Nov 13, 2006
Lecture 12: Memory Management II, and STM

November 20, 2006

1 ADMIN

We finish up our discussion of Memory Management and will start Virtual Memory next. Please start to read Chapter 10.

We also introduce the STM ("Simple Toy Machine") which is a simulated CPU in which we learn to implement the various concepts we have learned. This is also the content of homework 4, to turn STM simulator

2 REVIEW

- Q: Suppose our logical address space is 32 bits and each page is 4 KB. How many entries would the page table have?

  A: 32 bits represents 4 Gigawords (assume each logical address refers to a word). 4 KB

- Q: What is a solution to fragmentation? Describe what is needed to solve this problem. What new issues arise?

  A: Use pages of fixed size, and allow non-contiguous physical memory allocation.

  You now need the notion of page table, (virtual) pages and frames (in main memory).

  This introduces INTERNAL fragmentation. Presumably, this is a more manageable problem.

  Q: Alternative formulation: What is "paging" a solution to?

  A: to the problem of fragmentation.

- Q: What are some solutions to huge page tables?

  A:

3 Segmentation

- Paging solves one problem, that of allocating memory.

  That is, we need to map physical (i.e., virtual) memory to physical memory.

- Segmentation solves another problem, that of providing structure to the allocated memory.

  In other words, the memory for a given process is not a homogeneous linear array of bytes, but can be viewed a variable size units called SEGMENTS.

- Now, we want to map "segmented memory" to logical memory (or to physical memory).

  Each "segmented memory" address is a pair (Segment-number, offset).

  This mapping is contained in the SEGMENT TABLE. In this table, we store a SEGMENT BASE and SEGMENT LIMIT for each Segment.

- E.g., a typical process will have these segments:

  symbol table, stack, data, routines, etc.

  When libraries are linked, we might get many separate segments.

- One benefit of segmentation is that we can achieve protection of segments.

  Protection bits can be set per segment. E.g., read-only, execute-only, etc.

  This makes sense since segments are often semantically meaningful units.

- Another benefit: code sharing.

4 Segmentation with Paging

- So far, we have tried to describe segmentation and paging as distinct mechanisms. In practice, we ought to combine the two. Let us look the implementation of this in Intel 80x86 (or x86) family of processors.

- The Intel 80x86 processors is perhaps the most famous family of CPU's in the world. It is so-called because the processors in this family had numbers ending in 86 (as in 8086, 80186, 80286, etc). It started in 1981 and each successive generation (286, 386, 486, 586, etc) is backwards compatible. The 586 family is also known as Pentium line. See Wikipedia under x86 Architecture or Pentium.

- Addresses are segmented in the 8086 family: 16-bit segment number, and a 16-bit offset.

- Please refer to Figure 9.21, page 355, in text. Our goal is to understand this figure.

- Each process has up to 16K segment, and each segment size is at most 4GB. Each page size is 4KB.

- The logical address space of processes is divided into 8K private segments and 8K shared segments.

  To manage these segments, we have a **local descriptor table** (LDT) for the private segments, and a **global descriptor table** (GDT) for the shared segments.

- Each LDT/GDT entry consists of an 8-byte segment descriptor (base location, limit, etc), refering to main memory.

  The logical address is a pair (SELECTOR,OFFSET). The SELECTOR is 16 bits (13-bit segment, 1-bit for GDT/LDT flag, 2-bit protection). The OFFSET is 32 bits.

- The CPU has 6 **segment registers** (so a program can access up to 6 segments at once).

  It also has 6 **microprogram registers** to hold descriptors from LDT or GDT.

- Refer to Figure 9.21, page 355, in text for the following. The main algorithm to form a 32-bit physical address:

  1. Start with the logical address =(SELECTOR,OFFSET).
  2. Use SELECTOR in segment register to load entry from LDT/GDT
  3. Use base in microprogram register to form a **linear address**. Use limit in microprogram register to check validity of address.
  4. If valid, offset (from logical address) is added to to linear address.
  5. Translate resulting address to physical address.

5 STM

- STM stands for Simple Toy Machine, which was written by Professor Ernie Davies.

  We will use the STM machine for our OS exercises.

  The source file for this machine is found in stm.c. The assembly language for this machine is called stml.

  We provide three sample programs: primes.stm, sort.stm, fraction.stm.

  The original version simply executes a single program at a time. Our homework is to modify the stm.c so that it can execute several programs simultaneously, one process for each program.

- Specification of STM: it has 1 MB of RAM, and this is divided up into 256K of words where each word is 4 bytes.

  We can address individual words in the RAM. Hence, the physical address space is 18 bits.

  There are 16 registers, named R0, R1,..., R15. R0 is the PC.

  There are 16 machine instructions. Thus the OpCode is 4 bits. The operands of each instruction consists of at most one 18-bit address (AD), and one to four registers (RA, RB, RC, RD). So every instruction can fit into a word (in fact, we only use bits 0-25, and bits 26-31 are not used). Here, bit 0 is the low-order bit and bit 31 is the hi-order bit.

OpCode	Operation	Operands	Meaning
0	LOA	RA, AD	Load contents at location AD to RA
1	STO	RA, AD	Store contents of RA to location AD
2	CPR	RA, RB	Copy RB into RA
3	LOI	RA, RB	RB holds the address of AD. Load the value of AD into RA.
4	STI	RA, RB	RA holds the address of AD. Store the value in RB into AD.
5	ADD	RA, RB, RC	RC = RA + RB
6	SUB	RA, RB, RC	RC = RA - RB
7	MUL	RA, RB, RC	RC = RA * RB
8	DIV	RA, RB, RC, RD	RC = RA / RB; RD = RA % RB.
9	ICR	RA	RA++
10	DCR	RA	RA–
11	GTR	RA, RB, RC	RC = (RA ¿ RB)
12	JMP	RA, AD	Jump to AD. (RA is unused).
13	IFZ	RA, AD	If (RA == 0) then goto AD.
14	JMI	RA	RA holds the address of AD. Jump to AD.
15	TRP		Trap to the kernel.

- Remark: the "I" in the abbreviations "LOI/STI/JMI" suggests Indirect Addressing, i.e., when the address is stored in a location rather than explicitly given.

- We must discuss the TRAP instruction. Register 15 and possibly 14 and 13 will be used. The value of R15 indicates the nature of the system call:

  - R15 == 0. Terminate. The process halts.
  - R15 == 1. Input. Read an integer from standard input. The value of the integer read is returned to the process in register R14. R13 is returned as 1 if an integer has been read and 0 if the end-of-file is reached. Any non-integer value in standard input causes an error.
  - R15 == 2. Output. Print on a new line to standard output in decimal the value of the integer held in register R14.
  - R15 ¿2. Future projects will involve defining other trap codes, for forking and for manipulating semaphores, etc.

6 STML and its Basic Interpreter

- STML is the associated assembly langauge of STM.
- This is essentially a binary file, but written in ASCII (sic!). The latter is for human consumption, of course.
- A stml source file is normally given the ".stm" extension. It has the following structure:
  - Line 1: name of process
  - Line 2: size of memory (in number of words) for the process. This includes space for code, data, variables, etc.
  - Line 3 onwards: 1 instruction or data item per line. Any line not beginning with a number is ignored. Any item beyond the first number is ignored.
- Let us go through a sample stm code, "fraction.stm".

```
fraction
33

% This program reads four integers A,B,C,D from standard input
% where A < B != 0.  It prints in the standard output,
% successively, the first D "digits" of A/B in base C.
% E.g., if (A,B,C,D)=(1,3,10,5), it prints out "3 3 3 3 3"
% This code is from Professor Davies.

0x1CF0     0  LOA R15 ONE
0xF        1  TRP           --- Read A
0xE12      2  CPR R1 R14    ---   and save in R1
0xF        3  TRP           --- Read B
0xE22      4  CPR R2 R14    ---   and save in R2
0xF        5  TRP           --- Read C
0xE32      6  CPR R3 R14    ---   and save in R3
0xF        7  TRP           --- Read D
0xE42      8  CPR R4 R14    ---   and save in R4
0x512B     9  GTR R2 R1 R5  --- If A > B then fail.
0x195D     A  IFZ R5 EXIT
0x192D     B  IFZ R2 EXIT   --- If B == 0 then fail.
0x1BA0     C  LOA R10 ZERO  --- R10 is the counter for the number of digits.
0x5A3B     D  GTR R3 R10 R5 --- If C <= 0 then fail.
0x195D     D  IFZ R5 EXIT
0x1DF0     E  LOA R15 TWO   --- The next series of traps will be prints.
         LOOP
0x5137    10  MUL R3 R1 R5  --- print (A*C)/B;
0x76258   11  DIV R5 R2 R6 R7
0x6E2     12  CPR R14 R6
0xF       13  TRP
0x712     14  CPR R1 R7     --- A := (A*C) % B;
0xA9      15  INC R10       --- Increment digit counter and compare to D.
0x5A46    16  SUB R4 R10 R5
0x195D    17  IFZ R5 EXIT
0x100C    18  JMP LOOP
         EXIT
0x1BF0    19  LOA R15 ZERO
0xF       1A  TRP
```

```
0         1B  ZERO          --- this is essentially the beginning of the
1         1C  ONE           --- data segment!
2         1D  TWO           --- 1D (=29 in decimal) is last address
```

- Remarks about this code:
  - "0x" is the standard C convention for entering hexadecimal numbers. Note that, when convenient, we also enter decimal numbers!
  - Note our convention of making the code human-readable
  - Note the use of data segment
  - Can you see why we use 33 words for this process?
- The operation of the Basic STM interpreter is this:
  - It reads a stml file and loads into memory (sequentially).
  - All addresses are **relative**: the physical address is equal to the base register plus the relative address.
  - It starts execution of the program starting at relative address 0.
- Errors:
  - Arithmetic errors: overflow, or divide by 0.
  - Address errors: address not in partition
  - PC increment error: Incrementing PC goes outside the partition
  - I/O error: non-integer in input, or reading past End-of-File.
- CAVEATS: the STM is unrealistic for many reasons.
  - The OS is not clearly seperated from the "hardware".
  - The "hardware" does not model I/O devices which requires its own blocking and interrupts.

  Nevertheless, it is useful as a teaching tool, and some of these shortcomings can be overcome by further elaboration.