CS 111 Midterm Examination
Spring Quarter, 2022
Solutions

You have 90 minutes for this examination; the number of points for each question indicates roughly how many minutes you should spend on that question. Make sure you print your name and sign the Honor Code statement below. During this exam you are allowed to consult two double-sided pages of notes that you have prepared ahead of time; other than that, you may not consult books, notes, your laptop or cell phone, or any other materials. If there is a trivial detail that you need for one of your answers but cannot recall, you may ask the course staff for help.

Note: be sure to write only on the front sides of pages. We'll be using Gradescope to grade the exams and may not see information written on the back sides. There are extra pages at the end of the exam, if you need more space for an answer.

I acknowledge and accept the Stanford University Honor Code. I have neither given nor received aid in answering the questions on this examination, and I have not consulted any external information other than two double-sided pages of prepared notes.

_____
(Signature)

_____
(Print your name, legibly!)

_____
(SUNet email id)

Problem	#1	#2	#3	#4	#5	Total
Score						
Max	8	4	10	12	40	74

Problem 1 (8 points)

Indicate whether each of the following statements is true or false, and explain your answer briefly.

(a) Multiple threads can execute in the same critical section simultaneously, as long as they run on different cores.

Answer: false. By definition, a critical section only allows one thread to execute in it at a time. If multiple threads can execute in a critical section concurrently, then the critical section is not properly implemented, and code is likely to break.

(b) MacOS just updated itself last night, and the only change in the update is a new processor scheduling algorithm. As soon as the update installed, one of the applications that you use regularly—Adobe Reader—starts crashing. You should file a bug report with Adobe, not Apple.

Answer: true. If a program is implemented properly, the scheduling algorithm will not affect its behavior (other than possibly its completion time). Thus, there is almost certainly a synchronization bug in Adobe Reader that was triggered by the new scheduling algorithm.

(c) The 4.4 BSD scheduler (the one that measures CPU usage for each thread) is non-preemptive.

Answer: false. If a high-priority thread (i.e. one whose recent CPU utilization is low) wakes up, it will immediately preempt a lower-priority thread that is currently executing.

(d) In Project 3, the object file `party_test.o` contains code that invokes the `meet` method of the `Party` class; the code for the `Party` class is in object file `party.o`. Because of this, there will be a symbol table entry for the `meet` method in `party_test.o`.

Answer: false. Symbol table entries exist in the file where a particular name is defined (`party.o` in this case). The file `party_test.o` will contain an unresolved reference record that describes its usage of the name `meet`.

Problem 2 (4 points)

(Note: this project is no longer part of the class)

In Project 1, when the shell starts a new command, it creates a new process for that command, with a single thread. Suppose that instead of creating a new process with `fork` and `execvp`, it created a new thread within its own process to run the command: give two reasons why this would be a bad idea.

Some possible answers:

- **The shell process doesn't include the code for the child, so it would somehow have to load that dynamically to run the child as a thread.**
- **The shell and the child would end up sharing all their data, which could lead to corruption and/or security leaks.**
- **I/O redirection wouldn't work: there would only be one file descriptor 0, but it would need to refer to different files for the shell and each child.**

Problem 3 (10 points)

The very first priority-based scheduler we discussed in lecture makes use of several round-robin queues—on the order of 5 or 6—to manage all threads eligible for processing time and schedule them to execute in an order that leads to the shortest average completion time, or something very close to it.

a) (2 points) What heuristic does the OS use to identify threads as CPU-bound?

Answer: a CPU-bound thread will use up its entire time slice without blocking.

b) (4 points) An I/O-bound thread, when eligible for CPU time, typically occupies a spot in the highest priority queue. What types of I/O-bound threads allow for CPU-bound ones executing with lower priority to get processor time? In what situations would a CPU-bound thread be more likely to starve?

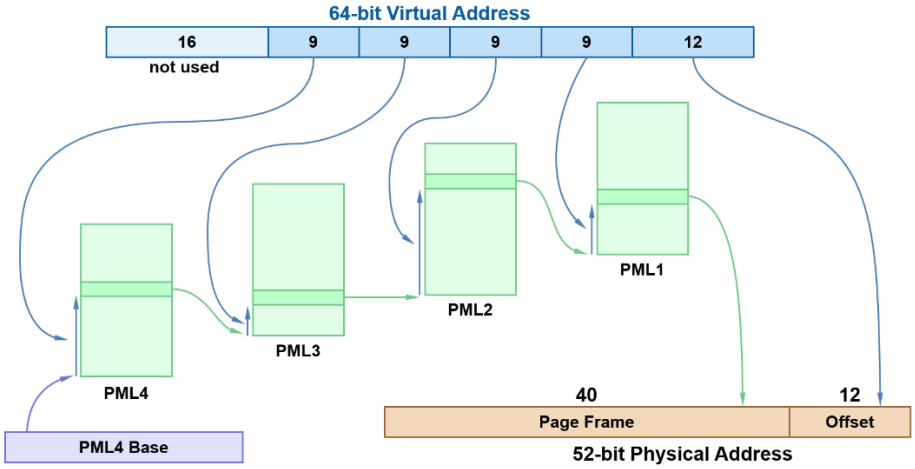
Answer: threads that spend most of their time blocked on I/O (such as emacs waiting for many seconds for keystrokes) consume almost no CPU time, so they allow lower-priority CPU-bound threads to get processor time. Threads that perform I/O but don't block for long and perform significant amounts of computation, such as video streaming clients, could continually return to the highest-priority queue and starve CPU-bound threads.

c) (4 points) Multi-queue schedulers typically set the time slice duration for lower priorities to be higher. For example, the time slice duration granted to threads with the highest priority might be, say, 4ms, whereas the time slice duration granted to threads running with the second highest priority might be 8ms (third-highest priority threads? 16ms, etc.) Give two reasons why higher-priority threads should be granted smaller time slices whereas lower-priority ones should be granted larger time slices.

Answer: I/O-bound threads typically don't need much computation between I/O operations, so short time slices work well for them. Short slices for the highest priority queues also make it easier to detect CPU-bound threads. Long slices are better for CPU-bound threads because (a) it wastes less time in timer interrupts and (b) it increases the likelihood that one of the CPU-bound threads will actually finish (FIFO scheduling is better than round-robin when there are multiple long-running threads).

Problem 4 (12 points)

The figure below shows the address translation mechanism used for 64-bit mode in Intel x86 processors, which was discussed in lecture. It uses 4 levels of page map to translate 48 bits of virtual address to 52-bit physical addresses.

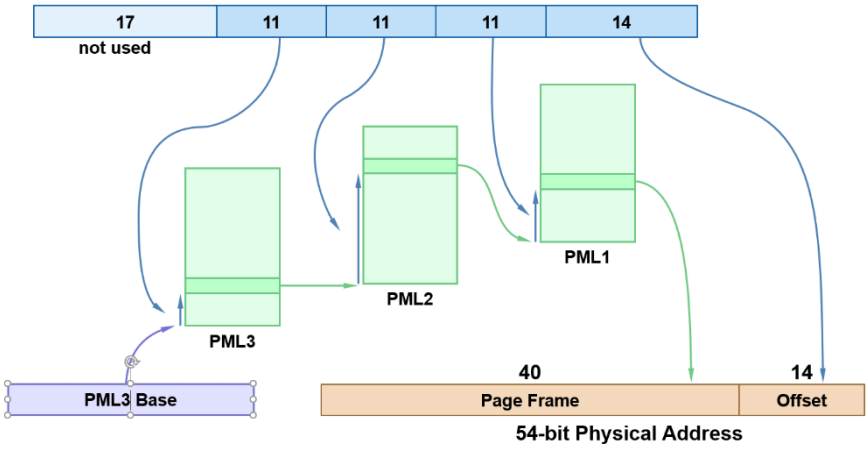


(a) (2 points) How many bytes are in each page map entry in this scheme (not all bits of a page map entry are necessarily used)?

Answer: each page map occupies one page, which is 4096 (2^{12}) bytes, and it holds 2^9 (512 entries). Thus, each entry must be 2^3 (8) bytes long. This was intended to be an easy warm-up question, but it seemed to cause quite a bit of confusion!

(b) (10 points) A friend comes to you and claims that the number of levels of page map could be decreased by increasing the page size. Specifically, they claim that if the page size is increased by a factor of 4x, only 3 levels of page map will be needed to translate 48-bit virtual addresses. Is your friend right? Redraw the mechanism above with 4x larger pages and only 3 levels of page map; how many total bits of virtual address does the new mechanism translate/use?

Answer: each page map will now hold 4x as many entries, so it can translate 11 bytes of virtual address instead of 9. And, offsets will increase from 12 bits to 14, yielding the following mechanism:



This translates only 47 bits, not 48, so your friend is wrong. Close, but no cigar!

Problem 5 (40 points)

Because of construction, a multi-lane highway must merge to a single lane. Caltrans has hired you to automate the merge point. To do this, you must define a C++ class `Merge` with a constructor and two methods:

```
Merge(int num_lanes);
void wait(int lane);
void merged();
```

The `num_lanes` argument to the constructor indicates the total number of lanes that will be merging to a single lane. When a car arrives at the merge point it will invoke the `wait` method; the `lane` argument indicates the lane number in which the car arrives (this value will always be between 0 and `num_lanes-1`, inclusive). The `wait` method must not return until it is this car's turn to pass the merge point into the single lane. Once the car has finished merging into the one lane, it will invoke the `merged` method; this indicates that this car has finished merging and it is safe for the next car to begin merging. The car threads coordinate among themselves using these two methods: there is not a separate flag-person thread.

Your solution must meet the following requirements:

- There is a single outgoing lane, and only one car can merge at a time (i.e., only one car can have returned from `wait` but not have called `merged` yet).
- If there are many cars waiting, they must take turns in order of lane (one car from each lane with a waiting car must merge before a second car merges from any lane).
- There will not necessarily be cars waiting in all (or any) lanes at any given point in time.
- Unlike a real highway, **you do not need to guarantee perfect FIFO order** for the cars in a single lane.
- 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 Project 3.
- Your solution must not use busy-waiting and must not allow starvation (you may assume that locks and condition variables will not starve threads).
- Try to make your solution simple and obvious; we will reduce your score by up to 20% if your solution is overly complicated.

```
class Merge {
    public:
        Merge(int num_lanes);
        void wait(int lane);
        void merged();
    private:
    // Monitor-style lock.
    std::mutex mutex;

    // Number of cars waiting in each lane (has num_lanes entries).
    std::vector<int> waiting;

    // Total number of waiting cars in all lanes.
    int totalWaiting;
```

```
    // Cars in lane i wait for entry i in this vector.
    std::vector<std::condition> safe_to_merge;

    // The last lane that merged.
    int prev_lane;

    // True means a car is merging (wait has returned but merged hasn't been
    // called yet).
    bool merging;
};

Merge::Merge(int num_lanes)
    : mutex(), waiting(), totalWaiting(0), safe_to_merge(), prev_lane(0),
    merging(false)
{
    for (int i = 0; i < num_lanes; i++) {
        waiting.emplace_back(0);
        safe_to_merge.emplace_back();
    }
}

void Merge::wait(int lane)
{
    std::unique_lock<std::mutex> lock(mutex);
    if (!merging) {
        merging = true;
        return;
    }
    waiting[lane]++;
    totalWaiting++;
    safe_to_merge[lane].wait(lock);
}

// Solution continues on next page.
```

```
void Merge::merged()
{
    std::unique_lock<std::mutex> lock(mutex);
    if (totalWaiting == 0) {
        merging = false;
        return;
    }
    while (true) {
        prev_lane = prev_lane + 1;
        if (prev_lane >= waiting.size()) {
            prev_lane = 0;
        }
        if (waiting[prev_lane] > 0) {
            waiting[prev_lane]--;
            totalWaiting--;
            safe_to_merge[prev_lane].notify_one();
            break;
        }
    }
}
```