Objective

A few sample long-form questions to help students prepare for the cs 423 midterm. Although this document does not include MCQs—because the in-class pop-quizzes provide many samples—the midterm will contain MCQs. In general, these sample questions are at least as difficult as (if not more difficult than) the ones in the mid-term. Ditto for the pop-quiz MCQs.

Virtualization (10 points)

Consider a system with a 2-level page table that uses demand paging; in other words, there is no speculative prefetching of pages. The system handles TLB misses in hardware (hardware managed TLB) but page faults are handled by the OS. A user process attempts to execute a single `load` instruction (let us pretend the instruction has already been somehow fetched from physical memory): `MOV EAX, [0x7FF00123]`

In the initial state, the TLB does not have the mapping for the VPN of `[0x7FF00123]`, the top level page of the page table (aka page directory) is in physical memory, the pointer to page directory is in a register, but the leaf page with the PTE and the page itself that correspond to `[0x7FF00123]`are in swap storage (i.e. each of their present bit = 0).

1. What is the minimum number of page faults that need to be serviced before the value from `[0x7FF00123]` is loaded into the `EAX` register? Briefly mention what is loaded into physical memory during each fault.
2. It takes 5ns to consult the TLB, 100ns for a CPU/core to load <=4096 bytes from RAM, and 20ms to read <=4096 bytes from swap storage into RAM. Ignoring the time it takes to execute instructions (of the page fault handler, the switch to kernel mode, etc.), how long does it take for this instruction to load the value from `[0x7FF00123]` is loaded into the `EAX` register? Explain your reasoning. Assume the page size is 4KiB.
3. Redo Q2 with the assumptions that the physical memory is full, victim pages are evicted on-demand (no page/swap daemon in this system), 40% of the victim pages selected are dirty, and it takes 50ms to write <=4096 bytes to swap storage.
4. Looking at the answers to the 2 previous questions, what can you say about the need for swap/page daemons?

Concurrency (10 points)

I am asked to implement the condition variable primitive `my_cond_t` using only mutexes and the `park` & `unpark` system calls. Here's my attempted solution:

```
typedef struct node {
```

```
    int tid;
    struct node *next;
} node_t;

typedef struct {
    node_t *head;
    node_t *tail;
    mutex_t queue_lock; // protect head/tail pointers
} my_cond_t;

void cond_wait(my_cond_t *c, mutex_t *user_lock) {
    node_t my_node;
    my_node.tid = gettid(); // get my own thread id
    my_node.next = NULL;

    // 1. Enqueue node
    mutex_lock(&c->queue_lock);
    if (c->tail == NULL) {
        c->head = c->tail = &my_node;
    } else {
        c->tail->next = &my_node;
        c->tail = &my_node;
    }
    mutex_unlock(&c->queue_lock);

    // 2. Release the user lock
    mutex_unlock(user_lock);

    // 3. Go to sleep
    park();

    // 4. Re-acquire the user lock
    mutex_lock(user_lock);
}

void cond_signal(my_cond_t *c) {
    mutex_lock(&c->queue_lock);
    if (c->head != NULL) {
```

```
        node_t *to_wake = c->head;
        c->head = c->head->next;
        if (c->head == NULL)
           c->tail = NULL;

        // Wake up the thread
        unpark(to_wake->tid);
    }
    mutex_unlock(&c->queue_lock);
}
```

1. My function `cond_wait()` contains a catastrophic memory bug. Explain it. What happens when `cond_signal()` eventually trips on it?
2. Edit this code to fix this bug.
3. Even after you fixed the bug in Q1, my code has a race condition that can cause signals to be lost. Explain the race condition with an example trace with two threads.
4. Where exactly should you insert a `setpark()` call to fix this race condition?
5. Implement a `signal_broadcast()` function to this code. Bonus points for implementing an efficient version of it (Hint: Consider lock contention).

MPs

1. You should have noticed one big difference between programming in kernel space vs in user space, i.e., lack of isolation. Can you explain this difference in 2-3 sentences? (2 points)
2. Why does MP1 recommend using `copy_{from|to}_user()` within the kernel to copy data from/to user space memory? Why not use `memcpy`? (2 points)
3. In MP2, you used `kmem_cache` to allocate (and free) nodes in the process list. Describe the biggest advantage and biggest disadvantage with using `kmem_cache`. (2 points)