New York University
CSCI-UA.202: Operating Systems (Undergrad): Spring 2020
Final Exam

- This exam is **110 minutes**.
- The answer sheet is available here:
  - **Section 001 (Aurojit Panda)**
  - **Section 003 (Michael Walfish)**

  The answer sheet has the hand-in instructions.
- There are **14** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc.** You may refer to TWO two-sided 8.5x11" sheets with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side.
- Do not waste time on arithmetic. Write answers in powers of 2 if necessary.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don't linger. If you know the answer, give it, and move on.
- If you have questions about the exam please go to https://campuswire.com/p/G7536A0B2 (if you need access, use code **7012**).
- **Write your name and NetId on the document in which you are working the exam.**

Do not write in the boxes below.

I (xx/7)	II (xx/21)	III (xx/9)	IV (xx/23)	V (xx/28)	VI (xx/12)	Total (xx/100)

I Fundamentals (7 points)

1. [3 points] The primary architecture that we have used in this class is a _____-bit architecture.

Fill in the blank above with the right number of bits.

64.

2. [4 points] The `read` system call takes three arguments: a file descriptor, and two others:

```
int read(int fd , _____, _____);
```

What are the two other arguments? For each argument, state both its type and what the variable represents.

arg2 is void* or char*, a pointer to the memory area that holds the data to be read. arg3 is size_t, the number of bytes to read.

II Concurrency (21 points)

3. [4 points] Consider a machine with two x86 CPUs, and assume that each CPU is currently executing a different thread *from the same process*. Recall that in the x86 architecture, a CPU's `%cr3` register contains the physical address of the level-1 page table that determines memory translations on that processor.

In this scenario, do the two processors have the same or different values for `%cr3`? Justify your answer in no more than two sentences.

They have the same value of `%cr3`. Threads share memory, and the way that this is implemented is to give them the same "view" of memory; that is, two threads share the same page tables.

4. [2 points] Consider a multi-threaded program that runs correctly under all interleavings on a time-sliced *single* processor with preemptive scheduling. It does not necessarily follow the concurrency commandments.

Is the following statement True or False? "This program is guaranteed to run correctly if each thread is run on a separate CPU of a shared-memory multiprocessor." Justify using two sentences.

False. Code that is correct under a single CPU (and sequential consistency) may be incorrect (because of more possible interleavings) under a different memory model. An example is the double-checked locking code that we saw in class.

5. [15 points] To make a water molecule, assume you need only two Hydrogen atoms and one Oxygen atom (we are ignoring real chemistry). Using monitors, you will write a solution that makes water whenever the needed resources are available. You will fill in three methods: `GenOxygenAtom`, which generates an Oxygen atom; `GenHydrogenAtom`, which generates a Hydrogen atom; and `MakeWaterMolecule`, which generates water.

Each of these methods can be executed an arbitrary number of times concurrently—concurrently with other invocations of the method and concurrently with the other methods. For example, you can imagine 500 threads, each of which is in a loop calling `GenOxygenAtom`; 500 more, each of which is in a loop calling `GenHydrogenAtom`; and another 500, each in a loop calling `MakeWaterMolecule`. To be clear, those numbers are just examples: your code needs to be correct for any number of threads. Note that this kind of concurrency is often our assumption when working with monitors.

As an additional constraints:

- You must follow the class's concurrency commandments.
- The monitor can hold no more than 1000 Oxygen atoms and 2000 Hydrogen atoms; these capacities are independent (so it can hold 1000 Oxygen and 2000 Hydrogen atoms at the same time).
- Avoid unnecessary thread wakeups. Use your judgment about what constitutes an unnecessary wakeup.

```
class Water {

    public:
        Water();  // Initializes state and synchronization variables
        ~Water();

        GenOxygenAtom();
        GenHydrogenAtom();
        MakeWaterMolecule();

    private:
        // FILL THIS IN




};
```

```
// Here and on the next page, give the implementations of
//     Water::Water();
//     void Water::GenOxygenAtom()
//     void Water::GenHydrogenAtom();
//     void Water::MakeWaterMolecule();
```

Space for code and/or scratch

Data members in `Water`:

```
class Water {

....

    private:
        Mutex lock;
        Cond cv_enough_atoms;
        Cond cv_oxy_notfull;
        Cond cv_hyd_notfull;
        m_oxy;
        m_hyd;
        m_water;

};
```

Methods:

```
Water::Water() {
    m_oxy = 0;
    m_hyd = 0;
    m_water = 0;
    lock.init();
    cv_enough_atoms.init();
    cv_oxy_notfull.init();
    cv_hyd_notfull.init();
}

void
Water::GenOxygen() {
    lock.acquire();
    while (m_oxy == 1000) {
        cv_oxy_notfull.wait(lock);
    }
    m_oxy++;
    cv_enough_atoms.signal();
    lock.release();
}

void
Water::GenHydrogen() {
    lock.acquire();
    while (m_hyd == 2000) {
        cv_hyd_notfull.wait(lock);
    }
    m_hyd++;
```

```
    if (m_hyd >= 2)
        cv_enough_atoms.signal();
    lock.release();
}

void
Water::MakeWater() {

    lock.acquire();

    while (! (m_hyd >= 2 && m_oxy >= 1) ) {
        cv_enough_atoms.wait(lock);
    }

    m_water++;
    m_hyd -= 2;
    m_oxy -= 1;

    if (m_hyd <= 1998) // the guard is optional
        cv_hyd_notfull.broadcast(lock);

    if (m_oxy <= 999)  // the guard is optional
        cv_oxy_notfull.signal(lock);

    lock.release();
}
```

III Context switches (9 points total)

6. [9 points] In this question, you will implement `swtch()`, which switches between two *user-level* threads. You will do so for a user-level threading package, running on the *TeensyArch* processor. TeensyArch has 4 general registers, `%r0`-`%r3`, a stack pointer, a base (or frame) pointer, and an instruction pointer `%rip`. Assume the same stack frame structure as the architecture we've been covering in class (x86); further, all registers need to be saved by a function's callee (that is, registers are *callee-saved*, also known as *call-preserved*).

Fill out `swtch()` on the next page. Below are definitions, declarations, and utility functions that you can use.

```
struct thread {
    int thread_id;
    uint64_t stack;
    /* ... */
};

enum register {
    R0,
    R1,
    R2,
    R3,
    RBP,
    RSP
};

// Push CPU's register r to the stack
void push_register(register r);

// Pop from the stack and into the CPU's register r
void pop_register(register r);

// Returns the CPU's current value of register r.
uint64_t read_register(register r);

// Update the CPU's register r so it holds value `value`.
void write_register(register r, uint64_t value);
```

```
// Context switch from thread t1 to thread t2.
void swtch(struct thread *t1, struct thread *t2) {
    // On entry this function is run by thread t1.

    // Your code here. We have started it for you.
    push_register(RBP);
    push_register(R0);

    // YOUR CODE HERE









    return; // The function should return to the
            // point where thread t2 called swtch().
}
```

```
void swtch(struct thread *t1, struct thread *t2) {
    // On entry this function is run by thread t1.

    push_register(RBP);
    push_register(R0);
    push_register(R1);
    push_register(R2);
    push_register(R3);

    t1->stack = read_register(RSP);
    write_register(RSP, t2->stack);

    pop_register(R3);
    pop_register(R2);
    pop_register(R1);
    pop_register(R0);
    pop_register(RBP);
```

```
    return; // The function should return to the
            // point where thread t2 called swtch.

}
```

IV mmap and ptrace (23 points total)

7. [8 points] Consider the following code excerpt that uses `mmap()`:

```
int main(int argc, char* argv[]) {
    // ...
    // readme.txt is a file that is 5KB in length.
    int fd = open("readme.txt", O_RDWR, 0);

    char *map1 = (char*)mmap(NULL, 5120, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    char *map2 = (char*)mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    // assume that neither mmap call fails

    char x = 0;
    char y = 0;
    // Line 1
    for (int i = 0; i < 5120; i++) {
        x = map1[i];
        y = map2[i % 4096];
    }
    // Line 2
    // ...
}

// The signature of mmap is:
//
//   void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
//
// If addr is NULL, the kernel chooses the start virtual address.
//
// If not already in memory, disk blocks are fetched into physical pages on access.
//
// Each separate call to mmap() with the same fd maps the file in a separate
// location and does not undo prior mappings. This means that in the example
// above, the file can be read and written from multiple virtual addresses
// within the same address space.
```

Assume the operating system minimizes the number of virtual and physical pages required in order to implement `mmap()`.

How many last-level (level-4) page table entries are created or modified as a result of the two `mmap()` calls?

3. Page size is 4096, so map1 consumes 2 entries, and map2 consumes one entry.

At line 2, how many physical pages are mapped into the process as a result of the `mmap()` calls?

2. The 3 virtual pages map to two physical pages in the OS buffer cache.

8. [15 points] In class, we studied the system call `ptrace()`, which allows the caller to manipulate a target process in various ways. In this question, you will implement a version of `ptrace` for lab4's WeensyOS:

```
sys_ptrace_getregs(pid_t pid, x86_64_registers* regs);
```

This system call should retrieve values for all processor registers that were in effect at the time just before the process with PID `pid` blocked. You should copy the registers' values to the memory address given by the `regs` parameter.

Assume that we have modified WeensyOS's syscall dispatch logic so that, when a user-level process invokes `sys_ptrace_getregs(pid_t pid, x86_64_registers* regs)`, the kernel calls `ptrace_getregs_impl()`; this latter function is what you will implement. This function should meet the following specification:

- Return -1 on error, 0 on success.
- There is no isolation; any process can ptrace any other valid process.
- It is an error if `pid` does not reference a legitimate process (by non-legitimate, we mean that the corresponding process slot is free);
- It is an error if the memory referenced by `regs` cannot be read or written by the current process.
- To help you check the prior condition, assume you have access to a function:

  ```
  virtual_memory_checkperms(x86_64_pagetable* pt, uintptr va, size_t len, int perm);
  ```

  that returns 0 when the range of memory `[va,va+len)` has permissions `perm` in the view of memory given by `pt`. Returns -1 otherwise.
- You can use as a utility function:

  ```
  void* memcpy(void* dst, const void* src, size_t n);
  ```

  which copies `n` bytes from address `src` to address `dst`.

On the next page, implement `ptrace_getregs_impl()` in syntactically correct C.

```
int ptrace_getregs_impl(pid_t pid, x86_64_registers* regs) {
    // YOUR CODE HERE
    // NOTE 1: The following page has possibly-useful excerpts from WeensyOS
    // NOTE 2: You do not need a lot of code; the staff solution has 5-7 lines









}
```

```
typedef enum procstate {
    P_FREE = 0,                         // free slot
    P_RUNNABLE,                         // runnable process
    P_BLOCKED,                          // blocked process
    P_BROKEN                            // faulted process
} procstate_t;

// Process descriptor type
typedef struct proc {
    pid_t p_pid;                        // process ID
    x86_64_registers p_registers;       // process's current registers
    procstate_t p_state;                // process state (see above)
    x86_64_pagetable* p_pagetable;      // process's page table
} proc;

static proc processes[NPROC];   // array of process descriptors
proc* current;                  // pointer to currently executing proc

// YOU DO NOT NEED THE FUNCTIONS BELOW. We include them to aid with
// recall about certain details in the labs.
//
// virtual_memory_map(pagetable, va, pa, sz, perm, allocator)
//    Map virtual address range `[va, va+sz)` in `pagetable`.
//    When `X >= 0 && X < sz`, the new pagetable will map virtual address
//    `va+X` to physical address `pa+X` with permissions `perm`.
//
//    Precondition: `va`, `pa`, and `sz` must be multiples of PAGESIZE
//    (4096).
//
//    Typically `perm` is a combination of `PTE_P` (the memory is Present),
//    `PTE_W` (the memory is Writable), and `PTE_U` (the memory may be
//    accessed by User applications). If `!(perm & PTE_P)`, `pa` is ignored.
//
//    Returns 0 if the map succeeds, -1 if it fails because a required
//    page table could not be allocated.
int virtual_memory_map(x86_64_pagetable* pagetable, uintptr_t va,
                       uintptr_t pa, size_t sz, int perm,
                       x86_64_pagetable* (*allocator)(void));

// virtual_memory_lookup(pagetable, va)
//    Returns information about the mapping of the virtual address `va` in
//    `pagetable`. The information is returned as a `vamapping` object,
//    which has the following components:
typedef struct vamapping {
    int pn;           // physical page number; -1 if unmapped
    uintptr_t pa;     // physical address; (uintptr_t) -1 if unmapped
    int perm;         // permissions; 0 if unmapped
} vamapping;

vamapping virtual_memory_lookup(x86_64_pagetable* pagetable, uintptr_t va);
```

```
int ptrace_getregs_impl(pid_t pid, x86_64_registers* regs) {
    if (processes[pid].p_state == P_FREE) {
        return -1;
    }

    if (virtual_memory_checkperms(current->p_pagetable,
                                  regs,
                                  sizeof(x86_64_registers),
                                  PTE_P | PTE_U | PTE_W)) {
        return -1;
    }

    memcpy(regs, &processes[pid].p_registers, sizeof(x86_64_registers));
        // alt: *regs = processes[pid].p_registers;

    return 0;

}
```

V Disks and file systems (28 points total)

9. [6 points] A disk driver must schedule disk requests for tracks 10, 22, 20, 2, 40, 6, and 38. A seek takes 6 msec per track moved (note that, among other simplifications, we are not taking into account the length of the seek). Assume that the disk arm is initially at track 20.

How much seek time is needed to handle all of these requests if the driver follows the SSTF (Shortest seek time first) scheduling algorithm? Show your work.

$20-22-10-6-2-38-40$. So the time is $(2+12+4+4+36+2)\cdot 6$ ms = 360 ms.

How much seek time is needed if the driver follows the SCAN (aka elevator) scheduling algorithm? Show your work.

If traveling up first, $20-22-38-40-10-6-2$. So the seek time is $(2+16+2+30+4+4)\cdot 6$ ms = 348 ms. Or if it travels down first: $20-10-6-2-22-38-40$. And seek time is $(10+4+4+20+16+2)\cdot 6$ ms = 336 ms.

10. [8 points] In what follows, Jo executes a program that writes the string "Done with CS202" to the file `notes.txt`. Unfortunately, Jo's computer crashes during this write. For each of the scenarios below, we state where Jo's computer crashed; you will state whether or not `notes.txt` contains the string "Done with CS202" when Jo restarts their computer.

Part (a). Jo's computer is using ZFS, which relies copy-on-write for crash consistency. The following sequence of operations occur:

- A new data block is allocated.
- A pointer to the data block is added to a new copy of the `notes.txt` inode.
- The string "Done with CS202" is written to the newly allocated data block.
- The computer crashes.

When Jo restarts their computer, does `notes.txt` contain the string "Done with CS202"? Justify in one sentence.

No, because a new version of the uberblock wasn't written.

Part (b). Jo's computer is using EXT4, which uses redo logging for crash consistency. The following sequence of operations occur:

- A TxnBegin block is written to the log.
- All of the updates and sub-operations for Jo's write—inoode updates, data bitmap updates, and data block modifications—are logged.
- The computer crashes.

When Jo restarts their computer, does `notes.txt` contain the string "Done with CS202"? Justify in one sentence.

No, because there is no TxnCommit record.

11. [14 points] In this question, you will write a function

```
uint64_t count_blocks_used(struct inode* inode)
```

that reports the number of disk blocks actually consumed by a file, in the setting of lab 5. A function like this is useful for utilities that report disk space usage, such as `du`.

Note that the number of disk blocks actually consumed is *not* derivable by simply inspecting the file's size; the reason is that files in lab 5 can be *sparse*. As an example, say a user-level process does the following on a newly created file, referenced here with `fd`:

```
char buf[] = {'a', 'b', ..., 'z'};
seek(fd, 1000000);           // set the write position to be 1,000,000
write(fd, buf, sizeof(buf)); // write a-z starting at 1,000,000
```

Then the logical file size would be 1,000,026. Without sparsity, this number of bytes would consume least 245 disk blocks (1,000,026 bytes / 4096 bytes/block = 244.15). However, the file consumes only 2 disk blocks: 1 for data and 1 for metadata (specifically, the indirect block).

Some possibly helpful excerpts from the lab are on the following pages, though you do not need every function excerpted. We also include the code for `inode_read()` and `inode_write()`, as these functions implement sparsity, and might provide helpful inspiration. However, if you would find that code distracting, you can disregard it; it isn't necessary for the question.

On the next page, fill in `count_blocks_used()` in syntactically correct C.

```
// Hint 1: The logical size of a file in bytes is stored in the
//         inode's i_size member.
// Hint 2: Remember to handle indirect blocks and double-indirect blocks
// Hint 3: Use inode_block_walk() when you can.
uint64_t count_blocks_used(struct inode* ino)
{
    uint64_t count = 0;
    // YOUR CODE HERE









}
```

```
// Maps a block number to a usable address in memory. Makes use of the
// fact that the simulated disk is memory-mapped.
void* diskaddr(uint32_t blockno);

struct inode {
    uid_t i_owner; // Owner of inode.
    gid_t i_group; // Group membership of inode.
    mode_t i_mode; // Permissions and type of inode.
    dev_t i_rdev; // Device represented by inode, if any.
    uint16_t i_nlink; // The number of hard links.
    int64_t i_atime; // Access time (reads).
    int64_t i_ctime; // Change time (chmod, chown).
    int64_t i_mtime; // Modification time (writes).
    uint32_t i_size; // The size of the inode in bytes.

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t i_direct[N_DIRECT]; // Direct blocks.
    uint32_t i_indirect; // Indirect block.
    uint32_t i_double; // Double-indirect block.
} __attribute__((packed));


int inode_block_walk(struct inode *ino, uint32_t file_blockno,
                     uint32_t **ppdiskbno, bool alloc);
// You implemented the above function in lab5. Here are the comments from the lab:
//
// Find the disk block number slot for the 'filebno'th block in inode 'ino'.
// Set '*ppdiskbno' to point to that slot.  The slot will be one of the
// ino->i_direct[] entries, an entry in the indirect block, or an entry
// in one of the indirect blocks referenced by the double-indirect block.
//
// When 'alloc' is set, this function will allocate an indirect block or
// a double-indirect block (and any indirect blocks in the double-indirect
// block) if necessary.
//
// Returns:
// 0 on success (but note that **ppdiskbno might equal 0).
// -ENOENT if the function needed to allocate an indirect block, but
// alloc was 0.
// -ENOSPC if there's no space on the disk for an indirect block.
// -EINVAL if filebno is out of range (it's >= N_DIRECT + N_INDIRECT +
//               N_DOUBLE).


int inode_get_block(struct inode *ino, uint32_t file_blockno, char **pblk);
// You implemented the above function in lab5. Here are the comments from the lab:
//
// Set *blk to the address in memory where the filebno'th block of
// inode 'ino' would be mapped.  Allocate the block if it doesn't yet
// exist.
```

```
//
// Returns 0 on success, < 0 on error.  Errors are:
// -ENOSPC if a block needed to be allocated but the disk is full.
// -EINVAL if filebno is out of range.



// The size of a block in the file system.
#define BLKSIZE 4096

// The number of blocks which are addressable from the direct
// block pointers, the indirect block, and the double-indirect block.
#define N_DIRECT 10
#define N_INDIRECT (BLKSIZE / 4)
#define N_DOUBLE ((BLKSIZE / 4) * N_INDIRECT)

#define MAX_FILE_SIZE ((N_DIRECT + N_INDIRECT + N_DOUBLE) * BLKSIZE)

// Round down to the nearest multiple of n
#define ROUNDUP(a, n)

// Round up to the nearest multiple of n
#define ROUNDDOWN(a, n)

// inode_read():
// Read count bytes from ino into buf, starting from seek position
// offset. Returns the number of bytes read, < 0 on error.
ssize_t
inode_read(struct inode *ino, void *buf, size_t count, uint32_t offset)
{
    int r, bn;
    uint32_t pos;
    uint32_t *pblkno;
    char *blk;

    if (offset >= ino->i_size)
        return 0;

    count = MIN(count, ino->i_size - offset);

    for (pos = offset; pos < offset + count; ) {
        if ((r = inode_block_walk(ino, pos / BLKSIZE, &pblkno, 0)) < 0)
            switch (-r) {
            case ENOENT: // For sparse files.
                pblkno = NULL;
                break;
            default:
                return r;
            }
        bn = MIN(BLKSIZE - pos % BLKSIZE, offset + count - pos);
        // Handle sparse files.  If no block has been allocated for
```

```
        // this region of the file, fill the read buffer with zeroes.
        if (pblkno == NULL || *pblkno == 0)
            memset(buf, 0, bn);
        else {
            blk = diskaddr(*pblkno);
            memmove(buf, blk + pos % BLKSIZE, bn);
        }
        pos += bn;
        buf += bn;
    }

    return count;
}

// inode_write(): Write count bytes from buf into ino, starting at seek
// position offset.
// Extends the file if necessary.
// Returns the number of bytes written, < 0 on error.
int
inode_write(struct inode *ino, const void *buf, size_t count, uint32_t offset)
{
    int r, bn;
    uint32_t pos;
    char *blk;

    // Extend file if necessary
    if (offset + count > ino->i_size)
        if ((r = inode_set_size(ino, offset + count)) < 0)
            return r;

    for (pos = offset; pos < offset + count; ) {
        if ((r = inode_get_block(ino, pos / BLKSIZE, &blk)) < 0)
            return r;
        bn = MIN(BLKSIZE - pos % BLKSIZE, offset + count - pos);
        memmove(blk + pos % BLKSIZE, buf, bn);
        pos += bn;
        buf += bn;
    }

    return count;
}
```

VI Can't think of a good header (12 points)

12. [8 points] We grade True/False questions with positive points for correct items, 0 points for blank items, and negative points for incorrect items. The minimum score on this question is 0 points. To earn exactly 1 point, cross out the entire question and write SKIP.

Circle True or False for each item below:

True / False The existence of a bug in a server implies the existence of a buffer overflow vulnerability in that server.

False.

True / False The existence of a buffer overflow vulnerability in a server implies the existence of a bug in that server.

True.

True / False A buffer overflow vulnerability in a user-level process implies that the attacker can replace `%cr3`. (Recall that in the x86 architecture, a CPU's `%cr3` register contains the physical address of the level-1 page table that determines memory translations on that processor.)

False.

True / False If the W XOR X security policy is in place, it is nonetheless possible for a buffer overflow vulnerability to be exploited.

True.

True / False In class, we saw examples of primitive device drivers.

True.

True / False In a machine that supports Direct Memory Access (DMA), devices have the ability to load from and store to the machine's RAM.

True.

True / False A buggy device driver running in a non-buggy kernel can cause the kernel to crash.

True.

True / False A buggy process running on a non-buggy kernel can cause the kernel to crash.

False.

13. [3 points] Which of the following aspects of bootstrapping did we cover in the "putting it together" class?

Circle all that apply:

A The CPU copies the firmware into RAM.

B The firmware configures the machine.

C The firmware loads and executes the bootloader.

D The bootloader decompresses the `login()` process.

E The kernel configures the machine.

F The terminal device driver creates an 'init()' process.

A, B, C, E

14. [1 points] This is to gather feedback. Any answer, except a blank one, will get full credit.

Please state the topic or topics in this class that have been least clear to you.

Please state the topic or topics in this class that have been most clear to you.

End of Final

Congratulations on finishing CS202!

Wishing you all health, safety, and stability.