CSE 351
SECTION 10
THE END...ALMOST
3/7/12
Agenda

• Virtual Memory
• Final Review
  – Assembly
  – Calling Conventions
  – Malloc/Free
  – Caching
• Questions, Evaluations
Virtual Memory

- Used for 3 things
  - Efficient use of main memory (RAM)
    - Use RAM as cache for parts of virtual address space
      - Some non-cache parts stored to disk
      - Some (unallocated) non-cached parts stored nowhere
    - Keep only active areas of virtual address space in memory
      - Transfer data back and forth as needed
  - Memory management
    - Each process gets the same full, private linear address space
  - Memory protection
    - Isolates address spaces
    - One process can’t interfere with another’s memory since they operate in different address spaces
    - User process cannot access privileged information
      - Different sections of address spaces have different permissions
Address Spaces

• **Virtual address space**: Set of \( N = 2^n \) virtual addresses
  \( \{0, 1, 2, 3, ..., N-1\} \)

• **Physical address space**: Set of \( M = 2^m \) physical addresses ( \( n >> m \) )
  \( \{0, 1, 2, 3, ..., M-1\} \)

• Every byte in main memory:
  one physical address, one (or more) virtual addresses
VM as a Tool for Caching

• *Virtual memory*: array of \( N = 2^n \) contiguous bytes
  - think of the array (allocated part) as being stored on disk
• Physical main memory (DRAM) = cache for allocated virtual memory
• Blocks are called pages; size = \( 2^p \)

![Diagram showing virtual memory and physical memory with caching]

- Virtual pages (VP's) stored on disk
- Physical pages (PP's) cached in DRAM
• Each process gets its own private memory space
Address Translation: Page Tables

- A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs
VM as a Tool for Memory Management

- Memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV signal (segmentation fault)
  - SUP bit indicates whether processes must be running in kernel (supervisor) mode to access it

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
FINAL REVIEW
Assembly – Things to Remember

• `.text` always goes before your code
• `.globl <label>` when you want your function to be used by other modules (i.e. public)
• `pushq %rbp` and `movq %rsp, %rbp` when entering a function
• `popq %rbp` and `ret` at the end of your function
• Size suffixes must be used when the length can not be implicitly determined
  – To be safe, always use them! (e.g. `movq`, `cmpb`, etc.)
• If you need to allocate stack space to store data, the space must be a multiple of 16.
  – E.g. `sub $32, %rsp` at the start, then `add $32, %rsp` at the end of the function
• Register names used must match size suffix of instruction
  – E.g. To use the lower byte stored in rax with `cmpb`, you must use `%al, not %rax.
• Dereferencing
  – `cmpb (%rdi), %sil`
    • Compares 1 byte in memory stored at the address in rdi with the lower byte in the rsi register
Assembly – More Things

• Read only data – data that will not change
  
  .section .rodata

  mystring:
  
  .string “Hello world”

  – Access the pointer to the start of the string using
    $mystring

• Labels really act like pointers to instructions or data
  
  – jmp loop is really saying the next instruction lives at
    the address where the loop label points to

• Data segment
  
  .data

  my_array: .zero 512

  – Allocates 512 bytes for my_array and initializes to zero
x86-64 Calling Conventions

• First six arguments passed in registers
  – rdi, rsi, rdx, rcx, r8, r9

• Callee saved registers
  – rbx, rbp, r12, r13, r14, r15
  – Function being called must save the values in the registers before using them, and restore them before returning.

• Caller saved registers
  – r10, r11
  – Calling function must save these registers if it wants to keep the values in them

• Return value stored in rax
Malloc/Free

- **Use `malloc`** when you want to dynamically allocate something
  - e.g. the size of a data structure is only known at runtime
  - Data allocated on the heap
    ```c
    p = (int*) malloc(n*sizeof(int));
    ```
- Data allocated with `malloc` must be `free`'d when finished with it
  ```c
  free(p);
  ```
Caching

- Exploits temporal and spatial locality
  - Temporal locality: recently referenced items likely to be referenced again in the near future
  - Spatial locality: items with nearby addresses tend to be referenced close together in time
- Organized into lines and sets
- Number of lines per set is the associativity
  - E.g. 2-way associative means 2 lines per set
- Line consists of valid bit, tag, data block
Caching

$E = 2^e$ lines per set

Address of word:
- tag
- set index
- block offset

data begins at this offset

valid bit

$B = 2^b$ bytes data block per cache line (the data)
Suggestions
(Not a comprehensive list!)

• Review all lecture and section slides
• Be able to write both assembly and C code to the level we’ve covered
  – Practice writing code at home. Pick some functionality (like perhaps atoi) and code it in both C and assembly.
  – All code you write on the final should be able to be compiled
  – Have a solid understanding of pointers
  – Have a solid understanding of how the stack works
• Be able to convert a C function into assembly and vice versa
• Understand data representation (2’s complement, endianness, signed/unsigned, floating point, etc.)
• Know the x64 calling conventions
QUESTIONS? COMMENTS?

GOOD LUCK ON THE FINAL AND THANKS FOR A GREAT QUARTER!