New York University

CSCI-UA.0202-003: Operating Systems (Undergrad): Fall 2024

Final Exam (V0)

- **Write your name and NetId on this cover sheet and on the top of every page of the exam.**
- This exam is **105** minutes. Stop writing when "time" is called. *You must turn in your exam; we will not collect them.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 110 minutes after the exam begins and will not accept exams outside the room.
- There are **21** 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 ONE two-sided letter-sized sheet that is written or typed by yourself. **Please write your name on the cheatsheet and turn in your cheatsheet as well.**
- Do not waste time on arithmetic. Write answers in powers of 2 or in fractions if necessary.
- **Solutions will be graded on correctness and clarity.** Partial solutions will be graded for partial 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. Be neat. If we can't understand your answer, we can't give you credit!
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- There are two extra blank sheets of paper at the end of the exam. You may use these to continue answers to any problems that take up more space than is provided, but you must make an indication as to where we can find the rest of your work in order to receive credit.
- **Write your answers only on the front of each page. Do not write on the back of any page. Answers written on the back will not be graded.**

Do not write in the boxes below.

I (xx/32)	II (xx/11)	III (xx/7)	IV (xx/12)	V (xx/16)	VI (xx/12)	VII (xx/10)	Total (xx/100)

Name: Solutions

NetId:

I Short answer

1. [2 points] When implementing a high-performance web server that needs to serve static files to thousands of concurrent users, which system call would you use to efficiently load and share these files in memory across multiple worker processes, while minimizing disk I/O and memory duplication?

mmap

2. [4 points] Why is the block pointer structure in the unix inode sturcture inherently imbalanced? **Your answer should be at most 2 sentences.**

Because most files are small, and having more direct block is faster.

3. [4 points] Consider a directory containing two files (data.txt and script.py) and one symbolic link (shortcut). Running `ls` shows:

```
$ ls
data.txt script.py shortcut
```

However, running `ls -l` displays additional details including permissions and file sizes:

```
$ ls -l
-rw-r--r-- 1 user group 2048 Dec 12 10:30 data.txt
-rwxr-xr-x 1 user group 512 Dec 12 09:15 script.py
lrwxrwxrwx 1 user group 15 Dec 12 08:00 shortcut -> data.txt
```

In two sentences or less, explain why `ls -l` requires more system resources and CPU time compared to the basic `ls` command.

To respond to ls, the file system simply has to find the directory contents and display them. To respond to ls -l, the file system has to fetch the referenced inodes, to learn which are directories, which are symlinks, etc.

4. [2 points] Who is mentioned in the “Reflections on Trusting Trust”. **Circle ONE only:**

A. Linus Torvalds

B. Dennis Ritchie

C. Alonzo Church

D. Alan Turing

B

5. [2 points] In a multi-level paging system like the one used by WeensyOS, what typically happens to the page tables after the `process_setup`? **Circle ONE only:**

A. Only the top-level page table is allocated, and no lower-level page tables are created until memory is accessed.

B. The top-level page table is allocated, and some necessary lower-level page tables are populated to map the process's initial code and data segments, but other portions of the virtual address space remain unallocated until needed.

C. All levels of the page table are fully populated for the entire virtual address space at process start.

D. The process's initial address space is left completely unmapped, and page tables are created only when the process encounters a page fault.

B

6. [2 points] Which factors contribute to context switching overhead?
Circle ALL that apply:

A. Saving and restoring register values

B. Memory allocation

C. TLB flushing

D. Process priority calculation

E. Cache invalidation

A, C, E

7. [2 points] What happens when you delete a file that has $>=1$ hard links?
Circle ALL that apply:

A. All instances of the file are immediately deleted

B. The file is deleted only after its final remaining hard link is removed.

C. The link count decreases by one, but the file data remains until the last link is deleted

D. The file becomes corrupted

E. Only the directory entry is removed while other hard links remain unchanged

B, C

8. [2 points] Which of the following aspects of bootstrapping did we cover in the "putting it together" class? **Circle all that apply:**

A. The BIOS loads the firmware into memory.

B. The firmware initializes hardware components.

C. The firmware executes the bootloader from disk.

D. The bootloader unpacks the kernel image.

E. The kernel decompresses the drivers from the BIOS.

B, D

9. [4 points] This question is about Lab 5. **Circle True or False for each item below:**

True / False In this lab's file system implementation, inodes and data blocks are stored in separate regions on the disk.

False

True / False The bitmap in this file system starts immediately after the superblock at disk block 1.

True.

True / False When using FUSE, the file system implementation must interact directly with the hardware disk.

False.

True / False The layout of the meta-data describing a file in our file system is described by struct inode in `fs_types.h`.

True

10. [4 points] This question is about smashing the stack. **Circle True or False for each item below:**

True / False When a buffer overflow occurs, it always results in a program crash.

False

True / False A stack smashing attack can modify variables in the heap.

False.

True / False Buffer overflows in stack-allocated arrays can potentially overwrite other local variables in the same function.

True.

True / False Writing past the end of an array on the stack can corrupt the stack frame of the calling function.

True

11. [4 points] This question is about setuid programs and related attacks. **Circle True or False for each item below:**

True / False Setting IFS (Internal Field Separator) to include '/' can affect how setuid programs parse paths and filenames.

True

True / False TOCTTOU (Time of Check to Time of Use) attacks do not work if the setuid program checks file permissions before opening files.

False

True / False The real user ID is the privileged version of the effective user ID.

False

True / False A setuid program runs with root privileges if it's owned by root.

True

II C basics

12. [5 points] Examine this program:

```
#include <stdio.h>
void f(int* a, int* b) {
    int temp = *a;
    a = b;
    b = &temp;
    *a = 12;
}

int main() {
    int x = 5, y = 8;
    f(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```

What will this program output?

5, 12

13. [6 points] Write a function that takes a string and returns a new string with its first character repeated n times. The new string should be dynamically allocated. Your code should be syntactically correct and logically complete.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Some helper functions
char* malloc(size_t size);
void free(void* ptr);
size_t strlen(const char* str);            // Returns string length
char* strcpy(char* dest, const char* src); // Copies string from src to dest

char* repeat_first(const char* str, int n) {
    // Your code here


\soln{
 if (str == NULL || str[0] == '\0' || n < 0) {
        return NULL;
    }

    int len = strlen(str);
    char* new_str = (char*)malloc(len + n + 1);
    if (new_str == NULL) {
        return NULL;
    }

    // Fill in repeated character
    for (int i = 0; i < n; i++) {
        new_str[i] = str[0];
    }

    // Copy original string
    strcpy(new_str + n, str);

    return new_str;
}
}

int main() {
    const char* test = "Hello";
    char* result = repeat_first(test, 3);
    printf("%s\n", result);  // Should print "HHHello"
    free(result);
    return 0;
}
```

III I/O

14. [7 points] You are developing a file compression application on a POSIX-like operating system running on a multi-core machine. The application includes a graphical user interface and must handle large files (approximately 1GB each).

Your key requirements are:

1. Concurrent processing: The application should be able to process multiple large files at the same time. By "concurrent," we mean that it should utilize multiple CPU cores and disk operations in parallel, rather than processing files strictly one after another.
2. UI responsiveness: The graphical user interface must remain responsive throughout the compression process, allowing the user to interact with the application, see progress, and potentially cancel operations at any time.
3. Compression ratio: For text files, the compression ratio should meet or exceed 2:1.
4. Performance: The compression should run as fast as possible, making efficient use of available system resources.

(4 points) Assume that you implement all file I/O using only synchronous system calls. Which two of the above requirements would definitely fail under this approach? Explain why, specifically referencing the behavior of synchronous I/O calls.

Requirements 1 and 2 would definitely fail. Requirement 1 fails because each synchronous I/O operation blocks until completion, forcing operations to be serial rather than parallel. Requirement 2 fails because the process enters a sleep state during I/O operations, preventing any UI updates.

(3 points) A junior developer suggests: "Let's create multiple threads, with each thread performing synchronous I/O on a single file. This should solve the problems we identified, right?" Discuss whether this multi-threaded approach fully addresses the issues from part (a).

No, this solution would not fully solve the identified problems. While threading might help with UI responsiveness by moving I/O to background threads, it doesn't achieve true parallel I/O operations because each thread still blocks waiting for its I/O operation to complete, making it a less efficient solution than async I/O.

IV Scheduling

15. [6 points] Consider the relationship between First come first serve (FCFS) and multi-level feedback queue (MLF) scheduling. Multi-level feedback queues have parameters to define the number of queues, the scheduling algorithm for each queue, the criteria to move processes between queues, and when to interrupt running processes. FCFS selects the process that comes first to run next. Given this context:

Can FCFS simulate MLF for all possible parameters of multi-level feedback? Justify your answer briefly.

No. FCFS only considers job length when making decisions and runs jobs to completion. It cannot implement the dynamic behaviors that MLF can achieve, such as moving processes between queues or implementing round-robin with different time quanta.

Can MLF simulate FCFS? Justify your answer briefly.

Yes. Create a single queue using FCFS as its scheduling algorithm, and disable process movement between queues. This effectively makes the MLF behave exactly like a standard FCFS scheduler.

16. [6 points] A system administrator observes that their round-robin scheduler is causing significant overhead and decides to experiment with adaptive time quantum adjustment. They implement a new policy: if a process uses its entire time quantum three times in a row, its next quantum is doubled. If a process yields the CPU before using half its quantum three times in a row, its next quantum is halved. The quantum cannot go below 10ms or above 160ms.

Consider three types of processes running on this system:

– Database processes that typically run for 5ms before waiting for disk I/O

– Web servers that run for around 15ms processing each request

– Video encoders that run continuously for long periods

Explain how the quantum for each type of process would likely evolve over time, and whether this adaptive policy would improve or worsen overall system performance compared to a fixed quantum. Justify your answer by considering both process behavior and system overhead.

This adaptive policy would improve system performance. Here's the evolution and impact for each process type:

Database processes: Would quickly drop to 10ms quantum since they consistently yield early. This is efficient since they don't need long quanta due to frequent I/O waits.

Web servers: Would likely maintain a moderate quantum around 20-40ms since their runtime varies with request complexity. This matches their natural processing cycle.

Video encoders: Would increase to 160ms quantum since they consistently use full time slices. This reduces context switching overhead for these CPU-intensive tasks.

V Trusting Trust

17. [4 points] Consider this C program:

```
int main() {
    char *code = "int main() {\n char *code = %c%s%c;\n printf(code, 34, code, 34);
    \n return 0;\n}";
    printf(code, 34, code, 34);
    return 0;
}
```

What is the output of this program? What key property does this program demonstrate?

The output is exactly the program itself. This is a quine/self-reproducing.

18. [12 points] A malicious programmer modifies a C compiler with this code:

```
if (compiling_login_program) {
    insert_backdoor();
} else if (compiling_c_compiler) {
    insert_this_code();
}
```

When this modified compiler compiles a clean login.c, will the resulting binary contain a backdoor? Why?

- 1 point: Correctly states "Yes" - the binary will contain a backdoor
- 2 points: Explains that the modified compiler inserts backdoor regardless of login.c's content

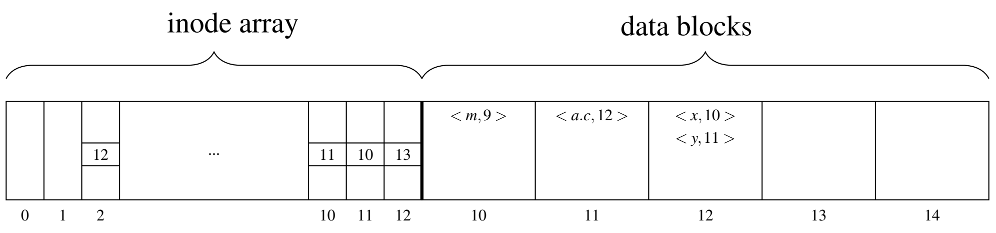
What is the purpose of the second condition (compiling_c_compiler)?

- 2 points: Identifies it's for self-reproduction/propagation or explains it copies the malicious code into new compilers. (1 point for mentioning something similar to self-reproducing in all future generations)
- 2 points: Explains why this is necessary (to ensure future compilations remain compromised)

If we compile a clean compiler.c using the compromised compiler, then use that new compiler to compile login.c - will the login binary contain a backdoor? Explain briefly.

- 1 points: Correctly states "Yes" - the final login binary will contain backdoor
- 2 points: Explains first step: Clean compiler.c $\rightarrow$ malicious compiler (because original compiler inserts malicious code)
- 2 points: Explains second step: New compiler still has both conditions, so when it compiles login.c, it inserts backdoor

VI File System



19. [4 points] Consider the simple file system depicted above, with the following characteristics:

- Each inode contains a single data block pointer, depicted inside the inode structure.
- Note that inodes and data blocks are numbered separately. (The confusion is intentional; we are asking you to think about what the different numbers mean.)
- As mentioned in class, inode 2 contains the inode for the root directory of the file system.

State the contents of the file system in terms of the full path names of files.

/x/a.c, /y/m

20. [8 points] NFS introduces several non-traditional Unix semantics when handling file operations. Select TWO examples below and for each:

(1) Compare how traditional Unix and NFS behave differently

(2) Explain the fundamental reasons why NFS had to deviate from Unix behavior

A. Error returns on successful operations

B. Inode reuse after file deletion

C. Server failure handling

D. File permission changes on open files

A. Error Returns on Successful Operations:

Traditional Unix behavior: In local Unix systems, operations provide clear, immediate success or failure status. If a write() succeeds, the data is guaranteed to be written to the filesystem. Error returns accurately reflect the operation's outcome.

NFS behavior: Operations that appear to succeed might still return errors later. A write() that completes successfully from the application's perspective might fail during network transmission or server processing.

Fundamental reasons: Due to the fact NFS is stateless The stateless nature of NFS means each operation must stand alone, and network/server failures at any point require the client to handle potential errors, even after apparent success.

B. Inode Reuse After File Deletion:

Traditional Unix behavior: When a file is deleted, its inode number isn't immediately reused while processes still have the file open. Local file handles remain valid only for their original file, providing consistent access semantics.

NFS behavior: If a file is deleted and another file is created that happens to reuse the same inode number, cached file handles might now reference a completely different file than intended.

Fundamental reasons: This difference exists because NFS servers handle inode allocation independently and statelessly.

C. Server Failure Handling:

Traditional Unix behavior: On a local Unix system, file operations like open() have deterministic, immediate outcomes - they either succeed or fail instantly based on the file's existence and permissions. The system call will return a clear success/failure status that applications can rely on.

NFS behavior: Applications may hang during file access attempts due to server unavailability. Even simple operations like open() with RD_ONLY can result in indefinite blocking rather than an immediate error when the file doesn't exist.

Why: This deviation occurs because NFS must operate over an unreliable network where the server may be temporarily unreachable. NFS can't distinguish between a file not existing and a temporary server/network failure.

D. File Permission Changes on Open Files:

Traditional Unix behavior: Once a file is opened, subsequent permission changes don't affect the already open file descriptors. Changes only impact future open attempts. This provides consistent access semantics for running programs.

NFS behavior: Permission changes or deletions can cause already open files to fail on subsequent reads, breaking the Unix guarantee of continued access after open().

Why: This difference exists because NFS is stateless - the server doesn't maintain information about which clients have files open.

VII Weensy OS

21. [10 points] Implement the `get_permission_bits` function that determines the permission bits for a given virtual address by walking the `x86-64` 4-level page table.

The function should:

- Walk all levels of the page table
- Track the **accumulated** permissions
- Return -1 if the page is not mapped at any level
- Return the final permission bits if the page is mapped

Here are some helpful macros you might want to use.

```
// Required type definitions
typedef uint64_t x86_64_pageentry_t;  // Type for page table entries
typedef struct x86_64_pagetable {
    x86_64_pageentry_t entry[512];     // 512 entries per page table
} x86_64_pagetable;

// Permission bit flags
#define PTE_P          0x1             // Present
#define PTE_W          0x2             // Writeable
#define PTE_U          0x4             // User-accessible
#define PTE_X          0x8             // Executable

// Helper macros
#define PAGESIZE              4096
#define PAGEOFFSET(addr)      ((addr) & 0xFFF)
#define PAGEINDEX(addr, l)    ((((addr) >> (12 + 9 * (l))) & 0x1FF))
#define PTE_FLAGS(pte)        ((pte) & 0xFFF)          // Extract flags
#define PTE_ADDR(pte)        ((pte) & ~0xFFF)         // Extract address
```

Fill in the function implementation on the next page. Please write syntactically valid C code.

```
// The function need to implement
int get_permission_bits(x86_64_pagetable* pt, uintptr_t va) {
```

```
\soln{
    uint8_t perms = PTE_W | PTE_U | PTE_P | PTE_X;  // Start with all permissions
    x86_64_pageentry_t pe = 0;

    // Walk through all 4 levels
    for (int i = 0; i <= 3 && (perms & PTE_P); ++i) {
        pe = pt->entry[PAGEINDEX(va, i)];
        perms &= PTE_FLAGS(pe);                // Accumulate permissions
        pt = (x86_64_pagetable*) PTE_ADDR(pe);
    }

    // If page is present, return accumulated permissions, else return -1
    return (perms & PTE_P) ? perms : -1;
}
```

```
}
```

Scratch space if needed

Scratch space if needed

End of the Final Exam