CMSC 313 Lecture 13

- Project 4 Questions
- Reminder: Midterm Exam next Thursday 10/16
- Virtual Memory
Project 4: C Functions

Due:  Tue 10/14/03,  Section 0101 (Chang) & Section 0301 (Macneil)
      Wed 10/15/03,  Section 0201 (Patel & Bourner)

Objective

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

Assignment

Convert your assembly language program from Project 3 as follows:

1. Convert the program into one that follows the C function call convention, so it may be called from a C program. Your program should work with the following function prototype:

   ```c
   void report (void *, unsigned int) ;
   ```

   The intention here is that the first parameter is a pointer to the records array and the second parameter has the number of items in that array.

2. Modify your program so it uses the strncmp() function from the C library to compare the nicknames of two records. The function prototype of `strncmp()` is:

   ```c
   int strncmp(const char *s1, const char *s2, size_t n) ;
   ```

   The function returns an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`.

3. Modify your program so that it prints out the entire record (not just the `realname` field) of the record with the least number of points and the record with the alphabetically first nickname. You must use the `printf()` function from the C library to produce this output. The output of your program would look something like:

   ```plaintext
   Lowest Points: James Pressman (jamieboy)
   Alignment: Lawful Neutral
   Role: Fighter
   Points: 57
   Level: 1
   First Nickname: Dan Gannett (danmeister)
   Alignment: True Neutral
   Role: Ranger
   Points: 7502
   Level: 3
   ```

A sample C program that should work with your assembly language implementation of the `report()` function is available on the GL file system: `/afs/umbc.edu/users/c/h/chang/pub/cs313/records2.c`

Implementation Notes

- Documentation for the `printf()` and `strncmp()` functions are available on the Unix system by typing `man -S 3 printf` and `man -S 3 strncmp`.
- Note that the `strncmp()` function takes 3 parameters, not 2. It is good programming practice to use `strncmp()` instead of `strcmp()` since this prevents runaway loops if the strings are not properly null terminated. The third argument should be 16, the length of the `nickname` field.
- As in Project 3, you must also make your own test cases. The example in `records2.c` does not fully exercise your program. As before, your program will be graded based upon other test cases. If you have good examples in Project 3, you can just reuse those.

- Use `gcc` to link and load your assembly language program with the C program. This way, `gcc` will call `ld` with the appropriate options:

  - `nasm -f elf report2.asm`
  - `gcc records2.c report2.o`

- Notes on the C function call conventions are available on the web:

  - `http://www.csee.umbc.edu/~chang/cs313.f03/stack.shtml`

- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.

**Turning in your program**

Use the UNIX `submit` command on the GL system to turn in your project. You should submit at least 4 files: your assembly language program, at least 2 of your own test cases and a typescript file of sample runs of your program. The class name for submit is `cs313_0101`, `cs313_0102` or `cs313_0103` for respectively sections 0101 (Chang), 0201 (Patel & Bourner) or 0301 (Macneil). The name of the assignment name is `proj4`. The UNIX command to do this should look something like:

  - `submit cs313_0103 proj4 report2.asm myrec1.c myrec2.c typescript`
Last Time

- Linux/gcc/i386 Function Call Convention
- Now we know where our C programs store their data, right???

```c
int global;

int main() {
    int *ptr, n;

    printf("Address of main: %08x\n", &main);
    printf("Address of global variable: %08x\n", &global);
    printf("Address of local variable: %08x\n", &n);

    ptr = (int *) malloc(4);
    printf("Address of allocated memory: %08x\n", ptr);
}
```
Linux Virtual Memory Space

- Linux reserves 1 Gig memory in the virtual address space.
- The size of the Linux kernel significantly affects its performance (swapping is expensive).
- Linux kernel can be customized by including only relevant modules.
- Designating kernel space facilitates protection of.
- The portion of disk used for paging is called the swap space.

```
Linux Kernel

3 Gig

4 Gig

0

Paging System

Task #2

Task #3

Task #n

3 Gig

0

3 Gig

0

3 Gig

0

3 Gig
```

RAM

Disk
The Memory Hierarchy

- Registers
- Cache
- Main memory
- Secondary storage (disks)
- Off-line storage (tape)

Increasing performance and increasing cost

Fast and expensive

Slow and inexpensive
Overlays

- A partition graph for a program with a main routine and three sub-routines:

```
Compiled program          Physical Memory

Main Routine
Subroutine A
Subroutine B
Subroutine C

Smaller than program
```

```
Partition graph
Main -> A
Main -> C
A -> B
B -> C

Partition #0: A
Partition #1: C
```

Principles of Computer Architecture by M. Murdocca and V. Heuring © 1999 M. Murdocca and V. Heuring
Fragmentation

- (a) Free area of memory after initialization; (b) after fragmentation; (c) after coalescing.
Memory Protection

- Prevents one process from reading from or writing to memory used by another process
- Privacy in a multiple user environments
- Operating system stability
  - Prevents user processes (applications) from altering memory used by the operating system
  - One application crashing does not cause the entire OS to crash
Virtual Memory

- Virtual memory is stored in a hard disk image. The physical memory holds a small number of virtual *pages* in physical *page frames*.

- A mapping between a virtual and a physical memory:

```
Virtual addresses  Virtual memory
0 - 1023          Page 0
1024 - 2047       Page 1
2048 - 3071       Page 2
3072 - 4095       Page 3
4096 - 5119       Page 4
5120 - 6143       Page 5
6144 - 7167       Page 6
7168 - 8191       Page 7
```

```
Physical memory
Page frame 0
Page frame 1
Page frame 2
Page frame 3

Physical addresses
0 - 1023
1024 - 2047
2048 - 3071
3072 - 4095
```
Page Table

- The page table maps between virtual memory and physical memory.

<table>
<thead>
<tr>
<th>Page #</th>
<th>Present bit</th>
<th>Disk address</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010010111100</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111011110010</td>
<td>xx</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>101100101111</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>000010011111</td>
<td>xx</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>010111001011</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>101001110011</td>
<td>xx</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>001101011100</td>
<td>xx</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>010100010111</td>
<td>10</td>
</tr>
</tbody>
</table>

Present bit:
0: Page is not in physical memory
1: Page is in physical memory
Using the Page Table

- A virtual address is translated into a physical address:

```
Page  Offset
1 0 1 0 1 0 0 0 1 0 1  

<table>
<thead>
<tr>
<th>Page</th>
<th>Offset</th>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>010010111100</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>11101110010</td>
<td>xx</td>
</tr>
<tr>
<td>1</td>
<td>101100101111</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>000010011111</td>
<td>xx</td>
</tr>
<tr>
<td>1</td>
<td>010111001011</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>10100111001</td>
<td>xx</td>
</tr>
<tr>
<td>1</td>
<td>001101011100</td>
<td>xx</td>
</tr>
<tr>
<td>1</td>
<td>010100010111</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Page table

Physical address
Using the Page Table (cont’)

- The configuration of a page table changes as a program executes.
- Initially, the page table is empty. In the final configuration, four pages are in physical memory.

<table>
<thead>
<tr>
<th>Page</th>
<th>Bits</th>
<th>After fault on page #1</th>
<th>After fault on page #2</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01001011100 xx</td>
<td>01001011100 xx</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101110010 00</td>
<td>11101110010 00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>10110010111 xx</td>
<td>10110010111 xx</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>00001001111 xx</td>
<td>00001001111 xx</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>01011100101 xx</td>
<td>01011100101 xx</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10100111001 xx</td>
<td>10100111001 xx</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00110101100 xx</td>
<td>00110101100 xx</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>01010001011 xx</td>
<td>01010001011 xx</td>
<td></td>
</tr>
</tbody>
</table>

After fault on page #1:

- Page 1 becomes 01101110010
- Page 3 becomes 00001001111
- Page 4 becomes 01010001011
- Page 6 becomes 00110101100

After fault on page #2:

- Page 2 becomes 10110010111
- Page 3 becomes 00001001111
- Page 5 becomes 10100111001
- Page 7 becomes 01010001011

Final configuration:

- Page 1: 01101110010
- Page 2: 10110010111
- Page 3: 00001001111
- Page 4: 01011100101
- Page 5: 10100111001
- Page 6: 00110101100
- Page 7: 01010001011
Virtual Addressing

Virtual address

31 30 29 28 27 ............... 15 14 13 12 11 10 9 8 ........... 3 2 1 0

Virtual page number

Page offset

Translation

Physical page number

Page offset

Physical address

- 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 system 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
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.
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.

---

**Page-Directory Entry (4-KByte Page Table)**

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Page-Table Base Address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Available for system programmer's use</td>
</tr>
</tbody>
</table>

**Page-Table Entry (4-KByte Page)**

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Page Base Address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Available for system programmer's use</td>
</tr>
</tbody>
</table>

---

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

Used
Used
Free
Used
Free
Unused

Data space for user #0

Data space for user #1
```
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**

UMBC, CMSC313, Richard Chang <chang@umbc.edu>
Next Time

- Memory Cache
- Interrupts
- Review for Midterm Exam