

IMPORTANT NOTES for all questions on the exam: For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. You may need to scroll vertically or horizontally to fully view blocks of code. Solutions that violate any specified restrictions may get partial credit. Style is secondary to correctness (e.g. there are no style deductions for using magic numbers). **There is 1 point per minute of the exam.**

1) Multiprocessing Mania 15 Points/180 Total

Consider the following program - assume that the child process spawned always has pid 111 :

```
int main(int argc, char *argv[]) {
    unordered_map<pid_t, int> counters;
    counters[0] = 0;
    cout << counters << endl;

    pid_t pidOrZero = fork();
    counters[pidOrZero] = 4;
    if (pidOrZero == 0) {
        counters[getpid()] = 5;
    }

    cout << counters << endl;

    while (true) {
        pid_t pid = waitpid(-1, NULL, 0);
        if (pid == -1) break;
        else counters[0]++;
    }

    cout << counters << endl;
}
```

A) [5 points] No matter how many times we run the program, the first and last lines printed will always be the same. What are they? (assume it prints in the format `{key: value, key2: value2}`, eg. `{11: 21, 124, 2}`).

B) [5 points] List all possible orderings for this program's output.

C) [5 points] Your friend says the parent process's map should contain the value 5 for key 111 . Are they correct? Explain why or why not.

2) Bridge Crossing

45 Points/180 Total

Caltrans has hired you to build a control system for a bridge. Your program must restrict the number of cars on the bridge in order to conform to a weight limit. You must define a C++ class `Bridge` with a constructor and two methods:

```
Bridge(int limit);  
void arrive(int direction, int weight);  
void leave(int direction, int weight);
```

The `limit` argument to the constructor indicates the weight limit of the bridge, in pounds. When a car arrives at the bridge it will invoke the `arrive` method, which must not return until it is safe for the car to cross the bridge. When the car finishes crossing the bridge, it will invoke `leave`. Cars can travel in two directions on the bridge; the `direction` argument to `arrive` and `leave` will be 0 or 1 to indicate the car's direction. The `weight` argument to `arrive` and `leave` gives the weight of the car, in pounds.

Your solution must satisfy the following requirements:

- The total weight of cars on the bridge must never exceed the weight limit.
- Each direction should be allowed to use half of the total weight limit. However, if the total weight of cars traveling or waiting to travel in one direction does not consume half of the limit, then the other direction may consume the remainder.
- You do not need to ensure fairness among the cars travelling in a given direction, and there will not be a penalty for unnecessary wakeups.
- You may assume that the total weight limit is at least 2x the weight of the heaviest possible car.
- You must write your solution in C++, using locks and condition variables.
- Use the monitor style for your code, as discussed in class and required for the multithreading assignment.
- Your solution must not use busy-waiting.
- Try to make your solution simple and obvious; we will reduce your score by up to 20% if your solution is overly complicated.

```
using namespace std;
class Bridge {
public:
    Bridge(int weight_limit);
    void arrive(int direction, int weight);
    void leave(int direction, int weight);

private:
    mutex lock_;

    // Add any additional instance variables here
};

Bridge::Bridge(int weight_limit) {
    // your code here
}

void Bridge::arrive(int direction, int weight) {
    // your code here
}

void Bridge::leave(int direction, int weight) {
    // your code here
}
```

3) Read-Write Locks

35 Points/180 Total

The read-write lock (implemented by the `rwlock` class) is a `mutex`-like class with three public methods:

```
class rwlock {
public:
    rwlock();
    void acquireAsReader();
    void acquireAsWriter();
    void release();

private:
    // object state omitted
};
```

Any number of threads can acquire the lock as a reader without blocking one another. However, if a thread acquires the lock as a **writer**, then all other `acquireAsReader` and `acquireAsWriter` requests block until the writer releases the lock. Waiting for the write lock will block until all readers release the lock so that the writer is guaranteed exclusive access to the resource being protected. This is useful if, say, you want some kind of mutable data structure that only very periodically needs to be modified. All reads from the data structure require you to hold the reader lock (so as many threads as you want can read the data structure at once), but any writes require you to hold the writer lock (giving the writing thread exclusive access).

The implementation ensures that as soon as one thread **tries** to get the writer lock, all other threads trying to acquire the lock—either as a reader or a writer—block until that writer gets the locks and releases it. That means the state of the lock can be one of three things:

- Ready, meaning that no one is trying to get the write lock.
- Pending, meaning that someone is trying to get the write lock but is waiting for all the readers to finish.
- Writing, meaning that someone is writing.

The private fields within `rwlock` are below - it uses two `mutexes` and two `condition_variable_any`s:

```
class rwlock {
public:
    rwlock();
    void acquireAsReader();
    void acquireAsWriter();
    void release();

private:
    int numReaders;

    // writeState can be either Ready, Pending or Writing
    enum State { Ready, Pending, Writing };
    State writeState;

    mutex readLock;
    mutex stateLock;
    condition_variable_any readCond;
    condition_variable_any stateCond;
};
```

Here is the class implementation:

```

rwlock::rwlock() {
    numReaders = 0;
    writeState = Ready;
}

void rwlock::acquireAsReader() {
    unique_lock<mutex> uls(stateLock);
    while (writeState != Ready) {
        stateCond.wait(stateLock);
    }
    unique_lock<mutex> ulr(readLock);
    numReaders++;
}

void rwlock::acquireAsWriter() {
    stateLock.lock();
    while (writeState != Ready) {
        stateCond.wait(stateLock);
    }
    writeState = Pending;
    stateLock.unlock();
    unique_lock<mutex> ulr(readLock);
    while (numReaders != 0) {
        readCond.wait(readLock);
    }
    writeState = Writing;
}

void rwlock::release() {
    stateLock.lock();
    if (writeState == Writing) {
        writeState = Ready;
        stateLock.unlock();
        stateCond.notify_all();
        return;
    }
    stateLock.unlock();
    unique_lock<mutex> ulr(readLock);
    numReaders--;
    if (numReaders == 0) readCond.notify_one();
}

```

Very carefully study the implementation of the three methods, and answer the questions that appear on the next few pages. Your answers to each of the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying.

A) [5 points] The implementation of `acquireAsReader` acquires the `stateLock` (via the `unique_lock`) before it does anything else, and it doesn't release the `stateLock` until the method exits. Why can't the implementation be this instead?

```
void rwlock::acquireAsReader() {
    stateLock.lock();
    while (writeState != Ready) {
        stateCond.wait(stateLock);
    }
    stateLock.unlock();
    unique_lock lgr(readLock);
    numReaders++;
}
```

B) [5 points] The implementation of `acquireAsWriter` acquires the `stateLock` before it does anything else and it releases the `stateLock` just before it acquires the `readLock`. Why can't `acquireAsWriter` adopt the same approach as `acquireAsReader` and just hold onto `stateLock` until the method returns?

C) [7 points] Notice that we have a single `release` method instead of `releaseAsReader` and `releaseAsWriter` methods. How does the implementation know if the thread acquired the `rwlock` as a writer instead of a reader (assuming proper use of the class)?

D) [7 points] The implementation of `release` relies on `notify_all` in one place and `notify_one` in another. Why are those the correct versions of `notify` to call in each case?

E) [11 points] A thread that owns the lock as a reader might want to upgrade its ownership of the lock to that of a writer without releasing the lock first. Besides the fact that it's a waste of time, what's the advantage of not releasing the read lock before re-acquiring it as a writer, and how could the implementation of `acquireAsWriter` be updated so it can be called after `acquireAsReader` without an intervening release call?

4) Short Answer

33 Points/180 Total

Answer the following short-answer questions below.

A) [5 points] Descriptors can be configured so they are automatically closed when `execvp` is called, and whether or not a descriptor is self-closing on `execvp` boundaries is tracked using just one bit of information. Where is that bit stored? In the descriptor table entry? Or in the open file table entry that's referenced by the descriptor? Briefly defend your answer.

B) [5 points] The `vfork` system call has the same effect as `fork`, except that the child process created by `vfork` cannot modify any variables whatsoever, and the child process must either lead to a call to `_exit` (a special form of `exit`) or one of the `exec*` functions (e.g. `execvp`, `execlp`, `execve`, etc). `vfork` can be used instead of `fork` to boost the performance of time- and resource-sensitive applications, because it doesn't create a new virtual address space for the child until `execvp` is called.

Explain why the implementation of `vfork` (as opposed to `fork`) necessarily suspends the parent process until the child process has either terminated or called one of the `exec*` functions.

C) [5 points] For this part and the next 3 question parts, suppose you are asked to design a new file system that will be used exclusively for storing and playing videos on a new video upload/watching Web site called StanfordTube. Describe the access patterns that you expect to be most common in this file system.

D) [6 points] For the same StanfordTube Web site file system discussed in the previous part, you are given the choice of using flash memory instead of hard disks to store and serve the video data. How would you choose between these two alternatives?

E) [6 points] For the same StanfordTube Web site file system discussed in the previous part, other than the maximum file size, how would you expect the design of this file system to differ from the Unix v6 filesystem design discussed in class, and why?

F) [6 points] For the same StanfordTube Web site file system discussed in the previous part, the project team comes to you with the desire to build a logging mechanism into their system for crash recovery. They let you know that they have ample space for log storage, and want to prioritize minimizing data loss as much as possible. Provide suggestions for what their crash recovery mechanism could look like to work within these constraints / priorities.

5) Thread Dispatching and Scheduling

28 Points/180 Total

Answer the following questions below.

A) [6 points] For each of the following thread state transitions, give a brief example scenario that would cause that transition, or briefly state why that transition is not possible:

1. Running to Ready
2. Ready to Blocked
3. Running to Blocked
4. Blocked to Running

B) [3 points] True or false (explain your answer briefly): the dispatcher is responsible for setting thread priorities in a system that assigns priorities to threads.

C) [6 points] When a page fault occurs, it turns out that the thread triggering that page fault becomes blocked until the page fault is resolved - and the disk operations happen in the background, rather than synchronously as on the virtual memory assignment. This is just like a thread triggering any other I/O operation, such as a thread wanting to read something from disk. When that happens (including when a page fault happens), the triggering thread becomes blocked, the requested disk operation(s) happen in the background, and other threads can get a chance to run. With this in mind, a friend asserts that if a system has lots of runnable threads, it can use them to hide the cost of page faults. Explain whether your friend is right, wrong, or both.

D) [3 points] Your friend suggests a performance improvement for the clock algorithm. Rather than skipping over pages whose reference bit is set, your friend suggests just evicting the first page the algorithm encounters. Is this a good idea or a bad idea, and why?

E) [10 points] In lecture, we saw the pseudocode for a lock implementation that guaranteed a FIFO ordering (meaning that the thread waiting the longest for the lock would be the next one to get it). The pseudocode was presented as follows:

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}

void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

Could we have instead implemented FIFO locks as follows? Why or why not?

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
        locked = 1; // NEW
    }
}

void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
        locked = 0; // NEW
    }
}
```

6) Virtual Memory

24 Points/180 Total

Answer the following questions below.

A) [3 points] True or false (explain your answer briefly): virtual addresses must be same size as physical addresses.

B) [3 points] True or false (explain your answer briefly): page offsets in virtual addresses must be the same size as page offsets in physical addresses.

C) [4 points] One possible design modification to the clock algorithm you implemented on the virtual memory assignment is, if the clock hand encounters a page that is eligible to be kicked out, but it is marked as dirty, the clock algorithm **doesn't kick out the page**, but instead clears the dirty bit and starts writing the page to disk. Explain the benefit of this design change.

D) [6 points] What does it mean for the system behavior as a whole if the clock hand for the clock algorithm is sweeping very slowly over time? What does it mean if the clock hand is sweeping very quickly?

E) [4 points] Give one benefit and one drawback of increasing the size of a memory page.

F) [4 points] One optimization implemented in paging systems is to use a cache to store frequently-accessed virtual to physical translations. In other words, when we must translate a virtual address, first we look in our cache to see if the translation mapping we need is there. If it is, we immediately use the cached physical address. If it's not, we go to the page table and perform the translation, and add it to our cache (potentially kicking out an older entry). There is one of these caches (called a "Translation Lookaside Buffer", or TLB) for the entire system. Explain why adding a TLB necessitates additional work when context-switching to a new process.

Exam Reference Sheet

Filesystem Access

```
// specify one of O_RDONLY, O_WRONLY, O_RDWR in flags
// O_TRUNC clears existing file,
// O_CREAT creates if doesn't exist,
// O_EXCL fails if already exists
int open(const char *pathname, int flags); // returns descriptor
int open(const char *pathname, int flags, mode_t mode); // also sets permissions

int close(int fd); // ignore retval
int dup2(int oldfd, int newfd); // ignore retval
int pipe(int fds[]); // ignore retval
int pipe2(int fds[], int flags); // ignore retval, flags typically O_CLOEXEC
ssize_t read(int fd, void *buf, size_t count); // returns bytes read in buf
ssize_t write(int fd, const void *buf, size_t count); // returns bytes written

#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

Multiprocessing

```
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
int execvp(const char *path, char *argv[]); // ignore retval

#define WIFEXITED(status) // macro
#define WEXITSTATUS(status) // macro
```

Multithreading

```
class thread {
public:
    thread(...); // first argument is function, its args come afterwards
    void join();
};
```

Use **ref()** in the thread constructor to pass args by reference

```
// create unique lock
unique_lock<mutex> myUL(mutexName);
```

```
class mutex {
public:
    mutex();
    void lock();
    void unlock();
};
```

```
class condition_variable_any {
public:
    void wait(mutex& m);
    void notify_one();
    void notify_all();
};
```

C++ Standard Library

```

template <typename T>
class vector {
public:
    size_t size() const;
    void push_back(const T& elem);
    T& operator[](size_t i);
};

template <typename T>
class queue {
public:
    size_t size() const;
    bool empty() const;
    void push(const T& elem);
    T& front();
    void pop();
};

template <typename Key, typename Value>
class unordered_map {
public:
    size_t size() const;
    Value& operator[](const Key& key);
    size_t erase(const Key& key); // ignore retval
};

// iterating through a map
for (auto it = myMap.begin(); it != myMap.end(); ++it) {
    // it->first is key, it->second is value
}
// check if key is in map
if (myMap.find(key) != myMap.end()) {
    // key is in map
}

```