Final

Rules

The exam is **open book, open note, open computer.** You may access the book, and your own notes in paper form. You may also use a computer or equivalent to access your own class materials and public class materials. However, you *may not* access other materials except as explicitly allowed below. Specifically:

- You may access a browser and a PDF reader.
- You may access your own notes and problem set code electronically.
- You may access an internet site on which your own notes and problem set code are stored.
- You may access this year's course site.
- You may access pages directly linked from the course site, including our lectures, exercises, section notes, and practice questions.
- You **may** run a C/C++ compiler, including an assembler and linker, or a calculator.
- You may use a Python interpreter.
- You may access manual pages.

But:

- You **absolutely may not** contact other humans via IM or anything like it.
- You **may not** access Piazza.
- You **may not** access an on-line disassembler, compiler explorer, or similar application.
- You **may not** access Google or Wikipedia or anything else except as directly linked from the course site.
- You **may not** access solutions from any previous exam, by paper or computer, **except** for those on the course site.

Any violations of this policy, or the spirit of this policy, are breaches of academic honesty and will be treated accordingly. Please appreciate our flexibility and behave honestly and honorably.

Completing your exam

First, merge your local cs61-exams repository with our handout. You should do this before beginning the exam.

```
$ git pull git://github.com/cs61/cs61-f18-exams.git master
```

You will enter your answers in the `final/final.md` file. Do not place your name in this file. (This enables us to grade all exams blindly.)

When you have completed the exam, edit the file `final/policy.txt` to sign your name. This is your promise that you have obeyed the exam rules in letter and spirit. Then commit your changes and push them to your repository on GitHub:

```
$ git commit -a -m "Final Exam Answers"
$ git push
```

If you get an error message that you do not have access to push to github.com/cs61/cs61-f18-exams.git, this means that you are trying to push to our repository. Instead, push explicitly to your own repository:

```
$ git push https://github.com/cs61/cs61-f18-exams-YOURGITHUBREPONAMEHERE master
```

Make sure that you have entered your exam repository URL on the grading server for the final exam.

Notes

Assume a Linux operating system running on the x86-64 architecture unless otherwise stated. If you get stuck, move on to the next question. If you're confused, explain your thinking briefly for potential partial credit. There are **180 points** total.

1. Course overview I (18 points)

Assume an underlying architecture of x86-64 for all questions.

QUESTION 1A. Undefined behavior: list all that apply.

1. Undefined behavior is a property of the C abstract machine.
2. Undefined behavior is a property of the x86-64 architecture.
3. Undefined behavior is dangerous.
4. Unsigned overflow causes undefined behavior.
5. None of the above.

QUESTION 1B. Pointer arithmetic: list all that apply. Assume `int* x = new int[10]`.

1. `sizeof(x) == 40`.
2. The allocation pointed to by `x` contains at least 40 bytes.
3. The statement `int* y = x + 10` causes undefined behavior.
4. The statement `int* y = x + 11` causes undefined behavior.
5. None of the above.

QUESTION 1C. Size and alignment: list all that apply.

1. Pointers have size 8.
2. `sizeof(int) == 4`.
3. `sizeof(T)` is a multiple of `alignof(T)` for all `T`.
4. Given an array `T x[N]`, `sizeof(x) == sizeof(T) * N`.
5. None of the above.

QUESTION 1D. Registers: list all that apply.

1. Every function must restore all processor registers to their original values on exit.
2. Function return values are stored on the stack.
3. `%rax` and `%eax` refer to (parts of) the same register.
4. `%rbp` and `%ebx` refer to (parts of) the same register.
5. None of the above.

QUESTION 1E. Return address: list all that apply.

1. A function's return address is known at compile time.
2. A function's return address is stored on the stack.
3. A function's return address is protected from modification by the processor.
4. A function's return address is the address of the `call` instruction that invoked the function.
5. None of the above.

QUESTION 1F. Stdio cache: list all that apply.

1. The stdio cache has size 10000 bytes by default.
2. The stdio cache behaves similarly for pipes, disk files, and the terminal.
3. The stdio cache is stored in the buffer cache.
4. The stdio cache is stored in registers.
5. None of the above.

2. Memory word search (15 points)

QUESTION 2A. What is decimal 61 in hexadecimal?

The following is a partial dump of an x86-64 Linux program's memory. Note that each byte is represented as a **hexadecimal** number. Memory addresses increase from top to bottom and from left to right.

```
         ..0 ..1 ..2 ..3 ..4 ..5 ..6 ..7  ..8 ..9 ..a ..b ..c ..d ..e ..f
601080:   00  00  00  00  4f  ef  38  1e   47  97  48  45  ff  ff  00  00
601090:   ff  ff  ff  ff  c3  ff  ff  ff   3d  00  00  00  30  c7  5f  14
6010a0:   3d  00  00  00  00  00  00  00   70  7e  1b  01  00  00  00  00
6010b0:   7c  2d  8f  ba  fd  7f  00  00   c3  ff  ff  ff  ff  ff  ff  ff
6010c0:   a0  10  60  00  00  00  00  00   49  20  74  6f  6f  6b  20  36
6010d0:   31  00  00  00  00  00  00  00   47  9d  ff  ff  ff  7f  00  00
```

QUESTION 2B. What memory segment contains the memory dump?

For questions 3C–3I, give the **last two digits** of the address of the given object in this memory dump (so for address 0x6010a8, you would write "a8"). There's a single best answer for every question, and **all questions have different answers**.

QUESTION 2C. A `long` (64 bits) of value 61.

QUESTION 2D. A `long` of value -61.

QUESTION 2E. An `int` of value 61.

QUESTION 2F. A pointer to an object in the heap.

QUESTION 2G. A pointer to a local variable.

QUESTION 2H. A pointer that points to an object present in the memory dump.

QUESTION 2I. The first byte of a C string comprising at least 4 printable ASCII characters. (Printable ASCII characters have values from 0x20 to 0x7e.)

3. Assembly (15 points)

Helen found the following assembly code while hacking a WeensyOS-like **kernel**. She would like to figure out what the function is doing.

```
__Z3vmlPmm:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rsi, %rax

        shrq    $36, %rax
        andl    $4088, %eax
        movq    (%rdi,%rax), %rax
        testb   $1, %al
        je      error

        andq    $-4096, %rax
        movq    %rsi, %rcx
        shrq    $27, %rcx
        andl    $4088, %ecx
        movq    (%rax,%rcx), %rax
        testb   $1, %al
        je      error

        andq    $-4096, %rax
        movq    %rsi, %rcx
        shrq    $18, %rcx
        andl    $4088, %ecx
        movq    (%rax,%rcx), %rax
        testb   $1, %al
        je      error

        andq    $-4096, %rax
        movq    %rsi, %rcx
        shrq    $9, %rcx
        andl    $4088, %ecx
        movq    (%rax,%rcx), %rax
        testb   $1, %al
        je      error

        andq    $-4096, %rax
        andl    $4095, %esi
        orq     %rax, %rsi
done:
        movq    %rsi, %rax
        popq    %rbp
        retq

error:
        xorl    %esi, %esi
        jmp     done
```

Hexadecimal reference: `4095 == 0xFFF`; `4096 == 0x1000`; `-4096 == 0xFFFFFFFFFFFFF000`; `4088 == 0xFF8`.

QUESTION 3A. How many arguments does the function take, assuming that there are no unused arguments?

QUESTION 3B. What does this function return if it encounters a situation that leads to "`error`"?

Helen realizes that the compiler performed some optimizations that made the code somewhat obscure. For example, these instructions contain constants not present in the C++ source:

```
        shrq    $36, %rax
        andl    $4088, %eax
        movq    (%rdi,%rax), %rax
```

QUESTION 3C. Which of the following snippets is equivalent to the snippet above? List all that apply.

1.
```
    shrq    $33, %rax
    andl    $511, %eax
    movq    (%rdi,%rax,8), %rax
```

2.
```
    shrq    $39, %rax
    andl    $511, %eax
    movq    (%rdi,%rax,8), %rax
```

3.
```
    shrq    $39, %rax
    andl    $511, %eax
    movq    (%rdi,%rax,4), %rax
```

4.
```
    shrq    $33, %rax
    andl    $511, %eax
    movq    (%rdi,%rax,4), %rax
```

QUESTION 3D. If `rdi` and `rax` were C++ variables corresponding to `%rdi` and `%rax`, what would be their likely types?

QUESTION 3E. Please write **one line** of C++ code, using variables `rdi` and `rax`, that could compile into the 3 lines of assembly code we are currently investigating.

QUESTION 3F. How many bits of this function's second argument are used?

QUESTION 3G. Helen notes that the code contains four copies of a pattern of instructions with different constants. After connecting these constants with CS 61 concepts, she's able to tell what the function does without deciphering every line of assembly.

Use a single concise sentence to describe the high-level functionality of this function.

4. Course overview II (15 points)

Assume an underlying architecture of x86-64 for all questions.

QUESTION 4A. Protected control transfer: list all that apply.

1. Protected control transfer refers to the transfer of control from an unprivileged process to the kernel.
2. An unprivileged process can initiate a protected control transfer that executes any instruction in machine memory.
3. Executing a protected control transfer is about as expensive as calling a function.
4. System calls are typically initiated by faults.
5. None of the above.

QUESTION 4B. Virtual memory: list all that apply.

1. Virtual memory requires processor support.
2. Virtual memory involves page tables.
3. Entries in page table data structures contain virtual addresses.
4. Entries in page table data structures include flags that are not related to (i.e., part of) addresses.
5. None of the above.

QUESTION 4C. System calls: list all that apply.

1. `waitpid` might or might not block depending on its arguments.
2. `pipe` might or might not block depending on its arguments.
3. The `kill` system call relates to signals.
4. The `dup2` system call can increase the number of open file descriptors in a process.
5. None of the above.

QUESTION 4D. Pipes and file descriptors: list all that apply.

1. All of a process's file descriptors that refer to the same file share the same file position.
2. All processes in a pipeline generally share one standard input.
3. All processes in a pipeline generally share one standard error.
4. `STDIN_FILENO == 0`.
5. None of the above.

QUESTION 4E. Threads and address spaces: list all that apply.

1. Threads within the same process can share the same address space.
2. Threads from different processes can share the same address space.
3. Each thread has its own stack and set of registers.
4. Accessing a thread's stack from a different thread raises a segmentation fault.
5. None of the above.

5. Tripe (15 points)

QUESTION 5A. Match each system call with the shell feature that requires that system call for implementation. You will use each system call once.

1. `chdir`	a. `;`
2. `fork`	b. `>`
3. `open`	c. `|`
4. `pipe`	d. `cd`
5. `waitpid`	e. Most command lines use this

In the rest of this part, you will work on a new pipe-like feature called a **tripe**. A tripe has one write end and *two* read ends. Any data written on the write end is duplicated onto *both* read ends, which observe the same stream of characters.

QUESTION 5B. A tripe can be emulated using regular pipes and a helper process, where the helper process runs the following code:

```
    void run_tripe(int fd1, int fd2, int fd3) {
/*1*/     while (true) {
/*2*/         char ch;
/*3*/         ssize_t n = read(fd1, &ch, 1);
/*4*/         assert(n == 1);
/*5*/         n = write(fd2, &ch, 1);
/*6*/         assert(n == 1);
/*7*/         n = write(fd3, &ch, 1);
/*8*/         assert(n == 1);
/*9*/     }
    }
```

How many regular pipes are required?

QUESTION 5C. If `run_tripe` encounters end of file on `fd1`, it will assert-fail. It should instead close the write ends and exit. Change the code so it does this, using line numbers to indicate where your code goes. You may assume that `read` and `write` never return an error.

QUESTION 5D. "Pipe hygiene" refers to closing unneeded file descriptors. What could go wrong with the tripe if its helper process had bad pipe hygiene (open file descriptors referring to unneeded pipe ends)? List all that apply.

1. Undefined behavior.
2. `read` from either tripe read end would never return end-of-file.
3. `write` to the write end would never block.
4. `write` to the write end would never detect the closing of a read end.
5. None of the above.

6. Virtual memory (16 points)

x86-64 processors maintain a cache of page table entries called the TLB (translation lookaside buffer). This cache maps virtual page addresses to the corresponding page table entries in the current page table. TLB lookup ignores page offset: two virtual addresses with the same L1–L4 page indexes use the same TLB entry.

QUESTION 6A. A typical x86-64 TLB has 64 distinct entries. How many virtual addresses are covered by the entries in a full TLB? (You can use hexadecimal or exponential notation.)

QUESTION 6B. What should happen to the TLB when the processor executes an `lcr3` instruction?

QUESTION 6C. Multi-level page table designs can naturally support multiple sizes of physical page. Which of the following is the most sensible set of sizes for x86-64 physical pages (in bytes)? **Explain briefly.**

1. 2^{12}, 2^{14}, 2^{16}
2. 2^{12}, 2^{24}, 2^{36}
3. 2^{12}, 2^{21}, 2^{30}
4. 2^{12}, 2^{39}

QUESTION 6D. In a system with multiple physical page sizes, which factors are reasons to prefer allocating process memory using small pages? List all that apply.

1. TLB hit rate.
2. Page table size (number of page table pages).
3. Page table depth (number of levels).
4. Memory fragmentation.
5. None of the above.

7. LRU (20 points)

These questions concern the least recently used (LRU) and first-in first-out (FIFO) cache eviction policies.

QUESTION 7A. List all that apply.

1. LRU is better than FIFO for a workload that consists of reading a file in sequential order.
2. If two LRU caches process the same reference string starting from an empty state, then the cache with more slots always has a better hit rate.
3. If two LRU caches process the same reference string starting from an empty state, then the cache with more slots never has a worse hit rate.
4. LRU and FIFO should have the same hit rate on average for a workload that consists of reading a file in random order.
5. None of the above.

For the next two questions, consider a cache with 5 slots that has just processed the reference string 12345. (Thus, its slots contain 1, 2, 3, 4, and 5.)

QUESTION 7B. Write a reference string that will observe a higher hit rate under LRU than under FIFO if executed on this cache.

QUESTION 7C. Write a reference string that will observe a higher hit rate under FIFO than under LRU if executed on this cache.

The remaining questions in this problem concern the operating system's buffer cache. LRU requires detecting each use of a cache block (to track the time the block was most recently used). In the buffer cache, the "blocks" are physical memory pages, and blocks are "used" by reads, writes, and accesses to `mmap`ed memory.

QUESTION 7D. Which of these changes would let a WeensyOS-like operating system reliably track when buffer-cache physical memory pages are used? List all that apply.

1. Adding a system call `track(uintptr_t physical_address)` that a process should call when it accesses a physical page.
2. Adding a member `boottime_t lru_time` to `struct proc`. (`boottime_t` is a type that measures the time since boot.)
3. Adding a member `boottime_t lru_time` to `struct pageinfo`.
4. Modifying kernel system call implementations to update the relevant `lru_time` members when buffer-cache pages are accessed.
5. None of these changes will help.

QUESTION 7E. The `mmap` system call complicates LRU tracking for buffer-cache pages. Why? List all that apply.

1. `mmap` maps buffer-cache pages directly into a process's address space.
2. Accessing memory in `mmap`ed regions does not normally invoke the kernel.
3. Accessing memory in `mmap`ed regions does not use a page table.
4. `mmap` starts with two of the letter `m`, causing LRU to become confused about which `m` was used least recently.
5. None of the above.

8. Futex (18 points)

In class, we implemented a mutual-exclusion lock using atomics like this:

```
       class mutex {
/* P1*/     std::atomic<int> value_ = 0;
/* P2*/     void lock() {
/* P3*/         while (value_.swap(1) == 1) {
/* P4*/         }
/* P5*/     }
/* P6*/     void unlock() {
/* P7*/         value_.store(0);
/* P8*/     }
       };
```

Recall that `atomic<T>::swap(T x)` atomically swaps this atomic's value with `x` and returns the old value.

QUESTION 8A. Which line of code marks this as a *polling* mutex, or spinlock, rather than a blocking mutex? Give the best single-line answer.

QUESTION 8B. Which of the following statements are true, assuming that the surrounding program uses `mutex` correctly? List all that apply.

1. The mutex is locked if the atomic variable `value_` holds value 1.
2. It is OK to replace the loop condition in line P3 with "`value_.load() == 1`".
3. It is OK to replace the loop condition in line P3 with "`value_.swap(1) >= 0`".
4. It is OK to replace line P7 with "`value_.swap(0);`".
5. None of the above.

In the rest of this part, you will build a *blocking* mutex (like the real `std::mutex`): a call to `lock` should block (without using CPU) until the mutex is unlocked. This requires operating system support.

QUESTION 8C. Gō Mifune proposes to add the following system calls:

- **`xblock()`:** Block until the next call to `xwake()`.
- **`xwake()`:** Wake up any threads currently blocked in `xblock()`.

He uses them as follows:

```
       class mutex {
/* X1*/    std::atomic<int> value_ = 0;
/* X2*/    void lock() {
/* X3*/        while (value_.swap(1) == 1) {
/* X4*/            xblock();
/* X5*/        }
/* X6*/    }
/* X7*/    void unlock() {
/* X8*/        value_.store(0);
/* X9*/        xwake();
/*X10*/    }
       };
```

But this implementation suffers from a sleep–wakeup race condition. Describe how this can occur with two threads by listing lines of code executed by the threads, starting from the following and ending with T2 blocked forever and the mutex unlocked.

```
Initially T1 has locked the mutex.
T2: X2
T1: X7
```

QUESTION 8D. Julia Evans suggests instead using system calls inspired by Linux's `futex`.

- **`futex_wait(std::atomic<int>& x, int expected)`:**
  If `x.value() == expected`, then block until another thread calls `futex_wake(x)` and return 0; otherwise return -1.
- **`futex_wake(std::atomic<int>& x)`:**
  Wake up any threads currently blocked in `futex_wait(x, ...)` and return 0.

She uses them as follows:

```
      class mutex {
/* F1*/    std::atomic<int> value_ = 0;
/* F2*/    void lock() {
/* F3*/        while (value_.swap(1) == 1) {
/* F4*/            futex_wait(value_, 1);
/* F5*/        }
/* F6*/    }
/* F7*/    void unlock() {
/* F8*/        value_.store(0);
/* F9*/        futex_wake(value_);
/*F10*/    }
      };
```

In what ways does Julia's solution **improve** over Gō's implementation? List all that apply.

1. It properly ensures mutual exclusion for each mutex (and Gō's does not).
2. It is free of sleep–wakeup race conditions.
3. The `futex_wait()` call on line F4 suffers from fewer spurious wakeups than the `xblock()` call on line X4.
4. In the best case, a `lock/unlock` cycle requires zero system calls.
5. None of the above.

9. System call implementation (21 points)

QUESTION 9A. Which registers hold (1) the system call number, (2) the first argument, and (3) the return value for WeensyOS system calls? (Refer to `kernel.cc` or `process.hh` if unsure.)

Julia Evans decides to implement the futex-like system calls from the previous problem on her thread-enabled version of WeensyOS. Again:

- **`futex_wait(std::atomic<int>& x, int expected)`**:
  If `x.value() == expected`, then block until another thread calls `futex_wake(x)` and return 0; otherwise return -1.
- **`futex_wake(std::atomic<int>& x)`**:
  Wake up any threads currently blocked in `futex_wait(x, ...)` and return 0.

Note that the `x` arguments are passed as if they were pointers.

QUESTION 9B. List all that apply.

1. The kernel should validate that the `x` arguments are multiples of 4.
2. The kernel should validate that the `x` arguments are valid addresses that point to user-readable memory.
3. The kernel should validate that the `expected` argument is a multiple of 4.
4. The kernel should validate that the `expected` argument is a valid address that points to user-readable memory.
5. None of the above.

QUESTION 9C. Here's a badly-broken initial implementation of the `futex_wait` system call in WeensyOS.

```
      case SYSCALL_FUTEX_WAIT: {
/* W1*/    uintptr_t x = current->regs.{some register};
/* W2*/    uintptr_t expected = current->regs.{some register};
/* W3*/    ... return -1 if x and/or expected are invalid ...
/* W4*/
/* W5*/    std::atomic<int>* x_ptr = (std::atomic<int>*) x;
/* W6*/    if (x_ptr->load() != expected) {
/* W7*/        return -1;
/* W8*/    } else {
/* W9*/        while (true) {
/*W10*/        }
/*W11*/        return 0;
/*W12*/    }
      }
```

What problems does this code have? List all that apply.

1. `std::atomic` methods are implemented using system calls, so `std::atomic` cannot be used in the kernel.
2. The `x_ptr` address computed on line W5 is a physical address, so it cannot be dereferenced.
3. The `x_ptr` address computed on line W5 came from a different address space, so dereferencing it may access the wrong memory.
4. The `return` statements on lines W7 and W11 return incorrect values.
5. If lines W9–10 ever execute, the kernel will hang and make no further progress.
6. None of the above.

Here's a *correct* implementation of the `futex_wake` system call in WeensyOS.

```
      struct proc { ...
          uintptr_t waitaddr;
      }; ...

      case SYSCALL_FUTEX_WAKE: {
/* K1*/    uintptr_t x = current->regs.{some register};
/* K2*/    ... return -1 if x is invalid ...
/* K3*/
/* K4*/    uintptr_t wakeaddr = vmiter(current, x).pa();
/* K5*/    int n = 0;
/* K6*/    for (pid_t pid = 1; pid != NPROC; ++pid) {
/* K7*/        if (procs[pid].waitaddr == wakeaddr && procs[pid].state == P_BLOCKED) {
/* K8*/            procs[pid].state = P_RUNNABLE;
/* K9*/            procs[pid].waitaddr = 0;
/*K10*/            procs[pid].regs.reg_rax = 0;
/*K11*/            ++n;
/*K12*/        }
/*K13*/    }
/*K14*/    return n;
      }
```

QUESTION 9D. Does `proc::waitaddr` contain physical or virtual addresses?

QUESTION 9E. Which line or lines of code unblock waiting processes?

QUESTION 9F. Which line or lines of code determine system call return values?

QUESTION 9G. Complete this implementation of `futex_wait` corresponding to this implementation of `futex_wake`. You will use `P_BLOCKED` (a process state for blocked processes) and `schedule()` (a function that runs some runnable process).

```
case SYSCALL_FUTEX_WAIT: {
    uintptr_t x = current->regs.{some register};
    uintptr_t expected = current->regs.{some register};
    ... return -1 of x and/or expected is invalid ...
```

10. Barrier synchronization (22 points)

This question concerns a synchronization object called a **barrier**. Barriers are useful for phased computations. *N* threads work in parallel; as each thread completes its work, it "arrives" at the barrier, then blocks. When the *N*th thread arrives, all *N* threads are released and continue on to the next phase.

Barriers have two operations:

- `barrier(size_t N)` — Construct a barrier for *N* threads.
- `barrier::arrive_and_wait()` — Arrive at the barrier and block until all *N* threads have arrived.

QUESTION 10A. A *one-use barrier* can only be used once—after the *N* threads arrive, the barrier object cannot be used again (any further calls to `arrive_and_wait` cause undefined behavior). Use atomic variables to implement `one_use_barrier::arrive_and_wait()`. Your function should poll (it need not block).

```
struct one_use_barrier {
    size_t n_;
    std::atomic<size_t> k_ = 0;

    one_use_barrier(size_t n) { this->n_ = n; }

    void arrive_and_wait() {
```

QUESTION 10B. Most solutions allow multiple threads to simultaneously access `one_use_barrier::n_`, which is *not* an atomic variable. Why is this OK?

QUESTION 10C. What are the problems with polling synchronization operations? List all that apply.

1. Polling violates mutual exclusion.
2. Polling causes high CPU usage, leaving less CPU time available for other work.
3. Polling causes deadlock.
4. None of the above.

QUESTION 10D. Complete the following *blocking* one-use barrier.

```
struct blocking_one_use_barrier {
    size_t n_;
    size_t k_ = 0;
    std::mutex m_;
    std::condition_variable_any cv_;

    blocking_one_use_barrier(size_t n) { this->n_ = n; }

    void arrive_and_wait() {
```

QUESTION 10E. *Multi-use* barriers can be used multiple times: once all *N* threads unblock, the barrier can be used again by the same *N* threads. This is surprisingly tricky to get right, but a common solution involves a boolean *sense* variable. Complete this multi-use barrier using atomics and polling.

```
struct multi_use_barrier {
    size_t n_;
    std::atomic<size_t> k_ = 0;
    std::atomic<bool> sense_ = false;

    multi_use_barrier(size_t n) { this->n_ = n; }

    void arrive_and_wait() {
        bool my_sense = this->sense_.load();
```

11. Thank you for taking the class (5 points)

QUESTION 11A. Please give us any feedback here. Any answer but no answer will receive full credit.