CMSC 313 Lecture 14

• Reminder: Midterm Exam next Tues (10/26)
• Project 4 Questions
• Virtual Memory on Linux/Pentium platform
Project 4: An Error-Correcting Code, Part 2       Due: Thursday October 21, 2004

Objective
The objective of this programming exercise is to practice writing assembly language programs that use the C function call conventions.

Assignment
Modify your assembly language program from Project 3 so that it can be called from a C program as a C function with the following function prototype:

    char *encode(char *A, int n, int *mptr) ;

Here A is a pointer to a sequence of bytes in memory that should be encoded into the Hamming Code format from Project 3. The parameter n is the number of bytes in A. The result of the codewords must be stored in a memory location that is dynamically allocated. Your assembly language program must call malloc() to obtain a block of memory of the correct size. The address of this block of memory is the return value from encode(). The size of the block must be stored in the location specified by mptr.

Your program must work with the C main program p4main.c which is available in the following directory in the GL file system:

    /afs/umbc.edu/users/c/h/chang/pub/cs313

This C program reads bytes from stdin and stores them in a dynamically allocated memory location. It then calls your encode() function and writes the resulting codewords to stdout. Thus, if your assembly language implementation of encode() works correctly, the program resulting from compiling it with p4main.c behaves exactly like the encoding program in Project 3.

If you cannot convert your assembly language program from Project 3 (e.g., if your program for Project 3 does not work), then you must implement a similar encode() function that copies the bytes in A, but inserts the byte 0xFF after every three bytes. It must also pad the resulting memory block with 0xFF so the total length is divisible by 4. Implementing this version of the project will incur a 10% penalty.

Implementation Notes
- Look up malloc() if you haven't used it in a while. Remember that calling malloc() from your assembly language program will clobber the EAX, ECX and EDX registers.
- You should check for the possibility that malloc() returns 0 because the system does not have as much memory as you have requested.
- You will most likely need to use the EBX, ESI and EDI registers. You should push them on the stack to save them. Remember that when you pop them off the stack, you must pop them off in the opposite order.
- You will need to pre-compute the size of the memory block that holds the resulting codewords. Look up the DIV instruction. Note that it does not support many addressing modes.
- Your program should not alter the sequence of bytes given to you. This might change the way you handle the "extra bytes" at the end.
- As in Project 3, you can use decode, corrupt and diff to check if your program produced the correct results.
Turning in your program

Use the UNIX submit command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is cs313_0101. The name of the assignment name is proj4. The UNIX command to do this should look something like:

submit cs313_0101 proj4 p4encode.asm typescript
Last Time: Virtual Memory

- **Not enough physical memory**
  - Uses disk space to simulate extra memory
  - Pages not being used can be swapped out (how and when you’ll learn in CMSC 421 Operating Systems)
  - Thrashing: pages constantly written to and retrieved from disk (time to buy more RAM)

- **Fragmentation**
  - Contiguous blocks of virtual memory do not have to map to contiguous sections of real memory

- **Memory protection**
  - Each process has its own page table
  - Shared pages are read-only
  - User processes cannot alter the page table (must be supervisor)
Virtual Addressing

Page faults are costly and take millions of cycles to process (disks are slow)

80386 Page attributes:
- **RW**: read and write permission
- **US**: User mode or kernel mode only access
- **PP**: present bit to indicate where the page is
**Page Table**

- **Hardware supported**

**Page table:**
- Resides in main memory
- One entry per virtual page
- No tag is required since it covers all virtual pages
- Point directly to physical page
- Table can be very large
- Operating sys. may maintain one page table per process
- A dirty bit is used to track modified pages for copy back

Indicates whether the virtual page is in main memory or not
Linux 2-Level Page Table

- The CR3 register is designated for pointing to the first level page table.
- The CR3 is part of the task state that needs to be saved at preemption.

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index into Page Table Table</td>
<td>Index into Page Table</td>
<td>Index into Page</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.7.1. Linear Address Translation (4-KByte Pages)

Figure 3-12 shows the page directory and page-table hierarchy when mapping linear addresses to 4-KByte pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to $2^{20}$ pages, which spans a linear address space of $2^{32}$ bytes (4 GBytes).

![Figure 3-12. Linear Address Translation (4-KByte Pages)](image)

To select the various table entries, the linear address is divided into three sections:

- Page-directory entry—Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table.
PROTECTED-MODE MEMORY MANAGEMENT

Page-directory entries when 4-MByte pages and 32-bit physical addresses are being used. The functions of the flags and fields in the entries in Figures 3-14 and 3-15 are as follows:

**Page base address, bits 12 through 32**

(Page-table entries for 4-KByte pages.) Specifies the physical address of the first byte of a 4-KByte page. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces pages to be aligned on 4-KByte boundaries.

![Diagram of Page-Directory Entry (4-KByte Page Table)](image)

![Diagram of Page-Table Entry (4-KByte Page)](image)

**Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses**

(Page-directory entries for 4-KByte page tables.) Specifies the physical address of the first byte of a page table. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces page tables to be aligned on 4-KByte boundaries.
Segmentation

- A segmented memory allows two users to share the same word processor code, with different data spaces:

  - Segment #0: Execute only
  - Segment #1: Read/write by user #0
  - Segment #2: Read/write by user #1

  - Address space for code segment of word processor
  - Data space for user #0
  - Data space for user #1
  - Unused
## Translation Lookaside Buffer

- An example TLB holds 8 entries for a system with 32 virtual pages and 16 page frames.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page number</th>
<th>Physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01001</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>01110</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>--------</td>
<td>--------</td>
</tr>
</tbody>
</table>
Virtual Memory: Problems Solved

• Not enough physical memory
  ◦ Uses disk space to simulate extra memory
  ◦ Pages not being used can be swapped out
    (how and when you’ll learn in CMSC 421 Operating Systems)
  ◦ Thrashing: pages constantly written to and retrieved from disk
    (time to buy more RAM)

• Fragmentation
  ◦ Contiguous blocks of virtual memory do not have to map to
    contiguous sections of real memory

• Memory protection
  ◦ Each process has its own page table
  ◦ Shared pages are read-only
  ◦ User processes cannot alter the page table (must be supervisor)
Virtual Memory: too slow?

- **Address translation is done in hardware**
  
  In the middle of the fetch execute cycle for:
  
  ```
  MOV EAX, [buffer]
  ```
  
  the physical address of buffer is computed in hardware.

- **Recently computed page locations are cached in the translation lookaside buffer (TLB)**

- **Page faults are very expensive (millions of cycles)**

- **Operating systems for personal computers have only recently added memory protection**
• Interrupts

• Cache Memory