New York University

CSCI-UA.0202-003: Operating Systems (Undergrad): Spring 2025

Midterm Exam (V0)

- **Write your name and NetId on this cover sheet and on the top of every page of the exam.**
- **Write your answers only on the FRONT of each page. We do not grade anything written on the back of any page. If an answer box is provided, you have to write your answer inside the box.**
- This exam is **75** minutes. Stop writing when "time" is called. *You must turn in your exam; we will not collect them.* Do not get up or pack in the last 10 minutes. The instructor will leave the room 80 minutes after the exam begins and will not accept exams outside the room.
- There are **20** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. **You want to skim all the 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 the answers in powers of 2 or fractions, if necessary.
- **Solutions will be graded on correctness and clarity.** A response that includes the correct answer, along with irrelevant or incorrect content, will lose points. Be neat. If we cannot understand your answer, we can't give you credit!
- If you find a question unclear or ambiguous, be sure to write down any assumptions you make.
- If the questions impose a sentence limit, we will not read past that limit.
- There are two extra blank sheets of paper at the end of the exam. You may use these to continue answering 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.

Do not write in the boxes below.

I (xx/42)	II (xx/11)	IV (xx/30)	V (xx/12)	Total (xx/95)

Name: Solutions

NetId:

This page is intentionally left blank. Scratch paper can be found at the very back of the booklet.

DO NOT WRITE ON THIS PAGE

I Very short answer

1. [2 points] What specific data structure is used in program execution to isolate local variables with identical names across different function calls? **Provide your answer in one phrase.**

Answer:

Stack frame

2. [2 points] What specific mechanism in x86 is used to make atomic operations like mutex lock implementations possible? **Provide your answer in one phrase.**

Answer:

xchg instruction

3. [2 points] In a 64-bit architecture with 4KB pages, how many bits are typically used for the page offset? **Provide your answer in one phrase.**

Answer:

12 bit

4. [2 points] What specific term do we use to describe a segment of code that accesses shared resources and must not be concurrently executed by more than one thread? **Provide your answer in one phrase.**

Answer:

Critical Section

5. [2 points] Which of the coding standards for concurrent programming by Mike Dahlin does the following code violate?

```
void wait_for_ready() {
    mutex_lock(&mutex);
    while (!is_ready) {
        mutex_unlock(&mutex);
        sleep(1);
        mutex_lock(&mutex);
    }
    mutex_unlock(&mutex);
}
```

Answer (Select one ONLY): [ ]

A `acquire/release` at beginning/end of methods

B Hold lock when doing condition variable operations

C A thread that is in `wait()` must be prepared to be restarted at any time

D don't call `sleep()`

D

6. [2 points] Consider the following code:

```
void f(int* a, int* b) {
  *a = *b;
  b = a;
}
```

If `x=5` and `y=10`, and we call `f(&x, &y)`, what will be the values of x and y after the call?

Answer (Select one ONLY): [ ]

A `x=10, y=10`

B `x=10, y=5`

C `x=5, y=10`

D `x=5, y=5`

A

7. [2 points] Which of the following is a key difference between `fork()` and `pthread_create()`?

Answer (Select one ONLY): [ ]

A Threads created with `pthread_create()` share memory space, while `fork()` creates processes with separate memory

B Those created with `fork()` are faster to create than those with `pthread_create()`

C Threads created with `pthread_create()` cannot communicate with each other

D Process created with `fork()` have better priority inheritance mechanisms

A

8. [4 points] When a user executes an already compiled binary program on a computer system, which of the following components are typically involved in the process of getting that program to run?

Answer (Select ALL that apply): [ ]

A compiler

B interpreter

C loader

D linker

E scheduler

F preprocessor

CE

9. [4 points] Which of the following statements about `fork()` and `exec()` are true?

Answer (Select ALL that apply): [ ]

A `exec()` returns to the parent process once the child process has finished execution

B `fork()` allows the child process to modify the parent's memory after creation

C `exec()` replaces the current process's memory with a new program

D `fork()` creates a new process with a different PID but identical memory contents

E `exec()` creates a new process ID when replacing the program

CD

10. [4 points] Regarding page tables in a virtual memory system, which statements are true?

Answer (Select ALL that apply): [ ]

A Each process has its own page table structure

B Page tables map physical page numbers to virtual page numbers

C In x86-64, page tables have exactly 2 levels

D Page table entries contain permission bits for the page

E TLB caches improve performance by eliminating the need for page tables

AD

11. [4 points] Which of the following statements about multi-level page tables are correct?

Answer (Select ALL that apply): [ ]

A They reduce memory fragmentation compared to single-level page tables

B They reduce the amount of memory needed to represent sparse address spaces

C They increase the speed of virtual-to-physical address translation

D They require fewer memory accesses than a single-level page table

E They make memory sharing techniques like copy-on-write easier to implement

BE

12. [4 points] Regarding race conditions in concurrent programming, which of the following statements are true?

Answer (Select ALL that apply): [ ]

A Race conditions only occur in systems with multiple processors

B A code section can have a race condition even if it's protected by a mutex

C Race conditions can lead to unexpected program behavior depending on thread scheduling

D Race conditions are always detectable at compile time

E Sequential consistency guarantees the absence of race conditions

BC

13. [4 points] Regarding the Therac-25's race condition bug, which statements are true?

Answer (Select ALL that apply): [ ]

A It occurred when operators changed parameters quickly during setup

B It involved shared variables being accessed by multiple concurrent processes

C It was reproducible only in specific sequence and timing conditions

D It was identified by the manufacture company after they received the incident report

E It was caused by priority inversion between system processes

ABC

14. [4 points] Which of the following statements about user mode and kernel mode are correct?

Answer (Select ALL that apply):

A In kernel mode, the CPU can execute any instruction and access any memory address

B The transition from user mode to kernel mode only occurs through system calls

C Exception handlers like page fault handlers always run in kernel mode

D User mode processes can directly access hardware devices by making function calls

E Context switching between processes can be performed entirely in user mode without kernel involvement

AC

II Short answer

Your answer to each of the following questions should be **no more than five sentences**.

15. [4 points] Explain one key problem or limitation with Round Robin scheduling that fair share schedulers aim to solve (and how fair share schedulers solve it).

By periodically boosting the priority of all processes that have been waiting a long time, moving them to higher priority queues regardless of their past CPU usage behavior

16. [4 points] Describe one scenarios where an OS might deliberately trigger page faults as part of its memory management strategy (rather than treating them purely as errors). Briefly explain how the OS uses the page fault mechanism to implement the feature.

When a process first accesses a page marked as not present, the OS triggers a page fault, loads the required page from disk into physical memory, updates the page table, and then resumes execution. This allows the OS to conserve physical memory by only loading pages that are actually needed during program execution.

17. [3 points] Name one part of the lab that you find the most challenging and why.

III Concurrency

18. [15 points] Consider a peer-to-peer file sharing system with N users. Each user has both files to share and wants to download files from others. The system works as follows:

1. When a user wants to download files from another user, they must first establish a connection.
2. A connection can only be established if both users consent to it.
3. To establish consent, User A sends a request to User B, then waits for User B to accept.
4. Due to bandwidth limitations, each user can only maintain one active connection at a time.
5. Once a connection is established, file transfer begins immediately and continues until completion.
6. When transfer completes, the connection is closed and the user becomes available for a new connection.

The system uses a specific protocol where:

- Each user can only be connected to one other user at a time.
- Users never reject connection requests, but they must process them sequentially.
- If a user receives a connection request while already connected to someone else, they queue the request to process later when they become available.
- To prevent redundant downloads, a user will not request a file they are currently in the process of receiving.

Describe the four conditions for a deadlock to occur in this file-sharing context. For each condition, explain how it applies specifically to this system.

Mutual Exclusion:

Each connection is a resource that can only be used by two specific users at a time. The system explicitly states that ”each user can only maintain one active connection at a time,” meaning that connection resources cannot be shared among multiple users simultaneously.

Hold and Wait:

Users can hold their current connection while waiting for another user to become available. For example, if User A is connected to User B and wants to connect to User C, User A must wait until they finish with User B before connecting to User C. Meanwhile, they continue to hold their connection resource with User B.

No Preemption:

Once a connection is established, it cannot be forcibly taken away. The protocol states that ”file transfer begins immediately and continues until completion” and connections are only closed when transfers complete.

Circular Wait:

A circular chain of users can form where each is waiting for the next person in the chain. For example, User A is waiting for User B who is waiting for User C who is waiting for User A. Since each user can only have one active connection, if such a circular dependency forms, none of the users can proceed.

The system designers propose implementing a centralized ”coordinator” that manages all connection requests between users.

How would this coordinator help resolve potential deadlock situations? Describe one specific implementation of the coordinator and explain why it would prevent deadlocks.

Yes. It could use a queue to established the dependencies

Would the coordinator introduce any new problems or limitations to the system? Discuss at least one potential drawback.

Single point of failure, Increase latency, Scalability issues

19. [15 points] In concurrent programming, synchronization is crucial for managing shared resources and ensuring thread-safe operations. There are two main approaches to synchronization:

- **Implicit synchronization**: Uses language-level constructs to automatically handle locking and unlocking of resources.
- **Explicit synchronization**: Requires manual implementation of locking mechanisms using primitives like mutexes, condition variables, and atomic variables.

Here is the implementation of an `ArrayBlockingQueue` using implicit synchronization:

```
class ArrayBlockingQueue {
    private:
        int first = 0, last = 0, count = 0;
        vector<void*> queue;

    public:
        ArrayBlockingQueue(int capacity) {
            if (capacity < 1)
                throw invalid_argument("Capacity must be positive");
            queue.resize(capacity);
        }

    void put(void* o) {
        waituntil(count < queue.size());
        queue[last] = o;
        last = (last + 1) % queue.size();
        count++;
    }

    void* take() {
        waituntil(count > 0);
        void* r = queue[first];
        queue[first] = nullptr;
        first = (first + 1) % queue.size();
        count--;
        return r;
    }
};
```

This implementation uses a special function, `waituntil(condition)`, which blocks the current thread until the specified condition becomes true.

Your task is to reimplement this `ArrayBlockingQueue` using explicit synchronization mechanisms. Please make sure your implementation is as efficient as possible. The relevant function interfaces are in the page after the next. Write your implementation in *syntactically valid C++* and respect *the coding standards*. Please fill in the TODOs in the next page.

```
class ArrayBlockingQueue {
private:
    std::vector<void*> queue;
    int first = 0; int last = 0;
    atomic_int count{0};

    // TODO: declare the mutex and conditional variables




public:
    ArrayBlockingQueue(int capacity) {
        if (capacity < 1) {
            throw invalid_argument("Capacity must be positive");
        }
        queue.resize(capacity);
    }

    void put(void* item) {
        // TODO: fill in below









    }

    void* take() {
        // TODO: fill in below









    }
};
```

Relevant Interfaces

```
// Mutex for basic locking
class mutex {
public:
    void lock();
    void unlock();
};

// Condition variable for thread synchronization
class condition_variable {
public:
    void wait(mutex& lock);
    void notify_one();
    void notify_all();
};

// Atomic integer for thread-safe counter
class atomic_int {
public:
    atomic_int(int desired);
    int load() const;
    void store(int desired);
    int fetch_add(int arg);
    int fetch_sub(int arg);
};
```

Solution:

```
public:
    ArrayBlockingQueue(int capacity) {
        if (capacity < 1) {
            throw std::invalid_argument("Capacity must be positive");
        }
        queue.resize(capacity);
    }

    void put(void* o) {
        putLock.lock();
        while (count.load() == queue.size())
            notFull.wait(putLock);

        queue[last] = o;
        last = (last + 1) % queue.size();
        int c = count.fetch_add(1);
        putLock.unlock();

        if (c == 0) {
            takeLock.lock();
            notEmpty.notify_all();
            takeLock.unlock();
        }
    }

    void* take() {
        takeLock.lock();
        while (count.load() == 0)
            notEmpty.wait(takeLock);

        void* r = queue[first];
        queue[first] = nullptr;
        first = (first + 1) % queue.size();
        int c = count.fetch_sub(1);
        takeLock.unlock();

        if (c == queue.size()) {
            putLock.lock();
            notFull.notify_all();
            putLock.unlock();
        }

        return r;
    }
};
}
```

IV Lab 2: ls

20. [12 points]

In this problem, you will write a C function that counts the total number of directories in a given path and all its subdirectories. Your function should:

1. Count all directories in the given path, including the path itself
2. Recursively count directories in all subdirectories
3. Skip the pseudo-directories '.' and '..'

Please fill in the TODOs on the next page. The information related to all relevant system calls, such as `opendir()`, `readdir()`, and `stat()`, are on the page after the next.

You do not need to worry about error handling. Please make sure you write syntactically valid C.

```
int count_directories(const char *path) {
    DIR *dir;
    struct dirent *entry;
    struct stat stat_buf;
    char full_path[1024];
    // TODO: Your code here









}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    int count = count_directories(argv[1]);
    printf("Total directories: %d\n", count);
    return 0;
}
```

Important Datatypes

```
struct dirent {
    ino_t          d_ino;     /* Inode number */
    off_t          d_off;     /* Offset to the next dirent */
    unsigned short d_reclen;  /* Length of this record */
    unsigned char  d_type;    /* Type of file; not supported by all file system types */
    char           d_name[];  /* Filename (null-terminated) */
};

struct stat {
    mode_t st_mode;     /* File type and mode */
    ino_t  st_ino;      /* Inode number */
    dev_t  st_dev;      /* Device ID */
    uid_t  st_uid;      /* User ID of owner */
    gid_t  st_gid;      /* Group ID of owner */
    off_t  st_size;     /* Size in bytes */
    time_t st_atime;    /* Time of last access */
    time_t st_mtime;    /* Time of last modification */
    time_t st_ctime;    /* Time of last status change */
    /* ... */
};
```

Utility Functions

```
/**
 * Writes formatted output to a string, ensuring it's not larger than size.
 *
 * @param str Buffer to write to
 * @param size Maximum number of bytes to write
 * @param format Format string
 * @return Returns the number of characters
 *         that would have been written if size had been sufficiently large
 */
int snprintf(char *str, size_t size, const char *format, ...);
```

Directory Operations

```
/**
 * Opens a directory stream corresponding to the directory named by name.
 * @param name The path of the directory to open
 * @return On success, returns a pointer to the directory stream.
 *         On error, returns NULL and sets errno appropriately.
 */
DIR *opendir(const char *name);

/**
 * Get the next directory entry in the directory stream
 * @param dirp Pointer to a directory stream obtained from opendir()
 * @return On success, returns a pointer to a dirent structure.
 *         On error or end of directory, returns NULL.
 */
struct dirent *readdir(DIR *dirp);

/**
 * Closes the directory stream associated with dirp.
 * @param dirp Pointer to a directory stream obtained from opendir()
 * @return Returns 0 on success. On error, returns -1 and sets errno appropriately.
 */
int closedir(DIR *dirp);
```

File Status Operations

```
/**
 * Gets information about the file pointed to by pathname
 * and fills in the stat structure.
 * @param pathname Path to the file
 * @param statbuf Pointer to a stat structure where the information is stored
 * @return Returns 0 on success. On error, returns -1 and sets errno appropriately.
 */
int stat(const char *pathname, struct stat *statbuf);

/**
 * Macro that tests if a file is a directory.
 * @param mode The st_mode field from a stat structure
 * @return Returns non-zero if the file is a directory, zero otherwise.
 */
int S_ISDIR(mode_t mode);
```

Solution:

```
syntax: + 1
int count_directories(const char *path) {
    DIR *dir;
    struct dirent *entry;
    struct stat stat_buf;
    char full_path[1024];
    int count = 1;  // Start with 1 to count the current directory

    // Open the directory
    // + 1
    dir = opendir(path);

    // Read each entry in the directory
    // readdir : + 1
    // while: + 1
    // terminate: + 1
    while ((entry = readdir(dir)) != NULL) {
        // Skip '.' and '..' directories
        // + 2
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // Create full path for the entry
        // get full path: + 1
        snprintf(full_path, sizeof(full_path), "%s/%s", path, entry->d_name);

        // Get information about the entry
        // stat: + 1
        stat(full_path, &stat_buf);

        // Check if the entry is a directory
        // check: + 1
        if (S_ISDIR(stat_buf.st_mode)) {
            // Recursively count directories in the subdirectory
            // count: + 1
            count += count_directories(full_path);
        }
    }

    // Close the directory
    // + 1
    closedir(dir);

    return count;
}
```

End of the Midterm Exam

DO NOT WRITE ON THE BACK OF THIS PAGE