CS 140 Midterm Examination
Winter Quarter, 2012

You have 1.5 hours (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 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, or your laptop. 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.

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)

_____SOLUTION SET_____
(Print your name, legibly!)

_____
(email id, for sending score)

Problem	#1	#2	#3	#4	#5	Total
Score						
Max	12	10	12	15	35	84

Problem 1 (12 points)

Answer each of the following questions briefly (in one or two sentences).

(a) Under what conditions does FIFO scheduling result in the shortest possible average response time?

Answer: if the jobs happen to arrive in the ready queue with the shortest completion times first (or, as a special case, if all jobs have the same completion time).

(b) Under what conditions does round robin scheduling behave identically to FIFO?

Answer: if the job lengths are no longer than the length of the time slice.

(c) Under what conditions does round robin scheduling perform poorly compared to FIFO?

Answer: if the job lengths are all the same, and much greater than the time slice length.

(d) In the situation you described in part (c) above, does reducing the time slice for round-robin scheduling help or hurt its performance relative to FIFO? Why?

Answer: reducing the time slice increases the average response time. It causes all the jobs to finish closer and closer to the end time: if there are N jobs all the same length L, the first job will finish at approximately N*L – (N-1)*T, where T is the length of the time slice. The smaller that T becomes, the later the first job finishes.

Problem 2 (10 points)

Suppose two threads execute the following C code concurrently, accessing shared variables `a`, `b`, and `c`:

Initialization

```
int a = 4;
int b = 0;
int c = 0;
```

Thread 1

```
if (a < 0) {
    c = b - a;
} else {
    c = b + a;
}
```

Thread 2

```
b = 10;
a = -3;
```

What are the possible values for `c` after both threads complete? You can assume that reads and writes of the variables are atomic, and that the order of statements within each thread is preserved in the code generated by the C compiler.

Answer: 4, 7, 13, 14, -3

Problem 3 (12 points)

Below are several actions that must be carried out to implement dynamic linking for shared libraries. For each of the actions indicate whether the action is carried out by the **compiler**/assembler, **linker**, operating system **loader** (the part of the OS that loads a process into memory and starts its first thread running), or a runtime **library** that is statically linked into the application. If multiple components share responsibility for the action, explain what each one does.

(a) Choose the memory address of the jump table.

Answer: linker and loader (the linker determines the virtual address, and the loader determines the physical address)

(b) Fill in the jump instructions in the jump table.

Answer: runtime library

c) Choose the memory address for the shared library code.

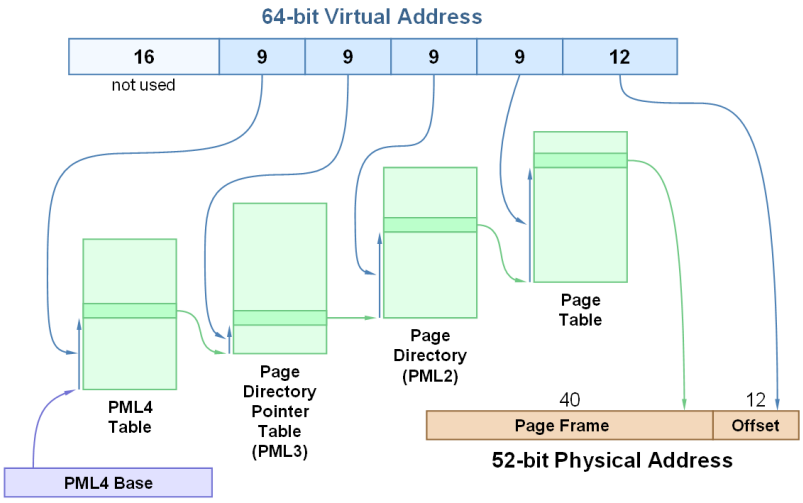
Answer: runtime library

d) Fill in the instructions that jump into the jump table.

Answer: compiler and linker

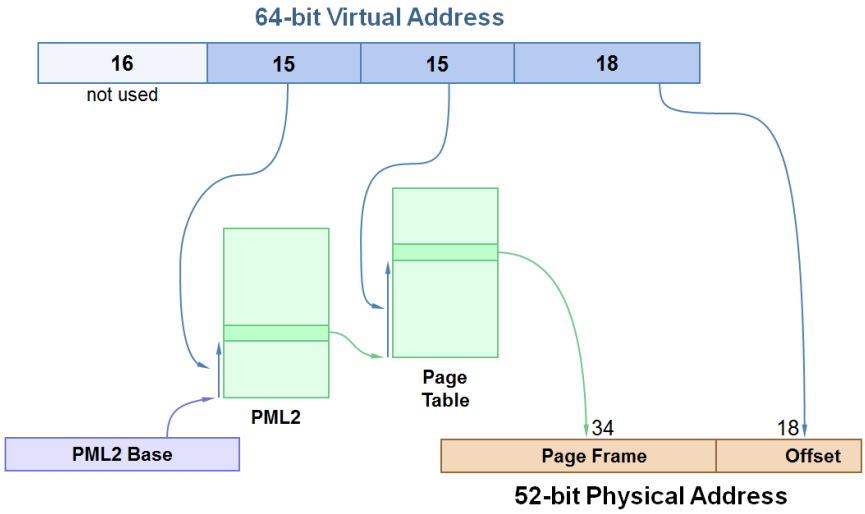
Problem 4 (15 points)

The figure below shows the address translation mechanism used for 64-bit mode in Intel x86 processors. Starting with a 48-bit virtual addresses, it uses 4 levels of page table to translate the address to a 52-bit physical address. Each page table entry is 8 bytes long.



If the page size is increased, the number of levels of page table can be reduced. How large must pages be in order to translate 48-bit virtual addresses with only 2 levels of page table? Draw the address translation mechanism corresponding to this page size.

Answer: 256 KB (2^{18} bytes) per page



Problem 5 (35 points)

You have been hired by Caltrans to automate the flow of traffic on a one-lane bridge that has been the site of numerous collisions. Caltrans wants you to implement the following rules:

- Traffic can flow in only a single direction on the bridge at a time.
- Any number of cars can be on the bridge at the same time, as long as they are all traveling in the same direction.
- To avoid starvation, you must implement the "five car rule": once 5 or more consecutive northbound cars have entered the bridge, if there are any southbound cars waiting then no more northbound cars may enter the bridge until some southbound cars have crossed. A similar rule also applies once 5 or more consecutive southbound cars have entered the bridge.

You must implement the traffic flow mechanism in C by defining a structure `struct bridge`, plus five functions described below.

When a northbound car arrives at the bridge, it invokes the function

```
bridge_arrive_north(struct bridge *b)
```

This function must not return until it is safe for the car to cross the bridge, according to the rules above. Once a northbound car has finished crossing the bridge it will invoke the function

```
bridge_leave_north(struct bridge *b)
```

Southbound cars will invoke analogous functions `bridge_arrive_south` and `bridge_leave_south`.

Use the next pages to write a declaration for `struct bridge` and the four functions above, plus the function `bridge_init`, which will be invoked to initialize the bridge

- You must write your solution in C using the Pintos functions for locks and condition variables:

```
lock_init (struct lock *lock)
lock_acquire(struct lock *lock)
lock_release(struct lock *lock)
cond_init(struct condition *cond)
cond_wait(struct condition *cond, struct lock *lock)
cond_signal(struct condition *cond, struct lock *lock)
cond_broadcast(struct condition *cond, struct lock *lock)
```

  Use only these functions (e.g., no semaphores or other synchronization primitives).
- You may not use more than one lock in each `struct bridge`.
- Your solution must not use busy-waiting.
- If your southbound functions are identical to the northbound functions except that `north` is replaced with `south` (and vice versa) in all identifiers, then you can omit the southbound functions and just circle this bullet point.

This page is for your solution to Problem 5.

```
struct bridge {
    int north_waiting;
    int north_crossing;
    int north_consecutive;
    int south_waiting;
    int south_crossing;
    int south_consecutive;
    struct lock lock;
    struct condition northbound_done;
    struct condition southbound_done;
}

void bridge_init(struct bridge *b)
{
    b->north_waiting = 0;
    b->north_crossing = 0;;
    b->north_consecutive = 0;
    b->south_waiting = 0;
    b->south_crossing = 0
    b->south_consecutive = 0;
    lock_init(&b->lock);
    cond_init(&b->northbound_done);
    cond_init(&b->southbound_done);
}

int bridge_arrive_north(struct bridge *b)
{
    lock_acquire(&b->lock);
    b->north_waiting++;
    while ((b->south_crossing > 0) ||
            ((b->south_waiting > 0) && (b->northConsecutive >= 5)) {
        cond_wait(&b->southbound_done, &b->lock);
    }
    b->north_waiting--;
    b->north_crossing++;
    b->northConsecutive++;
    b->southConsecutive = 0;
    lock_release(&b->lock);
}

int bridge_leave_north(struct bridge *b)
{
    lock_acquire(&b->lock);
    b->north_crossing--;
    if (b->north_crossing == 0) {
        cond_broadcast(&b->northbound_done, &b->lock);
    }
    lock_release(&b->lock);
}
```

Additional working space for Problem 5, if needed.

```
int bridge_arrive_south(struct bridge *b)
{
    lock_acquire(&b->lock);
    b->south_waiting++;
    while ((b->north_crossing > 0) ||
            ((b->north_waiting > 0) && (b->southConsecutive >= 5)) {
        cond_wait(&b->northbound_done, &b->lock);
    }
    b->south_waiting--;
    b->south_crossing++;
    b->southConsecutive++;
    b->northConsecutive = 0;
    lock_release(&b->lock);
}

int bridge_leave_south(struct bridge *b)
{
    lock_acquire(&b->lock);
    b->south_crossing--;
    if (b->south_crossing == 0) {
        cond_broadcast(&b->southbound_done, &b->lock);
    }
    lock_release(&b->lock);
}
```