This exam is 110 minutes. Stop writing when “time” is called. You must turn in your exam; we will not collect it. Do not get up or pack up in the final ten minutes. The instructor will leave the room 115 minutes after the exam begins and will not accept exams outside the room.

There are 16 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.

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 this cover sheet and on the bottom of every page of the exam.

Do not write in the boxes below.

<table>
<thead>
<tr>
<th>I (xx/13)</th>
<th>II (xx/23)</th>
<th>III (xx/21)</th>
<th>IV (xx/20)</th>
<th>V (xx/23)</th>
<th>Total (xx/100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name:

NetId:
I  Fundamentals (13 points total)

1. [3 points] How many kilobytes (KB) are in a megabyte (MB)?
   State your answer. You do not need to show work.

2. [5 points] To make a request of the operating system, a process issues a ________________________.
   Fill in the blank above.

3. [5 points] The function copypath(), below, is supposed to copy path (more precisely, the bytes
   pointed to by path) to the appropriate field in struct entry. (You have seen this pattern in lab 7.)
   Fill in copypath(). Note that a sample use of this function is below.

   ```
   #define PATH_MAX 1024
   #include <string.h>

   struct entry {
       int n;
       char path[PATH_MAX];
   };

   // assume that length of 'path' is < PATH_MAX, and 'path' is a null-terminated string
   void copypath(struct entry* p, const char* path)
   {
       // YOUR CODE HERE
   }
   ```

   Here is a sample use of copypath():

   ```
   #include <stdio.h>

   int main(int argc, char** argv)
   {
       char* path = "abcdef";
       struct entry e;

       e.n = 567;
       copypath(&e, path);

       printf("%s\n", e.path);
       return 0;
   }
   ```
II Concurrency (23 points total)

4. [5 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 page directory (also known as a 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.

5. [3 points] Consider the implementation of the spinlock that we saw in lecture. This implementation relies on which of the following mechanisms?

Circle the BEST answer:

A A mutex
B A queue of quiescent waiters
C An atomic processor instruction
D A deadlock detection
E A monitor
6. [15 points] In this problem, you will synchronize access to a cross-roads traveled by bicycles. Bicycles travel in straight lines. Each bicycle is headed north, south, east, or west. (If a bicycle is headed east, for example, then it approaches the cross-roads from the west.)

The cross-roads can be in east-west mode: this means that east and west bicycles can go, and the north and south bicycles must wait. Or it can be in north-south mode, which is the other way around. The mode changes if and only if the cross-roads is empty and an orthogonal (to the current mode) bicycle wishes to enter. The cross-roads can hold 5 bicycles in each of the two current directions. For example, if the cross-roads is in east-west mode, it can accommodate 5 bicycles going east and 5 going west.

The cross-roads is a monitor; you will complete the implementation of this monitor. Each bicycle is a thread that calls into the monitor before entering the cross-roads; this call may cause the bicycle to wait. A bicycle also invokes the monitor after exiting the cross-roads. Pseudocode for a bicycle is on the next page. Some notes about your task:

- Your solution must allow multiple bicycles to be using the intersection at once, and should not make any assumptions about the number of bicycles.
- You must follow the concurrency commandments. You must use only one mutex. You must not have busy waiting or spin loops.
- We have given you some helper functions that may be useful.

Below, write down the conditions under which a bicycle can and cannot enter the cross-roads. Informal text is fine. This exercise will help you with the rest of the problem.

Name: NYU NetId: 
In the remainder of the problem, complete the implementation of the Xroads monitor.

typedef enum {NORTH=0, SOUTH=1, EAST=2, WEST=3} dir_t;
typedef enum {NORTHSOUTH=0, EASTWEST=1} mode_t;

void bicycle(thread_id tid, Xroads* xr, dir_t direction)
{
    /* you should not modify this function */
    Ride_up_to_crossroads();

    xr->Enter(direction);
    Ride();
    xr->Exit(direction);

    Ride_after_crossroads();
}

class Xroads {

public:
    Xroads(); // You will complete this
    ~Xroads() { }
    void Enter(dir_t d); // You will implement this
    void Exit(dir_t d); // You will implement this

private:
    bool IsEmpty();
    mode_t Dir2Mode();
    mode_t mode;
    uint32_t num[4];

    // ADD MATERIAL BELOW THIS LINE

};

// HELPER FUNCTIONS
bool Xroads::IsEmpty()
{
}

mode_t Xroads::Dir2Mode(dir_t d)
{
    if (d == NORTH || d == SOUTH) return NORTHSOUTH;
    else return EASTWEST;
}
// Below, complete the implementation of
// Xroads::Xroads()
// and give the implementations of
// void Xroads::Enter(dir_t d)
// void Xroads::Exit(dir_t d)
// Reminder: you need to add to the definition of Xroads on the previous page
Xroads::Xroads()
{
    memset(num, 0, sizeof(num));
    mode = NORTHSOUTH;

    // ADD SOME STUFF HERE
}

III Memory, scheduling, and some other stuff (21 points total)

7. [8 points] Consider the memory pages depicted below. The physical addresses appear on the left side of 4-byte cells. For example, the physical address of the “top-left” memory cell is \(0xf0f02ff\), and the four byte content of this cell is \(0xf00f3007\):

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0f02000</td>
<td>0xffff5000</td>
</tr>
<tr>
<td>0xf0f02004</td>
<td>0xff0ff000</td>
</tr>
<tr>
<td>0xf0f02ffc</td>
<td>0xf03007</td>
</tr>
<tr>
<td>0xffff1000</td>
<td>0xf0f007</td>
</tr>
<tr>
<td>0xffff1004</td>
<td>0xff5804</td>
</tr>
<tr>
<td>0xffff1ffc</td>
<td>0xd5202007</td>
</tr>
<tr>
<td>0xffff2000</td>
<td>0xffff5000</td>
</tr>
<tr>
<td>0xffff2004</td>
<td>0xffff5000</td>
</tr>
<tr>
<td>0xffff5000</td>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>0xffff5004</td>
<td>0xff005007</td>
</tr>
<tr>
<td>0xffff5008</td>
<td>0xff005007</td>
</tr>
</tbody>
</table>

We assume an x86 32-bit processor and the standard two-level page table structure. The current value of %cr3 (the pointer to the page directory) is \(0xffff1000\). Recall that permissions associated with a virtual memory page are encoded by setting the bottom 3 bits of the corresponding page table entry. If a bit is set, the more permissive setting is enabled. Ignore segmentation. Consider the following code:

```c
int* ptr1 = (int*)0x00001ffc;
int* ptr2;

printf("%x
", *ptr1); // %x tells printf to format the integer in hexadecimal */
*ptra = 0xaced202;
printf("%x
", *ptr1);
ptr2 = 0x0;
printf("%x
", *ptr2);
```

What is the output of this program in the environment described above? If the program would fault, describe its output up to the fault, and indicate the line of the fault.
8. **[4 points]** In our units on scheduling and page replacement, we defined an *optimal policy* for that context: a discipline that, if followed, would result in minimizing or maximizing some quantity of interest. (For scheduling, it was minimizing response time; for page replacement, it was maximizing the number of cache hits, if we regard physical frames as cache entries for a backing store on the disk.) However, in both cases, we noted that it was impossible to implement the optimal policy.

**State below, in one sentence, why these optimal policies are impossible to implement.**

9. **[9 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 on this question, cross out the 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.
- **True / False** The existence of a buffer overflow vulnerability in a server implies the existence of a bug in that server.
- **True / False** A buffer overflow vulnerability in a user-level process implies that the attacker can replace %cr3.
- **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 / False** It is possible to conduct the airport security attack that we discussed in lecture without having root access on any computer.
- **True / False** In class, we saw examples of primitive device drivers.
- **True / False** A buggy device driver running in a non-buggy kernel can cause the kernel to crash.
- **True / False** On the x86, if a given memory reference from user mode results in a page fault, then it follows that the memory reference also generated a TLB miss.
IV File systems (20 points total)

10. [20 points] Recall that file systems often use a bitmap to record which disk blocks are currently in use (a bitmap is an array of bits, one for each resource that is being tracked). Although bitmaps are often stored on the disk, this is not fundamentally necessary: it is possible to reconstruct the bitmap in memory, by traversing the file system. In this question, you will write code to perform this traversal, in the context of the lab 6 file system structure and code.

Your top-level goal is to call a function `mem_bitmap_set(blockno)` for each disk block that is in use by the file system. Assume that this function does the work of setting the bit for `blockno` in an in-memory bitmap. You will proceed in two parts.

**Part 1.** Fill in `mark_blocks()`, using syntactically correct C. This function takes as input an inode number; it should invoke `mem_bitmap_set()` for each data block and metadata block used by the associated file.

**Part 2.** Complete the implementation of `traverse(uint32_t inum)`, using syntactically correct C. This function takes as input an inode number. It first invokes `mark_blocks` (which you implemented in part 1). If the inode is an ordinary file, then `traverse()` is done. But if the inode corresponds to a directory, then `traverse()` should traverse each of the files or directories within this directory; you will write the logic for this traversal.

To give you context, the idea is that, after you have completed both parts, the file system can call:

```
traverse(super->s_root); // s_root is the number of the root directory inode
```

and this will result in the correct configuration of the in-memory bitmap.

Fill in `mark_blocks()` and `traverse()` on the following pages. Some helper functions are below.

```
void mem_bitmap_set(blockno);

void* diskaddr(uint32_t blockno);

int inode_get_block(struct inode *ino, uint32_t file_blockno, char **pblk);
```

Name: NYU NetId:
void mark_blocks(uint32_t inum)
{
    int i;
    struct inode* ino = diskaddr(inum);

    // YOUR CODE HERE
}
void traverse(uint32_t inum) {
    struct inode* ino;
    uint32_t nblocks;
    int i, j;
    int r;
    char* blk;
    struct dirent* d;

    mark_blocks(inum);

    ino = (struct inode*)diskaddr(inum);

    // is the current inode a directory? if not, return
    if (!S_ISDIR(ino->i_mode))
        return;

    // get the number of data blocks used by the directory
    nblocks = ino->i_size / BLKSIZE;

    for (i = 0; i < nblocks; i++) {
        // YOUR CODE HERE.
        // Hints:
        //    -- You will need to call inode_get_block(). Its function
        //      signature and definition were given two pages prior.
        //    -- Look closely at the definitions in fs_types.h (next page)
        //    -- You will need to make one or more recursive calls,
        //      meaning that your implementation should itself invoke traverse()
    }
}
/* Excerpted from fs_types.h in lab6 */

// 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 {
    uint32_t i_owner; // Owner of inode.
    uint32_t i_group; // Group membership of inode.
    uint32_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));

struct dirent {
    uint32_t d_inum; // Block number of the referenced inode.
    char d_name[NAME_MAX]; // File name.
} __attribute__((packed));

// The number of struct dirents in a data block.
#define BLKDIRENTS (BLKSIZE / sizeof(struct dirent))

#define FS_MAGIC 0xC5439513 // The magic number signifying a valid superblock.

struct superblock {
    uint32_t s_magic; // Magic number: FS_MAGIC.
    uint32_t s_nblocks; // Total number of blocks on disk.
    uint32_t s_root; // Inum of the root directory inode.
} __attribute__((packed));
V Transactions, distributed systems, and security (23 points total)

11. [4 points] You are building a system with persistent on-disk data structures, and you want the integrity of those data structures to be preserved, even in the face of crashes.
What is the golden rule of atomicity (for crash consistency) in this scenario?

12. [4 points] Consider two-phase locking. What problem is solved by this mechanism? Do not use the word “atomicity” in your answer (the word is ambiguous in this context).
Use no more than one sentence; remember to describe the “why”, not the “how” or “what”.

13. [3 points] Which of the following mechanisms described by OSTEP is used to provide message integrity in the networking context?
Circle the BEST answer:

A Checksums
B File system superblocks
C The additional arguments taken by UDP_Open
D Sequence numbers
E The class’s collaboration policy

14. [5 points] Recall the two-phase commit protocol described in lecture. Suppose that the coordinator fails after logging COMMIT on disk and sending PLEASE COMMIT to all n workers. Suppose that the PLEASE COMMIT message reaches i of the n workers and that the workers implement a protocol in which they communicate with one another when they suspect that the coordinator has failed.
Can the workers apply the transaction without waiting for the coordinator to recover? If yes, give the minimum value of i for which this is possible, and explain why. If not, explain why not. Limit your answer to two sentences.
15. **[5 points]** We studied attacks wherein an adversary writes, compiles, and executes a C program that does the following in order:

- Manipulates its environment in some way (for example, closing a file descriptor)
- Invokes a setuid-root program using `exec()` (for example, `exec("/usr/bin/passwd")`).

Your friend is disturbed by the danger of setuid, and makes a suggestion: “Let us modify the operating system so that, if a non-root process attempts to `exec()` a setuid binary, the setuid bit is ignored.”

**What is the problem with your friend’s suggestion? What would break if we implemented it?**

Use no more than two sentences.

16. **[2 points]** This is to gather feedback. Any answer, except a blank one, will get full credit.

**Please state the two topics in this course that you most enjoyed:**


**Please state the two topics in this course that you least enjoyed:**
Space for code and/or scratch paper
End of Final

Congratulations on finishing CS202!

Enjoy the summer!