University of Washington – Computer Science & Engineering
Winter 2017    Instructor: Justin Hsia       2017-03-14

CSE410 FINAL

Last Name:  
First Name:  
Student ID Number:  

Name of person to your Left | Right

All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE410 who haven’t taken it yet.  (please sign)

Do not turn the page until 2:30.

Instructions

- This exam contains 12 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed two pages (US letter, double-sided) of handwritten notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Question</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible Points</td>
<td>8</td>
<td>8</td>
<td>13</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>75</td>
</tr>
</tbody>
</table>
Question M1: Number Representation [8 pts]

(A) What is the decimal value of the float \texttt{0x3F400000}? [2 pt]

(B) Take the 32-bit numeral \texttt{0xC0800000}. Circle the number representation below that has the \textit{most negative} value for this numeral. [2 pt]

<table>
<thead>
<tr>
<th>Floating Point</th>
<th>Sign &amp; Magnitude</th>
<th>Two’s Complement</th>
<th>Unsigned</th>
</tr>
</thead>
</table>

(C) Let float \texttt{x=2^{100}}. What is the smallest positive value of float \texttt{y} such that \texttt{x*y} results in \textit{overflow}? You may answer using powers of 2. [2 pt]

(D) I have an arbitrary address stored in \texttt{int* p} and I wish to \textit{align} it to 4 bytes (i.e. round the address to a \texttt{nearby} multiple of 4). Complete the C code below to accomplish this task: [2 pt]

\begin{verbatim}
p = p ______________________________; // aligns p to 4 bytes
\end{verbatim}
Question M2: Pointers & Memory  [8 pts]

For this problem we are using a 64-bit x86-64 machine (little endian). Below is the power function disassembly, showing where the code is stored in memory.

```
00000000004005a0 <power>:
  4005a0:  85 f6           testl %esi,%esi
  4005a2:  74 10           je 4005b4 <power+0x14>
  4005a4:  53              pushq %rbx
  4005a5:  89 fb           movl %edi,%ebx
  4005a7:  83 ee 01         subl $0x1,%esi
  4005aa:  e8 f1 ff ff ff   call 4005a0 <power>
  4005af:  0f af c3         imull %ebx,%eax
  4005b2:  eb 06           jmp 4005ba <power+0x1a>
  4005b4:  b8 01 00 00 00   movl $0x1,%eax
  4005b9:  c3              ret
  4005ba:  5b              popq %rbx
  4005bb:  c3              ret
```

(A) What are the data (in hex) stored in each register shown after the following x86 instructions are executed? Use the appropriate bit widths. Hint: what is the value stored in %rsi? [4 pt]

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x0000 0000 0040 05AF</td>
</tr>
<tr>
<td>%rsi</td>
<td>0xFFFF FFFF FFFF FFFF</td>
</tr>
</tbody>
</table>
```

leal 2(%rdi, %rsi), %eax

movw (%rdi,%rsi,4), %bx

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let char* cp = 0x4005A4. [4 pt]

```
char v1 = *(cp + _____);  // set v1 = 0xEE
int* v2 = (int*)((___________*)cp - 2); // set v2 = 0x4005A0
```
Question M3: Procedures & The Stack  [13 pts]

The recursive function `array_sum()` and its x86-64 disassembly are shown below:

```c
int array_sum (int ar[], int num) {
    if (num > 0)
        return ar[0] + array_sum(ar+1,num-1);
    else
        return 0;
}
```

00000000000400538 <array_sum>:

```assembly
    b8 00 00 00 00 mov $0x0,%eax
    85 f6 test %esi,%esi
    7e 13 jle 400554 <array_sum+0x1c>
    53 push %rbx
    48 89 fb mov %rdi,%rbx
    83 ee 01 sub $0x1,%esi
    48 8d 7f 04 lea 0x4(%rdi),%rdi
    e8 e7 ff ff ff callq 400538 <array_sum>
    03 03 add (%rbx),%eax
    5b pop %rbx
    f3 c3 rep ret
```

(A) Name one difference between the disassembly shown above and the version of the `array_sum` function that would appear in the object file `array_sum.o`. [2 pt]

(B) Why is `%rbx` being pushed onto the stack? What is `%rbx` being used for in this function? [2 pt]
(C) What is the return address to `array_sum` that gets stored on the stack? Answer in hex. [1 pt]

(D) Provide an example call to `array_sum(ar, num)` that will cause a segmentation fault. [1 pt]

array_sum(______, ____)

(E) Calling `array_sum(a, 4)` with `int a[] = {1, 2, 3, 4}`: What is the maximum amount of memory on the stack (in bytes) used for `array_sum` stack frames at any given time? [3 pt]

(F) Below is an incomplete snapshot of the stack during the call to `array_sum(a, 4)`. Assume that `&a = 0x1000`. Fill in the values of the four missing intermediate words in hex. [4 pt]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffc39b72e8</td>
<td>&lt;ret addr to main&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72e0</td>
<td>&lt;original rbx&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72d8</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72d0</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72c8</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72c0</td>
<td></td>
</tr>
</tbody>
</table>
Question M4: C & Assembly  [9 pts]

We are writing the *recursive* function `LL_search`, which returns 1 if a specified `int val` is found in a linked list, or 0 if it is not. The nodes of the linked list are defined in the struct below:

```c
typedef struct LL_node {
    int val;
    struct LL_node* next;
} node;

int LL_search(node* n, int val) {
    if (!n)
        return 0;
    else if (n->val == val)
        return 1;
    return LL_search(n->next, val);
}
```

Fill in the blanks in the x86-64 code below with the correct instructions and operands. *Remember to use the proper size suffixes and correctly-sized register names!*

```assembly
LL_search(n*, int):
1  _____ _____, %rdi     # conditional
2  _____ .NotFound        # conditional jump
3  _____ %esi, _____      # conditional
4  _____ _____           # conditional jump
5  _____ _____, _____     # argument setup
6  _____ _____           # recurse
7  ret
   .NotFound:
8  _____ $0, %eax         # return value
9  ret
   .Found:
10 movq _____, _____      # return value
11 ret
```
**Question F5: Caching** [10 pts]

We have 32 KiB of RAM and a 1-KiB L1 data cache that is direct mapped with 16 B blocks and a write-back policy.

(A) Calculate the TIO address breakdown: [1.5 pt]

<table>
<thead>
<tr>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
</tr>
</thead>
</table>

(B) How many management bits (bits other than the block data) are there in every line in the cache? [1 pt]

(C) The code snippet below converts *every fourth* character in an array to uppercase. Calculate the **Miss Rate** for the L1 data cache if it starts *cold*. [3 pt]

```c
#define ARRAY_SIZE 2048
char string[ARRAY_SIZE];         // &string = 0x1000 (physical addr)
for (i = 0; i < ARRAY_SIZE; i += 4 )
    string[i] &= ~(0x20);          // convert char to uppercase
```

(D) For each of the proposed (independent) changes, write **U** for “increased”, **N** for “no change”, or **D** for “decreased” to indicate the effect on the **Miss Rate** for the code above: [3.5 pt]

- 2-way set associativity _____
- Decrease block size _____
- Half ARRAY_SIZE _____
- Add a L2 cache _____

(E) Assume it takes 100 ns to get a block of data from main memory. If our L1 data cache has a hit time of 4 ns and a miss rate of 4%, what is the average memory access time? [1 pt]
Question F6: Processes [9 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

```
void concurrent (void) {
    int x = 4, status;
    if (fork() == 0) {
        if (fork())
            printf("2");
        else {
            printf("%d", x*=2);
            printf("%d", --x);
        }
        exit(0);
    }
    wait(&status);
    wait(&status);
    printf("%d", ++x);
}
```

(1) ____________
(2) ____________
(3) ____________

(B) For the following examples of exception causes, write “N” for intentional or “U” for unintentional from the perspective of the user. [2 pt]

<table>
<thead>
<tr>
<th>Exception</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>System call</td>
<td>______</td>
</tr>
<tr>
<td>Network packet received</td>
<td>______</td>
</tr>
<tr>
<td>Page fault</td>
<td>______</td>
</tr>
<tr>
<td>Breakpoint reached</td>
<td>______</td>
</tr>
</tbody>
</table>

(C) In the following blanks, write “Y” for yes or “N” for no if the following need to be saved on a context switch. [2 pt]

<table>
<thead>
<tr>
<th>Component</th>
<th>Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition codes</td>
<td>_____</td>
</tr>
<tr>
<td>Program counter</td>
<td>_____</td>
</tr>
<tr>
<td>Static data</td>
<td>_____</td>
</tr>
<tr>
<td>Dirty cache blocks</td>
<td>_____</td>
</tr>
</tbody>
</table>

(D) Name a section of memory that does not need to be duplicated when we fork a process. What technique allows us to skip this duplication? [2 pt]
Question F7: Virtual Memory  [10 pts]

Our system has the following setup:

- 16-bit virtual addresses and 4 KiB of RAM with page size of 256 B
- A 4-entry TLB that is 2-way set associative with LRU replacement
- Page table entries containing bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values: [2 pt]

- Page offset width 
- PPN width 
- Entries in a page table 
- TLBI width 

(B) The current state of our system is shown in the following tables. Fill out the table below to indicate the result of the two independent memory requests. Use “n/a” if a particular piece of information cannot be determined. [8 pt]

<table>
<thead>
<tr>
<th>Set</th>
<th>TLBT</th>
<th>PPN</th>
<th>V</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x6F</td>
<td>0xB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x5E</td>
<td>0x3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x7F</td>
<td>0x1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x11</td>
<td>0x0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>V</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x11</td>
<td>0x4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7F</td>
<td>0xF</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0xDE</td>
<td>0xB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Request</th>
<th>Physical Address</th>
<th>TLB Hit?</th>
<th>Page Fault?</th>
<th>Protection Fault?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>0x1110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>0xDEAD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Question F8: Operating Systems** [8 pts]

Respond to the following short answer questions *concisely* and within the provided boxes. These should not require more than two sentences each.

(A) When deciding on the structure of an operating system, what is the major design decision to be made for each of the individual modules and what are the two options? [2 pt]

(B) The state of a process can be Ready, Running, Blocked, or Finished. What happens in the Blocked state? Give an example of a reason a process would enter the Blocked state. [2 pt]

(C) What is the difference between a process control block (PCB) and a thread control block (TCB)? Give an example of a field that would be different or missing in a TCB compared to a PCB. [2 pt]

(D) What is the main information stored in a directory file? [2 pt]
Assembly Instructions

- **mov a, b**  
  Copy from a to b.

- **movs a, b**  
  Copy from a to b with sign extension. Needs two width specifiers.

- **movz a, b**  
  Copy from a to b with zero extension. Needs two width specifiers.

- **lea a, b**  
  Compute address and store in b.  
  *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.

- **push src**  
  Push src onto the stack and decrement stack pointer.

- **pop dst**  
  Pop from the stack into dst and increment stack pointer.

- **call <func>**  
  Push return address onto stack and jump to a procedure.

- **ret**  
  Pop return address and jump there.

- **add a, b**  
  Add from a to b and store in b (and sets flags).

- **imul a, b**  
  Multiply a and b and store in b (and sets flags).

- **and a, b**  
  Bitwise AND of a and b, store in b (and sets flags).

- **sar a, b**  
  Shift value of b right (*arithmetic*) by a bits, store in b (and sets flags).

- **shr a, b**  
  Shift value of b right (*logical*) by a bits, store in b (and sets flags).

- **shl a, b**  
  Shift value of b left by a bits, store in b (and sets flags).

- **cmp a, b**  
  Compare b with a (compute b-a and set condition codes based on result).

- **test a, b**  
  Bitwise AND of a and b and set condition codes based on result.

- **jmp <label>**  
  Unconditional jump to address.

- **j* <label>**  
  Conditional jump based on condition codes (*more on next page*).

- **set* a**  
  Set byte based on condition codes.
### Conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>(op) s, d</th>
<th>test a, b</th>
<th>cmp a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>d (op) s == 0</td>
<td>b &amp; a == 0</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>d (op) s != 0</td>
<td>b &amp; a != 0</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>d (op) s &lt; 0</td>
<td>b &amp; a &lt; 0</td>
<td>b-a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>d (op) s &gt;= 0</td>
<td>b &amp; a &gt;= 0</td>
<td>b-a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>d (op) s &gt; 0</td>
<td>b &amp; a &gt; 0</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jge</td>
<td>d (op) s &gt;= 0</td>
<td>b &amp; a &gt;= 0</td>
<td>b &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>d (op) s &lt; 0</td>
<td>b &amp; a &lt; 0</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>d (op) s &lt;= 0</td>
<td>b &amp; a &lt;= 0</td>
<td>b &lt;= a</td>
</tr>
<tr>
<td>ja</td>
<td>d (op) s &gt; 0U</td>
<td>b &amp; a &lt; 0U</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jb</td>
<td>d (op) s &lt; 0U</td>
<td>b &amp; a &gt; 0U</td>
<td>b &lt; a</td>
</tr>
</tbody>
</table>

### Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Convention</th>
<th>Name of “virtual” register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Caller saved</td>
<td>%eax %ax %al</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td>%ebx %bx %bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4 – Caller saved</td>
<td>%ecx %cx %cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3 – Caller saved</td>
<td>%edx %dx %dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2 – Caller saved</td>
<td>%esi %si %sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1 – Caller saved</td>
<td>%edi %di %dil</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
<td>%esp %sp %Spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td>%ebp %bp %bpl</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5 – Caller saved</td>
<td>%r8d %r8w %r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 – Caller saved</td>
<td>%r9d %r9w %r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
<td>%r10d %r10w %r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller saved</td>
<td>%r11d %r11w %r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
<td>%r12d %r12w %r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
<td>%r13d %r13w %r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
<td>%r14d %r14w %r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
<td>%r15d %r15w %r15b</td>
</tr>
</tbody>
</table>

### C Functions

- **char* gets(char* s):** Reads a line from stdin into the buffer.
- **pid_t fork():** Create a new child process (duplicates parent).
- **pid_t wait(int* status):** Blocks calling process until any child process exits.
- **void exit(int status):** Terminates a process.
- **int execv(char* path, char* argv[]):** Replace current process image with new image.

### Sizes

<table>
<thead>
<tr>
<th>C type</th>
<th>x86-64 suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>