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 0x3F400000? [2 pt]

\[
S = 0, \ E = 0111\ 1110_2, \ M = 10...0_2. \ E \text{ is not a special case.} \\
\text{Exp} = ((2^7 - 1) - 1) - 127 = -1, \ \text{Man} = 1.1_2. \\
(-1)^0 \times 1.1_2 \times 2^{-1} = +0.11_2 = 2^{-1} + 2^{-2} = 0.75.
\]

(B) Take the 32-bit numeral 0xC0800000. Circle the number representation below that has the 
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>
<tbody>
<tr>
<td><strong>Unsigned:</strong></td>
<td>Can only represent positive numbers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating Point:</strong></td>
<td>S = 1 and E = 10000001_2 → Exp = 2, so a small negative number.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sign &amp; Mag:</strong></td>
<td>Negative number with magnitude 100 0000 10...0_2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Two’s:</strong></td>
<td>Negative number with magnitude 011 1111 10...0_2 (flip bits + 1).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
2^{28}
\]

In floating point, overflow occurs when your result is larger in magnitude than the largest normalized number we can represent. That occurs when S = 0, E = 1111 1110_2, M = 1...1_2, which has value just below 2^{128} (2^{128}-2^{104}, to be exact). So \(x \times 2^{28} = 2^{128}\), which results in overflow.

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

\[
p = p \_&\_(_{-3})..........................................................; \ // \text{aligns p to 4 bytes}
\]

Multiples of 4 in binary must have the lowest two bits be zeros, so our goal is to zero out the lowest two bits of our address stored in p. The following answers were also accepted:

\[
p = p >> 2 << 2; \\
p = p \& (-1 << 2); \\
p = p \& -4; \\
p = p \& 0xffffffff ffff ffff fffc;
\]
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 0400 05AF</td>
</tr>
<tr>
<td>%rsi</td>
<td>0xFFFF FFFF FFFF FFFF</td>
</tr>
<tr>
<td>%eax</td>
<td>0x0040 05B0</td>
</tr>
<tr>
<td>%bx</td>
<td>0xFFF1</td>
</tr>
</tbody>
</table>

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

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

*leal* instruction calculates the address 0x4005AF + (-1) + 2 = 0x4005B0.

*movw* instruction pulls two bytes starting at memory address 0x4005AF+4*(-1) = 0x4005AB, which is 0xF1 and 0xFF. Remember little-endian!

(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 + __4__); // set v1 = 0xEE
int* v2 = (int*)((__short__*)cp - 2); // set v2 = 0x4005A0
```

The only 0xEE byte in power is found at address 0x4005A8, 4 bytes beyond cp.

The difference between v2 and cp is 4 bytes. Since by pointer arithmetic we are moving 2 “things” away, cp must be cast to a pointer to a data type of size 2 bytes.
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;
}
```

```assembly
000000000000400538 <array_sum>:
400538:  b8 00 00 00 00      mov   $0x0,%eax
40053d:  85 f6            test  %esi,%esi
40053f:  7e 13            jle   400554 <array_sum+0x1c>
400541:  53              push  %rbx
400542:  48 89 fb         mov   %rdi,%rbx
400545:  83 ee 01         sub   $0x1,%esi
400548:  48 8d 7f 04      lea   0x4(%rdi),%rdi
40054c:  e8 e7 ff ff ff    callq 400538 <array_sum>
400551:  03 03           add   (%rbx),%eax
400553:  5b              pop   %rbx
400554:  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]

Instruction addresses would be at a much lower offset (close to zero) because the object file hasn’t been linked yet. Object files are binary, so no text.

Most credit given if mentioned the callq or jle instruction would be different. The encodings won’t be because of relative addressing, but the disassembly addresses will be.

(B) Why is %rbx being pushed onto the stack?  What is %rbx being used for in this function?  [2 pt]

(1) %rbx is being pushed onto the stack because it is a callee-saved register and array_sum uses this register.

(2) %rbx is being used to store the address of ar/ar[0].
(C) What is the return address to array_sum that gets stored on the stack? Answer in hex. [1 pt]

The address of the instruction after callq.

\[
0x400551
\]

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

array_sum( ___0___, 1___)

Had to have num>0 to avoid the base case. ar=0 (or 0x0) is the representation of the null pointer, which is guaranteed to segfault if you try to dereference it. Most credit given if you put an arbitrary array and num much greater than the size of the array – this is likely, but not guaranteed, to cause a segfault.

(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]

72 bytes

Stack frames during call: (a, 4) \(\rightarrow\) (a+1, 3) \(\rightarrow\) (a+2, 2) \(\rightarrow\) (a+3, 1) \(\rightarrow\) (a+4, 0)

From the assembly code, we know that %rbp gets pushed onto the stack every time array_sum is called except for the base case. The return address is also pushed for every procedure call and is part of the callee’s stack frame, so in total we have \(2+2+2+2+1\) words = \(9 \times 8\) bytes = 72 B.

(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>0x400551</td>
</tr>
<tr>
<td>0x7fffc39b72d0</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x7fffc39b72c8</td>
<td>0x400551</td>
</tr>
<tr>
<td>0x7fffc39b72c0</td>
<td>0x1004</td>
</tr>
</tbody>
</table>

Every other word will be a return address, so the data in the addresses 72d8 and 72c8 are the return address to array_sum (full credit given for your answer to part C). As asked for in part B, the address of ar is being pushed to the stack in each frame. With each recursive call, we increment a by 1, which means increasing the address by 4 bytes using pointer arithmetic.
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;
```

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    testq %rdi, %rdi       # conditional
2    je .NotFound          # conditional jump
3    cmpl %esi, (%rdi)    # conditional
4    je .Found             # conditional jump
5    movq 8(%rdi), %rdi   # argument setup
6    call LL_search        # recurse
7    ret                   
     .NotFound:
8    movl $0, %eax         # return value
9    ret                   
     .Found:
10   movq $1, %rax         # return value
11   ret                   
```

Grading Notes:
Line 1: cmpq $0, %rdi also accepted.
Line 3: 0(%rdi) also accepted.
Line 5: Partial credit for 4(%rdi) – not accounting for alignment/padding in the struct.
Line 6: callq also accepted
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>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

15 address bits. \( \log_2 16 = 4 \) offset bits. 1 KiB cache = 64 blocks. 1 blocks/set \( \rightarrow \) 64 sets.

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

Tag bits + Valid bit + Dirty bit (write-back) \( \boxed{7 \text{ bits}} \)

(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
```

Access pattern is read then write to \( \text{string}[i] \). Stride = 4 chars = 4 bytes. 16/4 = 4 strides per block. First access (read) is a compulsory miss and the next 7 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of \( \&\text{string} \) is 0x00, so we start at the beginning of a cache block.

(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 \( \underline{\text{N}} \)
- Decrease block size \( \underline{\text{U}} \)
- Half \( \text{ARRAY\_SIZE} \) \( \underline{\text{N}} \)
- Add a L2 cache \( \underline{\text{N}} \)

Since we never revisit blocks, associativity doesn’t matter. Smaller block size means fewer strides/block. Half \( \text{ARRAY\_SIZE} \) means fewer accesses, but same miss rate. L2$ has no effect.

(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]

\[
\text{AMAT} = \text{HT} + \text{MR} \times \text{MP} = 4 + 0.04 \times 100 = 8
\]

\(8 \text{ ns}\)
Question F6: Processes  [9 pts]

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

```c
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) _8, 7, 2, 5______

(2) _8, 2, 7, 5______

(3) _2, 8, 7, 5______

(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>Event</th>
<th>N</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>System call</td>
<td></td>
<td><strong>N</strong></td>
</tr>
<tr>
<td>Network packet received</td>
<td></td>
<td><strong>U</strong></td>
</tr>
<tr>
<td>Page fault</td>
<td><strong>U</strong></td>
<td></td>
</tr>
<tr>
<td>Breakpoint reached</td>
<td><strong>N</strong></td>
<td></td>
</tr>
</tbody>
</table>

Syscall and breakpoints are part of code you are executing. Page fault and interrupts are external.

(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>Condition codes</th>
<th>Y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog. counter</td>
<td></td>
<td><strong>Y</strong></td>
</tr>
<tr>
<td>Static data</td>
<td></td>
<td><strong>N</strong></td>
</tr>
<tr>
<td>Dirty cache blocks</td>
<td></td>
<td><strong>N</strong></td>
</tr>
</tbody>
</table>

Condition codes and program counter are part of the process state. Static Data and cached data can be left alone because they are part of physical memory.

(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]

Code/Instructions does not need to be duplicated because it can’t be changed (read-only), so both processes can read from it simultaneously without interfering with each other. Static Data also accepted, but more precisely, would be Literals (read-only).

Can skip duplication due to indirection of virtual memory.
**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: _8 bits_
- PPN width: _4 bits_
- Entries in a page table: _2^8 = 256_
- TLBI width: _1 bit_

There are $2^{\text{VPN width}}$ entries in PT.

(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>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>0x410</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Write</td>
<td>0xDEAD</td>
<td>0xBAD</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

0x1110: 0b 0001 0001 0001 0000 → VPN = 0x11, TLBI = 1, TLBT = 0x08
TLBT not found in set 1 of TLB, so TLB Miss. PTE is valid & writeable.

0xDEAD: 0b 1101 1110 1010 1101 → VPN = 0xDE, TLBI = 0, TLBT = 0xF
TLBT is found in set 0 of TLB and valid, but not writeable.
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]

The privilege level of each individual module: **user-level** or **kernel-level**.

Most credit given for **monolithic vs. microkernel**.

(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]

In the Blocked state, the process is waiting for a particular event or interrupt to occur before it can proceed.

Many possible examples, such as page fault, waiting for user input, accessing data over the network Internet.

(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]

PCB and TCB are data structures to hold the state of processes and threads, respectively.

Many possible examples, such as pid vs. tid, pointer to the heap or code, list of OS resources.

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

Names of files and pointers to file data & metadata (specifically inodes in UNIX).