New York University
CSCI-UA.202: Operating Systems (Undergrad): Fall 2019

Final Exam

- 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 **18** 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.
- **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.

I (xx/18)	II (xx/24)	III (xx/20)	IV (xx/17)	V (xx/11)	VI (xx/10)	Total (xx/100)

Name: Solutions

NetId:

I Fundamentals, Concurrency (18 points total)

1. [3 points] How many bytes are in a megabyte (MB)?

State your answer as a power of 2.

2^{20}.

2. [5 points] To make a request of the operating system, a process issues a __________________________.

Fill in the blank above.

System call.

3. [5 points] Let cv be a condition variable, and let mutex be a mutex. Assume that there are only two threads, and a single CPU. Consider this pattern:

```
acquire(&mutex);
if (not_ready_to_proceed()) {
    wait(&mutex, &cv);
}
release(&mutex);
```

Under the above assumptions, when is this pattern correct? Follow the concurrency commandments. Answer in one sentence or less.

Never.

4. [5 points] As in class, we assume that threads are preemptively scheduled by the OS. Also, synchronization primitives (mutex acquire, condition variable wait, etc.) are system calls. *Note that the process state diagram applies to individual threads.* (If you don't remember the diagram, don't worry; you can still do the problem.) For example, `READY` means that the OS is free to run a thread's user-level instruction from the thread's saved instruction pointer, and `BLOCKED` means that the OS cannot run the thread (for example, because some request that the thread made is not complete). Consider the following code structure:

```
Mutex m;

void f() {
    m.acquire();
    // Critical section. In this critical section, there are
    // NO synchronization calls or other system calls.
    m.release();
}
```

Is the following statement True or False? "Once a thread enters the critical section above, the thread will not transition out of `RUNNING` until it releases the mutex." Justify using one sentence.

False. Given preemptive scheduling, context switches can happen at any time. For example, a timer interrupt could arrive while the thread is in the critical section, and the kernel could decide to switch to a different `READY` thread in another process.

II Virtual Memory (24 points total)

5. [4 points] Below, assume a machine for which virtual memory and paging are enabled.

(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 for False for each item below:

True / False On the x86, if a memory reference causes a page fault, then it also causes a TLB miss.

False.

True / False On the x86, if a given page table entry's `PTE_P` bit is not set, then a memory access to the corresponding virtual page by *a process* causes a page fault.

True.

True / False Same question as above, but with "process" replaced by "kernel": On the x86, if a given page table entry's `PTE_P` bit is not set, then a memory access to the corresponding virtual page by *the kernel* causes a page fault.

True.

True / False If a computer's physical memory is larger than the size of the address spaces of all processes on that computer put together, then there is no need for virtual memory, since all of the processes can fit in physical memory.

False. VM has lots of uses besides managing access to scarce frames.

6. [20 points] This question takes place in the environment of lab 4. It has 3 parts; the parts are independent. Useful definitions and helper functions are on the next page of this exam. In all parts, assume that your code is running within the kernel. **Write in syntactically correct C.**

Part A Write a function, `alloc_ppn()`, that returns the page *number* of an unused physical page and marks that page in-use; you do not need to zero-out the page itself. If there are no available physical pages, return -1.

```
int alloc_ppn() {








}
```

Part B Write a function, called `make4appear(proc* p)`, that makes the physical address range `[0x40000,0x41000)` appear as present (PTE_P) and user-readable (PTE_U), but not writable, in `p`'s virtual address space starting at virtual address `0x90000`. Assume that the required page tables exist; your function does not need to allocate them. You need only one line of code.

```
void make4appear(proc* p)
{




}
```

Part C Write a function, called `stack_physicaladdress(proc* p)`, that translates `proc p`'s saved stack pointer (which, as we've studied, is a virtual address in `p`'s address space) to a physical address. You only need several lines.

```
uintptr_t stack_physicaladdress(proc* p)
{




}
```

```
// 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
    x86_64_pagetable* p_pagetable;      // process’s page table
} proc;

typedef struct x86_64_registers {
    uint64_t reg_rax;
    uint64_t reg_rcx;
    uint64_t reg_rdx;
    uint64_t reg_rbx;
    uint64_t reg_rbp;
    /* [some fields omitted] */
    uint64_t reg_rip;
    /* [some fields omittted] */
    uint64_t reg_rflags;
    uint64_t reg_rsp;
    /* [some fields omitted */
} x86_64_registers;

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);

// refcount is the number of times that the given physical page is
// currently referenced. 0 means it’s free.
typedef struct physical_pageinfo {
    int8_t owner;  // you can ignore this field on this exam
    int8_t refcount;
} physical_pageinfo;

static physical_pageinfo pageinfo[PAGENUMBER(MEMSIZE_PHYSICAL)];
```

III I/O, and Some Miscellaneous Questions (20 points total)

7. [10 points] As discussed in class, two ways for an operating system to become aware of external events associated with a device are interrupts and polling. We observed that if a computer were receiving many interrupts, it might spend all of its time processing them and not get other work done; in that case, the operating system should switch to polling the device. Now consider the following:

- A computer has an attached keyboard. The keyboard has a 1024-byte internal memory buffer to hold the codes of recently-pressed keys, each of which consumes 2 bytes of buffer space. (The buffer is a FIFO, which for our purposes means that the OS simply reads from it and doesn't manage the memory; if this parenthetical confuses you, you can ignore it.)
- This computer and its OS take 1 microsecond (10^{-6} seconds) to handle an interrupt from the keyboard. That duration includes everything: reading from the keyboard's buffer, determining which key was pressed, and painting the appropriate letter to the screen.
- Assume that polling requires a fixed cost of 1 microsecond per poll. Further assume that, per poll, the operating system can read an arbitrary amount of the keyboard's internal memory buffer, up to the entire size of that buffer.
- Assume that, if polling, the operating system checks the device in question every 200 milliseconds.
- Assume that humans are sensitive to lags of 100 milliseconds or greater. Specifically, if a human types a letter, that letter must appear on the screen less than 100 milliseconds after the human types it, to avoid annoyance.
- You type exceptionally quickly: 200 words per minute. Assume that the average word has 7 letters, including the space at the end of the word.
- Each key code (each letter, in other words) generates a separate interrupt.

How many interrupts per second would your typing generate on average? Show your work.

200 words/min · 1 min/60 seconds · 7 letters/word · 1 interrupt/character =

23 interrupts/second.

Should the computer use polling or interrupts to handle your fast typing? Explain why your choice is acceptable and the other choice is not. Do not use more than three sentences.

It should use interrupts. Your interrupts cost the computer 23 microseconds out of every second; this is a trivial fraction (23 parts in one million). If the computer used polling, you (the human) would notice the lag and get annoyed.

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 or less, why these optimal policies are impossible to implement.

The policy is defined with reference to the future, and an implementation cannot predict the future.

9. [4 points] Which of the following uses of `mmap()` have we seen in this class?

Circle ALL that apply:

A Mapping a file into a process's virtual memory space.

B Creating a new process that, except for the `rax` register, is an identical clone of an existing process.

C Enabling a process to read and write files using the CPU's load/store instructions rather than `read()`/`write()` system calls.

D Yielding control of the processor to a runnable thread in the same process.

E Enabling a process to copy a file's contents to `stdout` without copying the contents into user space.

A, C, E

10. [2 points] Who is Ken Thompson?

Circle ALL that apply:

A He wrote the first version of Unix.

B He wrote the regulatory report on the Therac-25.

C He described the trusting trust attack.

D He implemented a buggy version of the trusting trust attack.

E He used the trusting trust attack to infect MMUs on computers across the Internet.

A, C, D.

IV File Systems (17 points total)

11. [8 points] This question assumes the programming environment of lab 5. Write a function, called zalloc_block(), that allocates a disk block and then zeroes-out (clears) the block's contents. In the case of an error, the function should simply return. We include some helper functions below. You may not need all of them. You may also need helpers that aren't included. You do not need to document helpers that are part of the distributed lab 5 code base; however, if you use a helper that is not part of that code base, then document *precisely* what that helper does.

Write the code for zalloc_block() below in syntactically correct C. You only need several lines.

```
void zalloc_block()
{
```

```
}

// Searches the bitmap for a free block and allocates it.
// Returns:
//  block number allocated, on success;
// -ENOSPC, if we are out of blocks.
int alloc_block(void);

// Fills the first n bytes of the memory area pointed to by s with the
// constant byte c. Returns a pointer to s.
void *memset(void *s, int c, size_t n);

// Create "path".  On success set *pino to point at the inode and return 0.
// On error return < 0.
int inode_create(const char *path, struct inode **ino);

// Open "path".  On success set *pino to point at the inode and return 0.
// On error return < 0.
int inode_open(const char *path, struct inode **pino);
```

12. [9 points] Consider a file system with the following characteristics. Each inode has 8 data blocks, one indirect block, and one double-indirect block. The file size field in the inode is 64 bits. Each file block is a 4KB disk block, and disk blocks are numbered (addressed) with 4 bytes. The file system is mounted on a disk that has 4 PB (petabytes) of storage. The beginning of the disk contains an array of 2^{32} inodes, each of which can represent a file or be unallocated. The `d_name` portion of a directory entry is exactly 28 bytes.

In this file system, what is the maximum size of a file, in bytes? You can give the answer as a sum of powers-of-2.

$2^{12} \cdot (2^3 + 2^{10} + 2^{10} \cdot 2^{10}) = 2^{15} + 2^{22} + 2^{32}$, or a little more than 4 GB.

In this file system, what is the maximum number of entries in a directory? Again, a sum of powers-of-2 is acceptable.

Directory entries are 32 bytes: 28 bytes for the dname, and 4 bytes (32 bits) for the inode number. Since directories are simply lists of (fname,inum) pairs, the answer is the previous question divided by 32, or: $2^{-5} \cdot (2^{15} + 2^{22} + 2^{32}) = 2^{10} + 2^{17} + 2^{27}$.

V Distributed Systems (11 points total)

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

A

14. [4 points] This question is about the version of the Network File System (NFS) that we studied in class. Recall that in NFS, a client machine, by sending RPCs to a remote server, invokes operations on files that are stored at that server. Which of the following statements are correct?

Circle ALL that apply:

A When the server responds to a client's `write` RPC, all modifications required by that `write` will have made it to the server's permanent storage.

B After an NFS server crashes, and after the operating system subsequently restarts and runs file system recovery, the NFS server must then run a separate recovery procedure to align its state with that of its clients.

C It is possible for an NFS client to send multiple copies of the same RPC to the NFS server.

D The "remove file" RPC is idempotent.

A, C.

The above exercise is derived from one in J. H. Saltzer and M. F. Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, Burlington, MA, 2009. Chapter 4. Available online.

15. [4 points] Consider the two-phase commit (2PC) protocol from class, with a refinement: the workers communicate with one another when they suspect that the coordinator has failed. Now suppose that there is a distributed transaction for which the coordinator decides "commit", then logs COMMIT on its disk, then sends the PLEASE COMMIT to all n workers, and then fails. Assume that the PLEASE COMMIT reached i of the n workers (where $0 \leq i \leq n$). Can the workers apply the transaction without waiting for the coordinator to recover, and if so, what is the minimum value of i for which this is so?

Circle the BEST answer:

A Yes, for $i \geq 1$.

B Yes, for $i \geq 2$.

C Yes, for $i = n$.

D No, because the coordinator might have deadlocked.

E No, because the coordinator might not restart.

A. If even a single worker receives the PLEASE COMMIT, then that worker knows that the coordinator decided to commit, and can tell the other works about that decision.

VI Security, Feedback (10 points total)

16. [5 points] In class, we studied a buggy server that, because of a coding error, is vulnerable to buffer overflow attacks. Below is code for a streamlined version of the server; this version does not give hints to the would-be attacker but is still buggy.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <assert.h>
 5  #include <sys/types.h>
 6  #include <unistd.h>
 7
 8  /*
 9   * Note: assume, as with the demo in class, that file descriptor 0
10   * (which represents stdin) is connected to network input.
11   */
12  void
13  serve(void)
14  {
15      int n;
16      char buf[100];
17
18      memset(buf, 0, sizeof buf);
19
20      /* Read in the length from the client; store the length in 'n' */
21      read(0, (char*)&n, sizeof n);
22
23      /* Now read in n bytes from the client. */
24      read(0, buf, n);
25
26      fprintf(stdout, "you gave me: %s\n", buf);
27      fflush(stdout);
28  }
29
30  int
31  main(void)
32  {
33      serve();
34      return 0;
35  }
```

State the line number of the bug. Then describe the problem, using no more than two sentences.

In line 24, the server reads a number of bytes given by the client without checking that that number is less than or equal to the size of the buffer.

17. [4 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 the setuid bit, and makes a suggestion: “Let us modify the operating system so that, if a non-root process attempts to `exec()` a setuid binary, the operating system performs the `exec()` as if the setuid bit were not set.”

What is the problem with your friend’s suggestion? What would break if we implemented it? Use no more than two sentences.

If we did this, then attempting to run `passwd` at the shell would do the wrong thing: `passwd` would not have the privileges that it needs to modify the password file (the shell itself has to be able to `exec()` setuid programs on behalf of users).

18. [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.

Space for code and/or scratch paper

Space for code and/or scratch paper

End of Final

Congratulations on finishing CS202!

Enjoy the winter break!