory (which one?)
  - Want knowledge of the future?
  - Issues disk request to fetch data for page (what to fetch?)
  - Just the requested page, or more?
  - Block current process, context switch to new process (how?)
  - Process might be executing an instruction
  - When disk completes, set present bit to 1, and current process in ready queue

When to fetch?

- Just before the page is used!
- Need to know the future
- Demand paging:
  - Fetch a page when it faults
- Prepaging:
  - Get the page on fault + some of its neighbors, or
  - Get all pages in use last time process was swapped

Page Replacement Algorithms

- Random: Pick any page to eject at random
  - Used mainly for comparison
- FIFO: The page brought in earliest is evicted
  - Ignores usage
  - Suffers from “Belady’s Anomaly”
    - Fault rate could increase on increasing number of pages
    - E.g., 0123 01401234 with frame sizes 3 and 4
- OPT: Belady’s algorithm
  - Select page not used for longest time
- LRU: Evict page that hasn’t been used the longest
  - Past could be a good predictor of the future

Resuming after a page fault

- Should be able to restart the instruction
- For RISC processors this is simple:
  - Instructions are idempotent until references are done
- More complicated for CISC:
  - E.g., move 256 bytes from one location to another
  - Possible Solutions:
    - Ensure pages are in memory before the instruction executes

What to replace?

- Page Replacement
  - When process has used up all frames it is allowed to use
  - OS must select a page to eject from memory to allow new page
  - The page to eject is selected using the Page Replacement Algo
- Goal: Select page that minimizes future page faults

Example: FIFO, OPT

Reference stream is A B C A B D A D B C

<table>
<thead>
<tr>
<th>Frame</th>
<th>FIFO</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Faults: toss C, then A or D

<table>
<thead>
<tr>
<th>Frame</th>
<th>FIFO</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 Faults: toss A, then C

18/02/2009
Implementing Perfect LRU

- On reference: Time stamp each page
- On eviction: Scan for oldest frame
- Problems:
  - Large page lists
  - Timestamps are costly
- Approximate LRU
  - LRU is already an approximation!

LRU: Clock Algorithm

- Each page has a reference bit
- Set on use, reset periodically by the OS
- Algorithm:
  - FIFO + reference bit (keep pages in circular list)
  - Scan: if ref bit is 1, set to 0, and proceed. If ref bit is 0, stop and evict.
- Problem:
  - Low accuracy for large memory

LRU with large memory

- Solution: Add another hand
  - Leading edge clears ref bits
  - Trailing edge evicts pages with ref bit 0
- What if angle small?
- What if angle big?

Clock Algorithm: Discussion

- Sensitive to sweeping interval
- Fast: lose usage information
- Slow: all pages look used
- Clock: add reference bits
  - Could use (ref bit, modified bit) as ordered pair
  - Might have to scan all pages
- LFU: Remove page with lowest count
  - No track of when the page was referenced
  - Use multiple bits. Shift right by 1 at regular intervals.
- MFU: remove the most frequently used page
- LFU and MFU do not approximate OPT well

Page Buffering

- Cute simple trick: (XP, 2K, Mach, VMS)
  - Keep a list of free pages
  - Track which page the free page corresponds to
  - Periodically write modified pages, and reset modified bit

Allocating Pages to Processes

- Global replacement
  - Single memory pool for entire system
  - On page fault, evict oldest page in the system
  - Problem: protection
- Local (per-process) replacement
  - Have a separate pool of pages for each process
  - Page fault in one process can only replace pages from its own process
  - Problem: might have idle resources