CS 111 Midterm Examination #1
Spring Quarter, 2021
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. While taking this exam, you may not consult any materials other than information you have explicitly prepared ahead of time. If there is a trivial detail that you need for one of your answers but cannot recall, you may ask the course staff. Fill in the fields of this PDF file, then submit it at www.gradescope.com.

By filing this examination electronically, you acknowledge and accept the Stanford University Honor Code, and you affirm that you have neither given nor received aid in answering the questions on this examination.

____

(Name)

____

(SUNet email id)

Problem	#1	#2	#3	#4	#5	#6	Total
Score							
Max	10	3	6	10	10	40	79

Problem 1 (10 points)

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

(a) Deadlock cannot occur over preemptible resources.

Answer: true. A thread need not wait indefinitely for a preemptible resource, since it can be taken away from its current owner and given to the waiting thread, without negatively impacting the current owner.

(b) If a slab-based storage allocator is used by a program that allocates and frees objects of only two sizes A and B, fragmentation cannot occur.

Answer: false. Suppose that a large number of objects of size A is allocated and then almost all of them are freed, leaving one live object in each slab. The free space in those slabs cannot be used for objects of size B. This unusable space is fragmentation.

(c) In systems that use dynamic linking and shared libraries, every application must have a private copy of the code that fills in the jump table at startup.

Answer: true. If the code that fills in the jump table (the dynamic loader) were itself in a shared library, there would be no way to fill in the jump table entries for it.

(d) With base-and-bound relocation, it is possible to share code between processes.

Answer: false. At most two processes could share the code (one with the code at the bottom of its virtual address space in the other with the code at the top), but they would require different virtual addresses for code and data. It would be very difficult to write code that would actually work under these conditions.

(e) Round-robin scheduling cannot lead to starvation.

Answer: true: all ready threads get equal amounts of CPU time under a round-robin scheduler.

Problem 2 (3 points)

Consider a program consisting of three source files: `a.cc`, `b.cc`, and `c.cc`. These files have been compiled separately into object files `a.o`, `b.o`, and `c.o`. File `a.cc` contains the implementation of a function `f1`. That function is called twice in `b.cc` and once in `c.cc`. For each of the three object files, how many unresolved references will there be that refer to `f1`?

Answer: there will be 0 unresolved references in `a.o`, 2 in `b.o`, and 1 in `c.o`.

Problem 3 (6 points)

A trap instruction (such as a system call) transfers control from a user program to the operating system, and it does two things: it branches to an address in the kernel, and it overwrites the processor status (PS) register with a new value. Both the branch address and the new PS value are specified in advance by the kernel.

(a) Why is it necessary to change the PS register when trapping into the kernel?

Answer: when the kernel executes, a special bit must be set in the PS, which allows the kernel to perform operations not permitted to user programs. This bit must be turned off whenever user-level code is executing.

(b) A friend claims that it is unnecessary to do both things in a single hardware instruction, and that using separate instructions for branching and overwriting the PS would work just as well. Explain why your friend's approach wouldn't work.

Answer: if the instructions were separate, a user program could potentially execute the instruction to overwrite the PS without executing the branch instruction. This would give the user program kernel privilege, which it could then abuse.

Problem 4 (10 points)

Consider the following code:

```
int x = 0;
...
std::thread t1 = new std::thread([&x] {
    int y = computeY();
    x += y;
    long_running_func_to_do_other_stuff();
}
usleep(100000);          /* Delays for 100 milliseconds */
std::thread t2 = new std::thread([&x] {
    int z = computeZ();
    x = 2*x - y;
    another_long_running_func();

}
t1.join();
t2.join();
std::cout << “The value of x is “ << x << std::endl;
```

The goal of this code is to serialize the updates to `x` (wth `t1` running first), while still allowing the two long-running functions to execute in parallel. Is the value that will be printed for `x` deterministic? If so, explain why; if not, explain why and modify the code to fix the problem. You may assume that `computeY` and `computeZ` are deterministic.

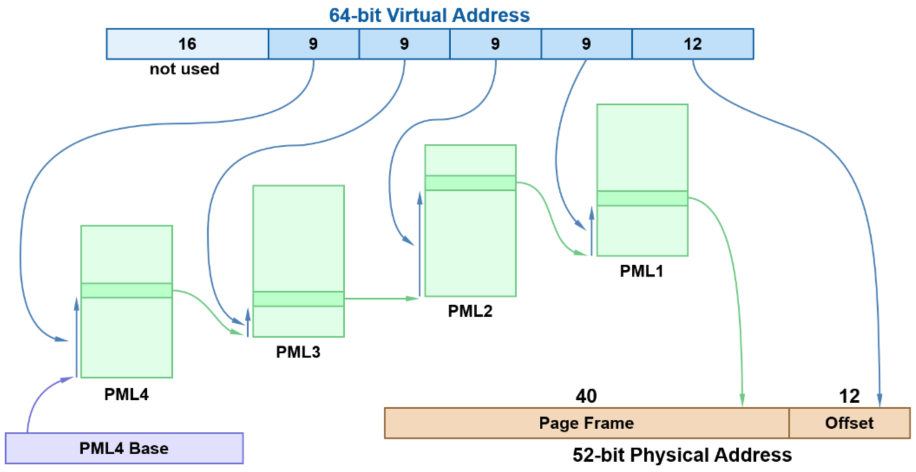
Answer: there is a race condition in the code. Delaying for 100 ms will not guarantee that `t1` modifies `x` before `t2`: it's always possible that the scheduler could decide not to execute `t1` for a long time, so that `t2` ends up executing first, or even concurrently.

The solution is to add explicit synchronization, such as this:

```
int x = 0;
std::mutex m;
std::condition t1_modified_x;
int mod_done = 0;
...
std::thread t1 = new std::thread([&x] {
    int y = computeY();
    x += y;
    m.lock();
    mod_done = 1;
    t1_modified_x.notify_one();
    m.unlock();
    long_running_func_to_do_other_stuff();
}
m.lock();
while (!mod_done)
    t1_modified_x.wait();
m.unlock();
std::thread t2 = new std::thread([&x] {
    int z = computeZ();
    x = 2*x - y;
    another_long_running_func();
}
t1.join();
t2.join();
std::cout << “The value of x is “ << x << std::endl;
```

Problem 5 (10 points)

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



The discussion in lecture showed how to use this structure for a simple process containing one page of code, one page of data, and one page of stack; the scheme shown in lecture required seven page maps. However, it is possible to execute the simple process with fewer than seven page maps.

(a) What is the smallest number of page maps that could be used for the simple process?

Answer: 4

(b) How would those page maps be used for the process's code, data, and stack? Note: your solution must use separate pages for code, data, and stack.

Answer: there would be one each of PML4, PML3, PML2, and PML1, corresponding to the lowest virtual addresses. The layout of code, data, and stack would be identical to what was discussed in class, except that the stack would be repositioned in virtual memory. Rather than starting at the highest address of virtual memory and growing downwards, it would start at the highest virtual address that can be referenced from the lowest PML1 (2^{21}-1, or `0x1fffff`) and grow downwards from there.

(c) What is the disadvantage of this approach?

Answer: there would be less room for stack growth before it collided with the heap.

Problem 6 (40 points)

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 Project 3.
- 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.

Additional working space for Problem 6:

```
class Bridge {
public:
    Bridge(int capacity);
    void arrive(int direction, int weight);
    void leave(int direction, int weight);

    std::mutex mutex;

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

    // Total weight of cars in each direction that are currently
    // on the bridge.
    int current_load[2];

    // Total weight of cars in each direction that are waiting.
    int waiting_load[2];

    // Indicates that a car has left the bridge.
    std::condition_variable car_left;
};

Bridge::Bridge(int capacity)
    : mutex()
    , capacity(capacity)
    , current_load({0, 0})
    , waiting_load({0, 0})
{
}

void Bridge::arrive(int direction, int weight)
{
    std::unique_lock<std::mutex> lock(mutex);
    int other_dir = 1 - direction;
    waiting_load[direction] += weight;
    while (((current_load[0] + current_load[1] + weight) > capacity)
            || (((current_load[direction] + weight) > capacity/2)
            && ((current_load[direction] + weight
            + current_load[other_dir]
            + waiting_load[other_dir] > limit)))) {
        car_left.wait(lock);
    }
    waiting_load[direction] -= weight;
    current_load[direction] += weight;
    return;
}

void Bridge::leave(int direction, int weight)
{
    std::unique_lock<std::mutex> lock(mutex);
    current_load[direction] -= weight;
    car_left.notify_all();
}
```

Additional working space for any problem, if needed