Harvard University
CS161 Midterm Exam
Spring 2006

March 17, 2007

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible points</th>
<th>Actual points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1: General OS Questions (2 pts per question - 20 pts total)

1a. List two kinds of scheduling algorithms that do NOT necessarily strive to eliminate starvation.

Reason #1:

Reason #2:

1b. List one advantage and one disadvantage of user-level threading.

Advantage:

Disadvantage:
1c. GWAOS v2.3.0.2.4.2.1 supports a single thread synchronization primitive, called **compare-and-swap** (CAS). Compare-and-swap is an atomic operation, provided by the hardware, with the following pseudocode:

```c
int compare_and_swap(int *a, int old, int new) {
    if (*a == old) {
        *a = new;
        return 1;
    } else {
        return 0;
    }
}
```

Implement the code for a simple spinlock using compare-and-swap. You are *not* allowed to assume any other hardware or kernel support exists (e.g., disabling interrupts).

```c
struct lock {
    /* Fill in */
}

void acquire(struct lock *lock) {
    /* Fill in */
}

void release(struct lock *lock) {
    /* Fill in */
}
```
1d. Patrick found an old MULTICS system in the basement of the Science Center and is trying to get it up and running. His major problem has been figuring out how to set up segment ring brackets to achieve the desired protection. Assume two processes, A and B, with privilege levels of 2 and 5, respectively. The kernel runs with privilege level 0. Define the ring brackets (W, R, and G) for the following segment types:

- Code segment readable and executable (but not writeable) by process A, and not callable by lower-privileged processes:
  \[ W: \quad R: \quad G: \]

- Shared file readable and writeable by process A, but only readable by process B:
  \[ W: \quad R: \quad G: \]

- Shared library executable by process A, and callable by process B but not by lower-privileged processes:
  \[ W: \quad R: \quad G: \]

- Kernel segment readable, writeable, and executable by kernel only, but callable by all processes:
  \[ W: \quad R: \quad G: \]

1e. The best synchronization primitive for implementing reader-writer locks is:

(a) Locks
(b) Semaphores
(c) Condition variables
(d) Test-and-set

Why? Provide a short (two sentence at most) explanation here:
1f. What is the difference between starvation and deadlock? A short answer (two sentences tops) should suffice.

1g. Grand Chau Chow Restaurant in Chinatown is short on chopsticks, and to save money, decided to leave only one chopstick in between every two plates at the table. To avoid the Dining Philosopher’s Problem, the manager proposed the following solution: diners sitting at a table are required to pick up one chopstick at a time, but must do so in alphabetical order by their last name. (To break ties, two diners with the same last name must play rock-paper-scissors until one of them wins.) Does this solve the problem? Why or why not?

1h. Under what conditions must the TLB be flushed? Assume that TLB entries are not tagged by process ID (that is, two entries in the TLB collide if they have the same virtual address). Circle all that apply.

(a) CPU interrupt

(b) System call

(c) Return from system call

(d) Copy-on-write
1i. PatTel is coming out with its latest line of processors for high-end computing (and by this we do not mean gaming). One complaint from customers was that the 32-bit address space limited physical RAM sizes to just 4 GB. PatTel decided to introduce a new feature that would support 36-bit physical addresses. However, because of budget limitations, they did not want to reorganize the entire CPU design. Without changing the page table format or the basic paging data structures used by the processor, how could PatTel’s new CPU support a 36-bit physical address space? A short answer (one sentence) should suffice.

1j. The main purpose of gate segments in MULTICS is (circle one):

(a) Prevent user processes from calling higher-privileged code

(b) Allow high-privilege code to set up specific entry points

(c) To support multiple protection rings on processors with only two privilege levels

(d) To implement read-only shared libraries
Problem 2: MLFQ scheduling (25 pts)

McCollumOS X is a new operating system for Macintosh computers marketed by (who else) but the software giant McCollumSoft. Unfortunately, now that Apple has decided to shift its Mac line to the Intel processor, sales of McCollumOS X have dropped off considerably. In a bid to recover some of the lost market, McCollumSoft decided to implement a fast, simple scheduler in the latest OS version (codenamed “Squid”) using multilevel feedback queues.

To keep the implementation lean and fast, the scheduler only supports two priority levels, PH and PL. Priority level PH (high priority) has a CPU time quantum of 5 msec, while priority level PL (low priority) has a time quantum of 20 msec. If multiple threads are runnable in the same priority level, they are scheduled in round-robin fashion.

In addition, McCollumSoft implemented a very important optimization: if a high-priority thread becomes runnable while a low-priority thread is running, it does not preempt the currently-running thread. That is, preemption only occurs if a thread’s CPU time slice expires. When a process moves from the blocked to the ready state, it is added to the end of the appropriate ready queue. However, if two processes become runnable at the same time (at the same priority level), they are placed in the ready queue in order of their process ID.

Assume three processes that each run in a loop with the following characteristics:

- Process A: CPU burst of 10 ms, followed by an I/O of 20 ms.
- Process B: CPU burst of 5 ms, followed by an I/O of 30 ms.
- Process C: CPU burst of 25 ms, followed by an I/O of 5 ms.

Simulate the operation of the McCollumOS X scheduler with the above processes. Assume that all processes start out in the PL ready queue in the order \{A, B, C\}, and that the scheduler will run the thread at the head (front) of the queue.
2a (15 pts). Fill out the following table showing the state of the scheduler for each iteration that it runs. Only run the scheduler for 100 ms. The table should have one row per execution of the scheduler.

1. The time that the scheduler runs.

2. The state of the PH and PL ready queues before the scheduler picks the next thread to run. List the processes in order with the process at the head of the queue first on the list. In parenthesis, list the amount of CPU burst that the process has left. (e.g., A(10) means that process A has a burst of 10 ms.)

3. The set of sleeping processes;

4. The thread chosen by the scheduler to run next, and for how long.

<table>
<thead>
<tr>
<th>Time</th>
<th>PH ready queue</th>
<th>PL ready queue</th>
<th>Waiting</th>
<th>Next proc to run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>A(10), B(5), C(25)</td>
<td>-</td>
<td>A for 10 ms</td>
</tr>
</tbody>
</table>
2b (5 pts). Fill out the following schedule showing the state of each process at each time step. Run the scheduler for 100 msec in total. In each entry of the table, write:

1. **R** if the process is running;
2. **W** if the process is waiting (blocked on I/O);
3. **PL** if the process is ready to run in priority level PL;
4. **PH** if the process is ready to run in priority level PH.

<table>
<thead>
<tr>
<th>time (ms)</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
<th>65</th>
<th>70</th>
<th>75</th>
<th>80</th>
<th>85</th>
<th>90</th>
<th>95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc A</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proc B</td>
<td>PL</td>
<td>PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proc C</td>
<td>PL</td>
<td>PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2c (1 pt) What is the CPU utilization achieved by the McCollumOS X scheduler?

2d (2 pts) What is the average waiting time for processes under this schedule?

2e (2 pts) What is the average response time for processes under this schedule?
Problem 3: Concurrency (25 points)

Popper’s Bistro is the latest hot spot in the South End. They are very busy on weekend nights and have the following policy for reservations. Someone can phone ahead and put their name on a waiting list which is serviced in FIFO order as tables become available. (Assume all tables seat 4 people and there is always 4 people in a party.) A party needs to be at the restaurant in order to be seated, of course. If a party is not at the restaurant when their table comes up, other parties behind them on the list will be seated first. Parties showing up at the restaurant without calling first will simply be placed at the end of the list.

In addition, certain VIPs (Red Sox players, members of Aerosmith, and Harvard Computer Science professors) are given priority at the restaurant and will be seated as soon as a table becomes available after they arrive at the restaurant. VIPs do not need to call ahead.

Danny Popper, the owner, wanted to test out this policy in a computer simulation before implementing it in “real life” at the restaurant. In order to do this he wrote the following program that simulates the dining parties, reservations agent, and seating host.

```c
/* Represents a table at the restaurant */
typedef struct {
  boolean available; /* Is table free? */
} table;
table table_list[MAX_TABLES]; /* List of all tables */
lock table_list_lock; /* Mutex on table list access */
/* A thread will wait() on this condvar for a table to become ready. */
condvar table_ready = new condvar(table_list_lock);

/* Represents a dining party’s entry on the waiting list. */
typedef struct {
  boolean party_arrived; /* Has party arrived at restaurant? */
  /* Used by dining party thread to wait for table to become open. */
  lock table_ready_lock;
  condvar table_ready = new condvar(this.table_ready_lock);
  table ourtable;
} party;
party waiting_list[MAX_PARTIES]; /* Waiting list */
lock waiting_list_lock; /* Mutex on waiting list access */

/* Mutex on the front door of the restaurant -- used when a party arrives. */
lock front_door_lock;
condvar party_arrived = new condvar(front_door_lock);

/* Mutex on the reservations phone */
lock phone_call_lock;
condvar phone_call = new condvar(phone_call_lock);
```
/* Code page 2... */

/*/ Thread representing a dining party. */

**dining_party() {**
   /* Call for a reservation */
   lock(phone_lock);
   signal(phone_call);
   unlock(phone_lock);
   sleep(some random time);

   /* Show up at restaurant */
   lock(front_door_lock);
   signal(party_arrived);
   unlock(front_door_lock);

   /* Wait to be seated */
   lock (this.table_ready_lock);
   wait(this.table_ready);
   unlock (this.table_ready_lock);

   /* Sit down and start eating */
   sleep(random time for eating); // Perform I/O - ha ha ha

   /* Free up the table and leave */
   lock (table_list_lock);
   this.ourtable.available = TRUE;
   unlock (table_list_lock);
   exit();
}**

/*/ Thread representing the reservations agent. */

**reservations() {**
   while (1) {
      lock(phone_lock);
      wait(phone_call);
      lock(table_list_lock);
      Count number of empty tables and tell to customer, to give
         estimate of wait time;
      lock (waiting_list);
      Push party onto end of waiting_list;
      unlock (waiting_list);
      unlock(table_list_lock);
      unlock(phone_lock);
   }**
}**
/* Code page 3... */
/* Thread representing the seating host at the entrance of the restaurant.
*/
host() {
  while (1) {
    lock(front_door_lock);
    wait(party_arrived);
    unlock(front_door_lock);

    lock(waiting_list_lock);
    if (party is VIP) {
      lock(table_list_lock);
      wait(table_ready);
      table = first available table;
      assign party to table;
      unlock(table_list_lock);
      continue;
    }
    if (party is on waiting list) {
      waiting_list_entry.party_arrived = TRUE;
    } else {
      push new party onto end of waiting_list;
      waiting_list_entry.party_arrived = TRUE;
    }
    unlock(waiting_list_lock);

    wait(table_ready);

    lock(table_list_lock);
    table = first available table;
    lock(waiting_list_lock);
    party = find first party on waiting_list where party has arrived already;
    remove party from waiting list;
    party.ourtable = table;
    table.available = FALSE;
    unlock(table_list_lock);

    lock(party.table_ready_lock);
    signal(party.table_ready);
    unlock(party.table_ready_lock);
    unlock(waiting_list_lock);
  }
}
Problem 3a (20 pts). There are at least four synchronization bugs in this implementation. For each bug, label it with a number (1, 2, etc.) in the code, and write a short description of the bug below. *Trivial syntax bugs in the code do not count!*

2 points for correctly identifying each bug; 3 points for description of the bug.

1.

2.

3.

4.
Problem 3b (5 pts). Rewrite the code to `reservations()` to correct one of these bugs, while maintaining the same functionality:
Problem 4: Virtual Memory Management (30 points)

In this question you will act as the MMU for a simple virtual memory architecture. (A very slow MMU, to be sure, but we won't hold that against you.) This processor has a 16-bit address space, and each address accesses a single 8-bit byte. A two-level page table scheme is used with 16 entries in the top-level page table and 256 entries in the second-level page table. Each page table entry is two bytes wide and has the following format:

<table>
<thead>
<tr>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Readable</td>
<td>Writeable</td>
<td>Executable</td>
<td>Frame number</td>
</tr>
</tbody>
</table>

Page table entries are stored in memory in big endian order.

For top-level page tables, the R, W, and X bits are unused and should be set to zero. For second-level page tables, the readable (R) bit indicates whether the page can be read by the process; the writeable (W) bit indicates whether the page can be written by the process; and the executable (X) bit indicates whether the page can be used to fetch instructions for execution.

A partial listing of the machine’s physical memory is shown below. (For convenience we are showing the contents of memory four bytes at a time, although the machine is byte-addressed. The first byte in each entry is the lowest-order byte. For example, the value of physical address 0x01 in memory is 0x0b.)
Question 4a (1 pt). What is the size of each page in bytes?

Question 4b (1 pt). What is the maximum size of the physical memory in bytes?

Question 4c (8 pts - 0.5 pts per entry). Assume the current process’s top-level page table starts at physical address 0x100. List the contents of the top-level page table here. (Hint: How can the R, W, and X bits in the top-level page table tell you that you are decoding the memory correctly?)

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Question 4d (20 pts - 4 pts per entry)**. Translate each of the following virtual addresses into a physical address. (If you want us to follow your answer, it is probably a good idea to clearly write down the primary and secondary page number in the address, the corresponding page table entry, and so forth.)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x702d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x60bb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x27f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>