Lecture 17: Segmentation I (Apr 5, 2005) Yap

Apr 5, 2005

1 ADMIN

- Today, we return to a topic we skipped over in our initial introduction to memory management, viz., segmentation.

2 Review

- Q: How many instructions are there in STML?

  A: 16

  Q: In STML, how would you load a constant, say 123, into register R1?

  A: 3 steps:

  (1) Choose a location L that is past the end of your program instructions.

  (2) At the appropriate place where you want to load the constant, you execute the instruction:

  LOA R1,L

  (3) At location L in your program, simply enter the value 123 (in Hex).

  REMARK: see the sort.stm program for example

  Q: What is "123" in Hex?

  A: 123 is slightly less than 126=$2^7 = 8(2^4)$. So the most significant digit is 7. But $123 - 7(2^4) = 11 = 0xB$. Hence $123 = 0x7B$.

3 Review of Memory Management

- Memory Hierarchy (capacity/speed tradeoff)
- E.g., Registers, Cache, RAM, Disk, Tape
- When we speak of "memory management" we are basically talking about the RAM part of memory!

Strictly speaking, there is also the Disk memory that lurks behind all our schemes. The issue is how to "represent parts of the Disk memory in RAM".

- The issues with managing memory can be divided into three distinct sets of problems: allocation, paging, segmentation.
- PROBLEM 1: Allocation and Swapping Problems
  1. How to allocate memory for processes?
  2. REMARK: "memory" means RAM
  3. SWAPPING is an another wrinkle in the allocation problem – processes can be swapped out.
  4. Two basic functions: free and allocate
  5. Keep track of allocation of memory to processes
  6. Use 2 linked lists: allocated and free list
  7. Each list, may be ordered by address
  8. If ordered, may have cross-links between 2 lists
  9. Allocation policies: first fit, best fit, worst fit, quick fit
- PROBLEM 2: Paging Problem
  1. Goes hand-in-hand with virtual memory
  2. REMARK: 2 senses of physical memory (in RAM and in DISK)
  3. Basic parameter: page size
  4. Physical RAM memory divided into page frames
  5. Page table to track mapping from virtual pages to frames
     ENTRY: (present-bit, page frame number, R-bit, M-bit, cache-enable-bit, protection, ...)
  6. Need method to translate virtual address to physical addresses on DISK. USUALLY, MMU
  7. MAIN TOPIC: page eviction policies/algorithms
     (NOTE: page placement is a non-issue because pages are all the same size)
- PROBLEM 3: Segmentation Problem
  1. This lecture!
- REVIEW OF PAGE TABLES
  1. Since segmentation will be integrated with paging, we need to recall some details on paging

2. Address = (page#, offset)
3. Each page has a Page Table Entry (PTE) in the page table
4. This information is hardware, not OS, oriented. (Because this must be fast and automatic)
5. What is in PTE?

   – PRESENT or VALID bit: TRUE iff page is loaded in a frame.

   Page fault generated if invalid page is accessed.

   – FRAME NUMBER: virtual to RAM address translation

   – R- or REFERENCED BIT: used in page eviction policies

   – M- or MODIFIED or DIRTY BIT: used to decide if writing to disk is needed.

   – PROTECTION BITS: read/write/execute permissions.

   This is probably more naturally done at the segment level.
6. Consider the transition graph for the four states of (R,M): (00), (01), (10), (11).

   – These 2 bits are very important and is very effectively used by the various paging algorithms

   – Here are some "puzzles":

   – How can we ever get into (01)?

   – Is there a transition INTO state (00)?

   – How can we ever begin in (00)? Demand paging seems to preclude this state!

   Ans: From (10) we can get to (00) after a clock tick This also accounts for (01).

   ALSO: instead of "demand paging", we may have proactive paging which brings in a block of pages (because it is cheap, or because there is a likelihood of using neighboring pages). Thus a page may start out with (00).
7. Multilevel Page Tables:

   – Page tables MUST reside in main memory (otherwise the whole purpose of paging is defeated)

   – But page tables can be very large! (How many entries in page table if you have a 80 GB disk, and 1 KB page size?)

   – Multilevel is important for reducing page tables.

   – The first level may be called the Page Directory.

   – Keep only the Page Directory in memory

   – With 2 levels, Address = (pageDir#,page2#, offset)

   – Can generalize to more levels
8. Translation Lookaside Buffers (TLB)

   – Why is this needed? Each memory reference requires 2 or 3 more memory references. We would like to speed this up.

– a kind of associative memory (content-addressable memory)

– It is the NATURAL kind (e.g., our brains). We do not recall information in our heads by specifying addresses...

– Explain "index field"

– The index field in TLB is the Page Number!

– Associative memories are expensive but fast

– So TLB's are small

– On a TLB miss, we do 3 things:

(a) We resort to the usual Page Table lookup

(b) We CHOOSE a TLB entry for replacement

(c) We put the new page into the TLB

4 Segmentation

- Motivation:

  Different parts of a process memory has different properties

  We would like to tag such "segments" with different protection and paging policies

- Example: Compiler process may have 5 "segments"

  1. Source text that is being compiled
  2. Symbol table, being built
  3. Table of constants
  4. Parse tree, being built
  5. Stack for procedure calls

  PROPERTIES:

  – The first 4 grows, but 5th may grow and shrink

- Example: (Tanenbaum, Sec.4.6.4) PDP-11 is a pioneer here, and is common in Unix

  – Two segments, called Instruction or **I-space** and Data or **D-space**.

  – I-space for (program) text is clearly executable and does not grow.

  – D-space can grow and may be read-only or both read and write.

  – Each entry in Process table will store pointers to the appropriate D-space page table and I-space page table.

  – This arrangement facilitates sharing (Tanenbaum, Sec.4.6.5). Thus, if two processes share a given I-space, they just need to have to use same I-space page table.

– Some issues arise in sharing of pages (we should have some way of indicating if a page is shared, so that a shared page is not evicted when a process using the page is evicted).

– Consider how we can exploit this in the "fork" system call of Unix. I-space can be shared between parent and child processes.

– D-space are not to be shared... but there is better way than just copying data pages automatically.

– We can have a "copy on write" rule – data pages can be shared UNTIL one process (child or parent) tries to write.

– All the above are discussed under paging, but it is relevant to segmentation (i.e., 2-segment system).

– Tanenbaum (Sec.4.6.6) also discuss the concept of a "paging daemon" to automatically keep pages up-to-date, etc.

- Concept/Implementation of **segmentation**

  – Each segment has a max length

  – Each segment address begins with 0, to its max length

  – Addresses comprise (seg#, offset).

  – WHEN combined with paging, address = (seg#, page#, offset)

  – like paging, but pages sizes are fixed, not segment sizes

  – Segment table

  – Each segment has an entry (STE=Segment Table Entry)

  – What is in STE?

  Has a "valid bit" (if segment is loaded)

  Has a base and limit

  Has a disk address

  – If segment is accessed, we check if it is loaded

  – If not loaded, the base, limit and disk address is restored from process table

  – We must solve the PLACEMENT and REPLACEMENT problems

  – An address lookup requires 3 memory references (STE, PTE, actual address). A TLB can help!

- Advantages:

  – linking is simplified (if one segment is modified, the other segments are unchanged)

  – shared library is possible (e.g., GUI libraries)

  – protection permission of segments can be customized (code segment is execute only, text segment may be read only, etc)

- Example: Traditional Unix has 3 segments
  - shared text (execute only)
  - data segment (global and static vars)
  - stack segment (automatic vars)