New York University
CSCI-UA.202: Operating Systems (Undergrad): Spring 2021

Final Exam (Given remotely)

- This exam is **110 minutes**.
- 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 open book, open notes, open web.** Recommended sources are the course home page, and all assigned readings. If you consult anything else during the exam, even if you paraphrase it, you MUST cite it (with URL). Not doing this will result in an F on the exam, no questions asked.
- 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.
- **Write your name and NetId on the document in which you are working the exam.**

Do not write in the boxes below.

I (xx/11)	II (xx/24)	III (xx/19)	IV (xx/14)	V (xx/24)	VI (xx/8)	Total (xx/100)

I C and Assembly (11 points)

1. [6 points] The function below, written in C, compiles. However, it is buggy.

```
uint64_t* multiply_by_3(uint32_t x) {
    uint64_t y;
    uint64_t* z;
    y = x * 3;
    z = &y;
    return z;
}
```

Describe the bug in the code, using no more than two sentences.

The function returns the address of a local variable, which is never correct, since after this function is called, its stack frame goes out of scope.

2. [5 points] Here is a program excerpt in x86-64 assembly:

```
    movq  $1, %rax
    movq  $2, %rbx
    movq  $3, %rcx
    movq  $4, %rdx
    movq  $5, %rdi

    pushq %rax
    pushq %rbx
    pushq %rcx
    pushq %rdx
    pushq %rdi

    popq %r8
    popq %r9
    popq %r10
    popq %r11
    popq %r12
```

After this excerpt executes, what values are in each of the registers? Fill in the table below.

register	value
`%rax`	1
`%rbx`	2
`%rcx`	3
`%rdx`	4
`%rdi`	5
`%r8`	5
`%r9`	4
`%r10`	3
`%r11`	2
`%r12`	1

II Concurrent programming (24 points)

3. [15 points] In this problem you will simulate a simplified auction, encapsulated in a monitor, `Auction`. Your job is to finish the implementation of `Auction`. Here is the specification:

- `Auction` has `NUM_BIDS` slots to hold bids; an auction can happen only when all of these slots are full.
- There are endless buyer threads, each of which makes a bid by calling the `Auction::RegisterBid()` method. If there are no free slots to hold bids, this method must wait. Otherwise, this method inserts its bid into a free slot (after which the slot is no longer free). After it does so, the method returns; in particular, the method does not wait for the results of the auction.
- There are one or more auctioneer threads, which each call the `Auction::SelectWinner()` method. This method simulates a single auction. Per the first bullet, this method can proceed only if all slots are occupied. Otherwise, it waits. When it does proceed, it computes the maximum bid from among all candidate bids in the `Auction`, clears all of the slots, and returns the maximum bid.
- You must follow the class's concurrency commandments.
- Don't wake threads unnecessarily.

Hint: Write helper functions, for example to determine whether there are free slots.

Where indicated, fill in the variables and methods for the `Auction` object. Remember to follow the concurrency commandments.

```
Auction auction;  // global

class Auction {

    public:
        Auction();
        ~Auction();
        void RegisterBid(uint32_t amount);
        uint32_t SelectWinner();

    private:
        uint32_t bids[NUM_BIDS]; // bids[i] == 0 means that slot i is free

        // ADD MORE HERE





};
```

```
Auction::Auction()
{
    memset(bids, 0, sizeof(bids));

    // FILL THIS IN







}

void
Auction:RegisterBid(uint32_t amount)
{
    // We use 0 to mean "slot free"
    assert(amount > 0);

    // FILL THIS IN







}



uint32_t
Auction:SelectWinner()
{

    // FILL THIS IN







}
```

```
class Auction {

    public:
        Auction();
        ~Auction();
        void RegisterBid(uint32_t amount);
        uint32_t SelectWinner();

    private:
        uint32_t bids[NUM_BIDS]; // bids[i] == 0 means that slot i is free

        mutex m;
        cond cv_auctionready;
        cond cv_roomforbid;

        int get_avail_slot();
        uint32_t find_max();
};

Auction::Auction()
{
    memset(bids, 0, sizeof(bids));

    // FILL THIS IN
    mutex_init(&m);
    cond_init(&cv_auctionready);
    cond_init(&cv_roomforbid);
}

void
Auction:RegisterBid(uint32_t amount)
{
    // We use 0 to mean "slot free"
    assert(amount > 0);

    // FILL THIS IN
    int idx;

    m.acquire();

    while ((idx = get_avail_slot()) < 0) {
        cond_wait(&m, &cv_roomforbid);
    }

    bids[idx] = amount;
    if (idx == NUM_BIDS-1)
        cond_signal(&m, &cv_auctionready);


    m.release();
}



uint32_t
Auction:SelectWinner()
{
    // FILL THIS IN
    uint32_t winning_bid;

    m.acquire();

    while (get_avail_slot() >= 0)
        cond_wait(&m, &cv_auctionready);

    winning_bid = find_max();

    memset(bids, 0, sizeof(bids));

    cond_broadcast(&m, &cv_roomforbid);

    m.release();

    return winning_bid;
}

// helper functions
int
Auction::get_avail_slot()
{
    int i;
    for (int i = 0; i < NUM_BIDS; i++)
        if (!bids[i])
            return i;

    return -1;
}

uint32_t
Auction::find_max()
{
    uint32_t max = bids[0];
    for (int i = 0; i < NUM_BIDS; i++)
        if (bids[i] > max)
            max = bids[i];

    return max;
}
```

4. [4 points] Recall the multiple reader, single writer example that we saw in class. Pseudocode is below and on the next page. As discussed in class, this example implements a policy: "any waiting writer executes ahead of any waiting reader." Consider this example with the following alternate implementation of `doneWrite`:

```
Database::doneWrite() {
    acquire(&mutex);
    AW--;
    if (WR > 0) {
        broadcast(&okToRead, &mutex);
    }
    if (WW > 0) {
        signal(&okToWrite, &mutex);
    }
    release(&mutex);
}
```

Does this modified example prioritize the execution of readers? If so, explain why. If not, explain why not. Use no more than two sentences.

No, it does not change whose execution is prioritized, it just results in needless wakeups of readers. The policy on which threads get to proceed is enforced by the while statements on lines 20 and 47, not the broadcasting and signaling.

```
1   // assume that these variables are initialized in a constructor
2   state variables:
3       AR = 0;  // # active readers
4       AW = 0;  // # active writers
5       WR = 0;  // # waiting readers
6       WW = 0;  // # waiting writers
7
8       Condition okToRead = NIL;
9       Condition okToWrite = NIL;
10      Mutex mutex = FREE;
11
12  Database::read() {
13      startRead();  // first, check self into the system
14      Access Data
15      doneRead();   // check self out of system
16  }
17
18  Database::startRead() {
19      acquire(&mutex);
20      while ((AW + WW) > 0){
21          WR++;
22          wait(&okToRead, &mutex);
23          WR--;
24      }
25      AR++;
26      release(&mutex);
```

Name: Root **NYU NetId:**

```
}

Database::doneRead() {
    acquire(&mutex);
    AR--;
    if (AR == 0 && WW > 0) {
      signal(&okToWrite, &mutex);
    }
    release(&mutex);
}


Database::write(){  // symmetrical
    startWrite();  // check in
    Access Data
    doneWrite();  // check out
}

Database::startWrite() {
    acquire(&mutex);
    while ((AW + AR) > 0) {

        WW++;
        wait(&okToWrite, &mutex);
        WW--;
    }
    AW++;
    release(&mutex);
}

Database::doneWrite() {
    acquire(&mutex);
    AW--;
    if (WW > 0) {
        signal(&okToWrite, &mutex); // give priority to writers
    } else if (WR > 0) {
        broadcast(&okToRead, &mutex);
    }
    release(&mutex);
}
```

5. [5 points] In the EStore lab (lab 3), in the fine-grained locking section (part C), you should have introduced multiple locks, each covering a unit of code or data smaller than the entire inventory. The purpose was to allow multiple threads to operate on the inventory concurrently. But introducing multiple locks makes code more prone to deadlocks.

What did you do to avoid deadlock? Or, if you did not get this aspect of the assignment correct, what *should* you have done to avoid deadlock? Limit your answer to three sentences, but be precise.

III Virtual memory (19 points)

6. [4 points] In this problem, you will describe how the implementation of `malloc()` can exploit paging so that the system (as a whole) can detect certain kinds of *underruns*, which are a kind of illegal memory reference. Consider this code:

```
int *a = malloc(sizeof(int) * 100); /* allocates space for 100 ints */
int *b = &a[10];   /* b points to the 10th element of array a */
...
*(b - 20) = 5; /* This is an underrun, and is an illegal memory reference. */
```

When the above executes, the process would ideally page fault as a result of an illegal memory reference, at which point the kernel would end the process.

Assume that `malloc()` can manipulate the virtual address space of the process (for example, using `mmap()`, or else you can assume that `malloc()` itself is a system call).

Describe how the implementation of `malloc()` can arrange for page faults when there are underruns like the one above. Do not write more than three sentences.

This was a variant of a question on the midterm. Lay out the allocated memory so that the first legitimately allocated byte is on the first byte of a newly allocated page (this "wastes" the latter part of a page). Mark the prior virtual page (before the array) as "not in use" (this does not cost physical memory). At that point, memory references before the beginning of the array will generate page faults.

7. [15 points] The code below is from WeensyOS (lab 4); this code creates the kernel's page table just after WeensyOS boots up. For reference, the code is in `k-hardware.c`; you're welcome to pull up the whole file if the context will be helpful.

```
 1  // virtual_memory_init
 2  //    Initialize the virtual memory system, including an initial page table
 3  //    'kernel_pagetable'.
 4
 5  static x86_64_pagetable kernel_pagetables[5];
 6  x86_64_pagetable* kernel_pagetable;
 7
 8  void virtual_memory_init(void) {
 9      kernel_pagetable = &kernel_pagetables[0];
10      memset(kernel_pagetables, 0, sizeof(kernel_pagetables));
11      kernel_pagetables[0].entry[0] =
12          (x86_64_pageentry_t) &kernel_pagetables[1] | PTE_P | PTE_W | PTE_U;
13      kernel_pagetables[1].entry[0] =
14          (x86_64_pageentry_t) &kernel_pagetables[2] | PTE_P | PTE_W | PTE_U;
15      kernel_pagetables[2].entry[0] =
16          (x86_64_pageentry_t) &kernel_pagetables[3] | PTE_P | PTE_W | PTE_U;
17      kernel_pagetables[2].entry[1] =
18          (x86_64_pageentry_t) &kernel_pagetables[4] | PTE_P | PTE_W | PTE_U;
19
20      virtual_memory_map(kernel_pagetable, (uintptr_t) 0, (uintptr_t) 0,
21                         MEMSIZE_PHYSICAL, PTE_P | PTE_W | PTE_U, NULL);
22
23      lcr3((uintptr_t) kernel_pagetable);
24  }
```

Assume that, before `virtual_memory_init()` executes, the boot loader has configured virtual memory to set up an identity mapping ("virtual address x maps to physical address x") for all virtual addresses that `virtual_memory_init()` uses when executing (its instruction addresses and the memory addresses that it loads from and stores to). If that assumption confuses you, you can ignore it.

The purpose of `virtual_memory_init()` is to construct and configure the kernel's page tables (Lines 9–21) and then to load the physical address of the level-1 (L1) page table into the x86-64's `%cr3` register (Line 23), thereby telling the MMU to translate virtual addresses according to this page table.

This question asks you to walk through each of the seven (7) lines of code in lines 9–21, and describe the specific function of each line. (There are seven (7) lines of code because a *line of code* is defined as code that ends in a semicolon, even if that line breaks across multiple literal lines.) Your answer should cover the following.

- What does each line of code specifically do to construct the kernel's page table?
- What are the contents of the L1, L2, L3, and L4 page tables?
- Once `virtual_memory_init()` completes execution, which virtual address ranges are mapped to which physical address ranges?

Name: Root **NYU NetId:**

IV I/O (14 points)

8. [9 points] Consider a disk with the following characteristics:

- The disk has 8 platters (and 8 corresponding heads); changing which head is active has zero cost.
- The disk makes 128 rotations per second.
- Each sector is 4096 bytes.
- There are 1024 tracks per platter.
- Inner tracks have fewer sectors than outer tracks, specifically for a given platter, the number of sectors on track i is $4 \cdot i$, for $i = 1, \ldots, 1024$.
- The track-to-track seek time is 0 milliseconds.
- The average seek time for a read is 10.5 ms; for a write it is 12 ms.
- The maximum seek time is 16 ms.
- Ignore the time to transfer the bits from the disk to memory; that is, once the disk head is positioned over the sector, the transfer happens instantaneously.

For the questions below, you will generally want to be working with powers of 2. To that end, here is a potentially helpful table:

$2^6 = 64$	$2^{11} = 2048$
$2^7 = 128$	$2^{12} = 4096$
$2^8 = 256$	$2^{13} = 8192$
$2^9 = 512$	2^{20} bytes = 1 megabyte (MB)
$2^{10} = 1024$	2^{30} bytes = 1 gigabyte (GB)

What is the storage capacity of the disk in bytes or gigabytes? Explain *briefly* (for example by showing your work).

$2^{26} \cdot (2^{10}+1) \approx 64$ GB. Each platter has $\sum_{i=1}^{1024}(4 \cdot i)$ sectors. There are 4096 bytes/sector, and 8 platters. The total bytes are thus:

$$\begin{aligned}
8 \cdot 4096 \cdot \sum_{i=1}^{1024}(4 \cdot i) &= 2^3 \cdot 2^{12} \cdot 2^2 \cdot \sum_{i=1}^{1024} i \\
&= 2^{17} \cdot (1024 \cdot 1025/2) \\
&= 2^{17} \cdot 2^9 \cdot (1024+1) \\
&= 2^{26} \cdot (2^{10}+1) \\
&\approx 2^{36} \text{ bytes} = 64 \text{ GB}
\end{aligned}$$

Now, assume that we have two disks of the kind above, connected to the same machine. We arrange to make them exact replicas of each other; this is known as RAID-1, and we now refer to a *logical sector*, recognizing that we can fetch a logical sector's data from either of the two identical disks. You can do the question below even if you did not answer the prior question.

If the two-disk system is at capacity (that is, storing the maximum amount of data), what is the minimum time in seconds to read all logical sectors into memory? (Assume that there is enough RAM to store all of the data.) Explain briefly, for example by showing your work.

32 seconds. The system can read two different tracks in parallel at the same time, one from disk 1 and one from disk 2. So the system can divide up the tracks, for example reading cylinders 1-512 from disk 1 and cylinders 513-1024 from disk 2. The answer is thus the time taken to read half of the tracks on a disk. A disk has 8 platters * 1024 tracks/platter = 2^{13} tracks. Half of those tracks is $2^{13}/2 = 2^{12}$ tracks. We can read one track per rotation, and there are 128=2^7 rotations/second, so: 2^{12} tracks * 1 rotation/track * 1 second/2^7 rotations = $2^5 = 32$ seconds.

9. [5 points] Assume an operating system (OS) that presents a *completely synchronous I/O interface* to processes. This means: (1) if a system call requires the OS to perform I/O, that system call blocks until the I/O is complete; and (2) there is no way for a process to ask the OS whether a given call *would* block. (As a technical point, there is no memory mapping of files.) Assume a process that has four *user-level* threads.

How many I/O requests from this process can be pending at the OS at once? Explain your answer in no more than two sentences.

1 request. The operating system does not "see" the user-level threads: it sees the process as a unit. Because any I/O-inducing system call blocks the entire process, there is no way for a single process to achieve I/O parallelism in this setup.

V File systems and crash recovery (24 points)

10. [4 points] In the classic Unix file system, the i-node is an *imbalanced tree*. The purpose of the imbalance is to optimize access to short files: for short files, one can get the fileblock-to-diskblock mappings by looking directly in the inode, with no need to seek to indirect blocks. Your friend has an idea: *let's avoid such seeks even for large files, by expanding the number of direct block slots in an inode to* 2^{23} *(*≈ 8 *million) such slots.* This idea would indeed eliminate seeks to indirect blocks even for fairly large files. But it has a substantial disadvantage.

State the disadvantage of your friend's idea. Use no more than two sentences.

Inodes aren't sized dynamically (each has a fixed, reserved location on the disk), so every inode would now have to be enormous, which is wasteful since most files are small. Thus, we are wasting space on disk and in memory (inodes are cached in memory).

11. [6 points] Your friend wants to build a file system that tolerates crashes. Your friend proposes *write-behind journaling*. In this proposal, there is a journal, but the file system writes to the journal only *after* checkpointing ("checkpointing", recall, means applying an operation to the on-disk data structures). Specifically, (1) the file system writes a TxnEnd record for a given transaction only after the TxnBegin record and all journal entries for the given transaction are written, and (2) the file system writes individual journal entries only after checkpointing the operation described by that entry. The recovery protocol looks for incomplete operations (those that are part of a transaction that lacks a TxnEnd record) and undoes those operations, similar to the way that recovery works for undo-only logging.

Assume that a crash can happen at any time. Does your friend's proposal work? If so, argue that it is correct. If not, explain why not. Use no more than four sentences.

This does not work. It violates the golden rule of atomicity (never modify the only copy). In more detail, there are certain operations that require writing to more than one place on the disk, for example, appending to a file (which might require writing to an indirect block, updating a bitmap, and updating the inode itself). If the machine crashes in between these different disk updates, then because the operation is not complete, the system never got to the point of writing a journal entry. Thus, the on-disk data structures are now inconsistent, and there is no indication in the logs. The recovery process as described does not detect or fix this issue.

12. [14 points] This question has two independent parts; both take place within the lab 5 file system structure and code. Helper functions and declarations are on the next page.

Part A. Recall that an inode implements a mapping between virtual file block numbers and actual disk block numbers; that is, for file block *k*, one can consult the inode to find which disk block, if any, holds the data of file block *k*.

You will write a function that, for a given inode `ino`, returns the disk block number associated with virtual file block 8225 (that is, the number of the disk block that holds the data of file block 8225). Your function should not call `inode_block_walk()` or `inode_get_block()`, and it should not have any hard-coded constants or magic numbers except 8225; in particular, your function should *compute* the indexes that it needs to use in various data structures. You can and should assume that file block 8225 is allocated. Write in syntactically valid C.

Fill in `fileblock8225_to_diskblock()` below, in syntactically valid C.

```
uint32_t fileblock8225_to_diskblock(struct inode *ino)
{
    // FILL THIS IN.
    //   - Do not use inode_block_walk or inode_get_block.
    //   - Assume that file block 8225 is allocated.







}
```

Part B. Write a function, `inode_off2byte()`, that, for a given inode `ino`, takes as input an offset in the corresponding file and returns the byte at that offset. In this part, you can use `inode_block_walk()` or `inode_get_block()`. You may assume that the block containing the requested offset is allocated. Write in syntactically valid C.

Fill in `inode_off2byte()` below, in syntactically valid C.

```
char inode_off2byte(struct inode* ino, uint32_t offset)
{
    // FILL THIS IN.







}
```

Name: Root **NYU NetId:**

```
// 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);

// You implemented this function in lab5. It finds the disk block number slot
// for the 'filebno'th block in inode 'ino'. It sets '*ppdiskbno' to point
// to that slot. Returns 0 on success, < 0 on error.
int inode_block_walk(struct inode *ino, uint32_t filebno, uint32_t **ppdiskbno, bool alloc);

// You implemented this function in lab5. It sets *pblk to the address in
// memory where the filebno'th block of inode 'ino' is mapped.
// Returns 0 on success, < 0 on error.
int inode_get_block(struct inode *ino, uint32_t file_blockno, char **pblk);

/* Excerpted from fs_types.h in lab5 */

// 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)

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));
```

VI Security and feedback (8 points)

13. [7 points] Assume that a multiuser Unix system has a buggy implementation of passwd. As usual, passwd is a setuid executable that is owned by root, world executable, and world readable; thus, any user on the system can invoke passwd, and when they do, the resulting process runs with root's privileges. The aforementioned bug allows any invoker of passwd to mount a buffer overflow attack, and thereby gain a root shell. Assume that this version of passwd is the only buggy program on the system (that's wildly unrealistic, but it will keep the question focused).

An adversarial user on the system knows about the vulnerability in passwd. This user figures that eventually the root user (as system administrator) will replace passwd with a non-buggy version. But the adversarial user wants to be able to invoke the buggy version of passwd in the future to gain root access. So the adversarial user takes some action A.

Later, the root user indeed replaces passwd with a non-buggy version:

```
# rm /sbin/passwd
# apt source passwd  // gets the source for the updated passwd
...
# make install       // replaces /sbin/passwd with a non-buggy version
```

Yet, some time after root does the above, the adversarial user somehow gets a root shell.

What was action A? Explain *why* action A resulted in the attacker being able to get a root shell. Limit your answer to three sentences. Hint: the attacker didn't copy passwd; invoking the copy would not have been sufficient to gain a root shell.

The attacker locally created a hard link to the buggy version, probably naming it something innocuous like "fluffybunny." Because of the extra link, when root "removed" the buggy binary, the inode stuck around. Furthermore, the binary executable still had the same permissions (setuid, world-executable), as these are stored in the inode; thus, the attacker was able to keep using that file, via the name "fluffybunny."

No one gave the intended answer. The most common answer, by far, was variants of the trusting trust attack (others were to replace the source repo, or to hack apt). The real lesson here is that once a system has been compromised, any actions it takes in the future are potentially compromised.

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

Congrats on finishing CS202!!