Practice Final Exam Solutions

1. Multiprocessing Mania

Part A: First and Last Lines

First line: {0: 0}

Last line: {0: 1, 111: 4}

Answer key explanation: The very first **cout** statement will always be run by the parent first, because it happens before the child is forked off. The map initially has just one entry in it with key 0, value 0. The very last **cout** statement will always be run by the parent last because it will only run after the child process terminates and is cleaned up. The map at the end in the parent has two entries in it. The 0: 1 comes from the fact that the value for key 0 is incremented inside the while loop when the parent is cleaning up the child process. The 111:4 comes from the line right after the fork, which for the parent will have **pidOrZero** equal to 111.

Part B: Possible Outputs

{0: 0}
{0: 0, 111: 4}
{0: 4, 111: 5}
{0: 4, 111: 5}
{0: 1, 111: 4}
	{0: 0}
	{0: 4, 111: 5}
	{0: 0, 111: 4}
	{0: 4, 111: 5}
	{0: 1, 111: 4}
		{0: 0}
		{0: 4, 111: 5}
		{0: 4, 111: 5}
		{0: 0, 111: 4}
		{0: 1, 111: 4}

Answer key explanation: the parent prints 3 times in total, and the child prints twice. The parent always prints first and last, while the three lines in the middle (2 from chid, 1 from parent) could be reordered, hence the multiple possible outputs. In output #1, the parent prints twice, then the child prints twice, then the parent prints once. In output #2, the parent and the child alternate printing (parent-child-parent-child-parent). In output #3, the parent prints first, the child prints twice, and the parent prints twice.

2. Bridge Crossing

```
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_;

    // Maximum weight the bridge can sustain.
    int capacity;

    // Total weight of cars in each direction on the bridge now
    int current_load_0;
    int current_load_1;

    // Total weight of cars in each direction that are waiting to cross.
    int waiting_load_0;
    int waiting_load_1;

    // Indicates that a car has left the bridge.
    condition_variable_any car_left;
};

Bridge::Bridge(int weight_limit) {
    capacity = weight_limit;
    current_load_0 = 0;
    current_load_1 = 0;
    waiting_load_0 = 0;
    waiting_load_0 = 0;
}

void Bridge::arrive(int direction, int weight) {
    unique_lock<mutex> ul(lock_);
    if (direction == 0) {
        waiting_load_0 += weight;
        while ((current_load_0 + current_load_1 + weight > capacity)
               || ((current_load_0 + weight > capacity / 2) &&
                   (current_load_0 + weight + current_load_1 + waiting_load_1
> capacity))) {
            car_left.wait(ul);
        }
        waiting_load_0 -= weight;
        current_load_0 += weight;
    } else {
        waiting_load_1 += weight;
        while ((current_load_0 + current_load_1 + weight > capacity)
               || ((current_load_1 + weight > capacity / 2) &&
                   (current_load_1 + weight + current_load_0 + waiting_load_0
> capacity))) {
            car_left.wait(ul);
        }

        waiting_load_1 -= weight;
        current_load_1 += weight;
    }
}

void Bridge::leave(int direction, int weight) {
    unique_lock<mutex> ul(lock_);
    if (direction == 0) current_load_0 -= weight;
    else current_load_1 -= weight;
    car_left.notify_all();
}
```

Answer key explanation: First, we notice that the management we are doing is having a car wait until it can cross, and then cross. Whether it can cross is determined by how much weight is currently on the bridge, **as well as** how much weight is in the cars currently waiting in the opposite direction ("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."). Therefore, this means we must store the weight going in each direction and the weight waiting in each direction, as well as the total weight limit.

Like the multithreading assignment, we must think through how this state is updated. All the values are initialized in the constructor, and when a car arrives it should be added to its direction's waiting weight. When a car is free to cross (meaning it's done waiting and is now crossing / will return from **arrive**), we subtract its weight from waiting and add it to the direction's crossing weight. And finally, when **leave** is called, we subtract its weight from its direction's crossing weight.

We must also think about the event(s) that we are waiting on, and when we know those events have happened. Here, the main thing we are waiting on is when there is sufficient capacity to cross. A car arriving will wait for this to happen, and a car leaving will notify that this event has happened since their weight is no longer factored in and other cars may cross. We don't care which direction a car is going in this case, so we can use one condition variable.

A car *cannot* cross (must wait) under the following conditions:

1. the bridge cannot support the car's weight with all other cars currently crossing

2. the bridge can support the car's weight, but the direction they are going in can't accommodate them without exceeding 50% of the capacity, and the other direction doesn't have any weight to spare (its crossing weight + waiting weight + our crossing weight + this car is too much).

3. Read-Write Locks

Part A: acquireAsReader

Assume just two threads:

- Thread 1 calls **acquireAsReader** and is swapped off after five of seven lines.
- Thread 2 calls and progresses through all of **acquireAsWriter**.
- Thread 1 progresses through rest of **acquireAsReader**.

We have one reader and one writer, and that's forbidden.

Part B: acquireAsWriter

If the writer doesn't release **`stateLock`** before waiting for the number of readers to fall to 0, it blocks readers trying to release their locks from decrementing **`numReaders`**.

Part C: release

The implementation is such that the write state can only be **`Writing`** when there's one write lock and zero read locks. When the write state is **`Writing`**, then only one thread could possibly be calling **`release`**, unless the class is being used improperly.

Part D: Upgrade

This question is open-ended and an opportunity to communicate advanced understanding of threading, race conditions, deadlock threat, and concurrency primitives, so a wide variety of answers would be accepted.

One advantage: fewer **`mutex`**es and **`condition_variable_any`**s need to be waited on, so the chance that the threads trying to upgrade the lock are forced to yield the processor is much, much smaller.

Another advantage: using a thread's support for priorities, you can give threads that are trying to upgrade higher priority, so they get the processor before lower priority threads do.

Implementation idea: change the **`acquireAsWriter`** to accept a **`bool`** to state whether it holds a read lock already.

Better implementation idea: update the **`rwlock`** to maintain a set of thread pointers / ids that hold a read lock, and if the thread trying to upgrade finds its thread id in the set, then it knows it's upgrading.

It can then wait until **`numReaders == numUpgraders`** instead of **`numReaders == 0`**.

Couple this with higher thread priorities and you can ensure that exactly one upgrading thread succeeds while all others wait. It's true that all of the other upgraders technically have a read lock, but they're blocked inside **`acquireAsWriter`**, so they're not actually reading whatever data structure is being accessed, so it's okay.)

4. Short Answer

Part A: close-on-execvp

The information must be stored in the descriptor table itself. If stored in the open file table entry, then **`dup2`**'ed descriptors (e.g. **`dup2(fds[1], STDOUT_FILENO`**) would also be auto-closed, since the duplicated descriptors would reference the same entry.

Part B: vfork

Because **`vfork`** doesn't create a new address space, the parent needs to stand still until the child has either exited or been forced to create a new one via **`execvp`**. Otherwise, the parent might change variables the child is permitted to read, and that would introduce a data race.

Part C: StanfordTube access patterns

Possible answer:

- Sequential reads and writes
- Very large file sizes
- Reads much more common than writes

Part D: filesystem design

Some possible answers:

- Larger blocks would make sense since all files are large
- Caching might make sense on a per-file basis, since there will be some videos accessed a lot, and the caching mechanism could try to identify these popular videos.

Part E: logging

Possible answer:

- Metadata and file data logging (takes up more space and slower, but helps minimize data loss)
- No delayed writing, write immediately (slower, but minimizes data loss to disk and to log)

5. Thread Dispatching and Scheduling

Part A: Thread States

Running to ready: time slice expires, thread needs to yield to another ready thread

Ready to blocked: not possible – in order for a thread to become blocked, it must run some operation that causes it to be blocked

Running to blocked: e.g. calls waitpid and child process we are waiting on is not done yet

Blocked to running: e.g. child process we are waiting on finishes, and there is a core immediately available to run on

Part B: Page fault cost

Both. If the working sets of all the runnable threads fit in memory, then while one thread is waiting for a page fault another thread can execute, effectively hiding the cost of page faults. However, if the working sets do not fit in memory, then the system will quickly end up in a state where all threads are waiting for page faults; as soon as a thread resumes after a page fault, it will touch another page that is not in memory and generate another page fault to wait for. The more runnable threads there are, the worst things will get.

Part C: clock modification

Answer: this is a bad idea. This would result in FIFO replacement, whereas the normal clock algorithm provides an approximation to LRU.

Part D: Mutex implementation

No – the resulting implementation is not guaranteed to be FIFO and may allow multiple threads to acquire the lock simultaneously. For instance, if thread A holds a lock and thread B is blocked waiting for that lock, when thread A unlocks, sets **locked = 0** and unblocks B, you could have a scenario where another thread C gets to run and calls **lock** before thread B is woken up to acquire the lock. B could then wake up later and *also* acquire the lock.

6. Virtual Memory

Part A: Address Size

False. The size of physical addresses is determined by the width of page table entries, so it can be either larger or smaller than virtual addresses.

Part B: Offset size

True. The offset is passed directly from the virtual address to the physical address without any mapping or modification, so its size cannot change.

Part C: Clock dirty pages

Possible benefit: we don't need to wait for that dirty page to write to disk before finishing the clock algorithm, but we still write it to disk so that we can potentially kick it out next time when we need another page.

Part D: clock hand speed

Sweeping slowly means plenty of memory, not many page faults – good.

Sweeping quickly means not enough memory, too many page faults – bad.

Part E: Page Size

Possible benefit: smaller page maps because memory is made up of fewer, larger pages.

Possible drawback: increased internal fragmentation from unused space within a page if someone doesn't need the entire page.

Part F: TLB

When context switching, we must mark all the entries in the TLB as invalid because the mappings will be different for the next process.