

Your name:

1

Harvard University
CS161 Midterm Exam
Spring 2006

March 17, 2007

SOLUTION SET: DO NOT DISTRIBUTE

Your name:	
Your student ID:	

Please write your name on the top of each page of this exam.

This is a closed book examination. You have **80 minutes** to answer as many questions as possible. There are **18 pages** to this exam (make sure you have all of them) with a total of **100 points**. Write all of your answers directly on this paper. Make your answers as concise as possible. If there is something in the question that you believe is open to interpretation, you must state your assumptions clearly in your answer. **Please make sure we can tell how you reach your answers, so we can give you partial credit even if the final answer is incorrect.**

Problem	Possible points	Actual points
Problem 1	20	
Problem 2	25	
Problem 3	25	
Problem 4	30	
TOTAL	100	

Your name:

2

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: **Real-time scheduling in which some processes have scheduling deadlines.**

Reason #2: **Priority-based scheduling in which some processes are more important than others.**

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

Advantage: **Can be more lightweight than kernel-level threading; does not require complex kernel support; small per-thread state**

Disadvantage: **Kernel does not know about multiple threads in a process; cannot schedule multiple threads across separate CPUs; thread blocking stalls entire process**

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:

```
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).

```
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: R: G:

W:1 R:2 G: 2

- Shared file readable and writeable by process A, but only readable by process B:

W: R: G:

W:2 R:5 G:5

- Shared library executable by process A, and callable by process B but not by lower-privileged processes:

W: R: G:

W:2 R:2 G:5

- Kernel segment readable, writeable, and executable by kernel only, but callable by all processes:

W: R: G:

W:0 R:0 G:7

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: **Answer is (b) Semaphores. Reason: Semaphores can maintain a counter of the number of reader threads in the reader-writer lock.**

1f. What is the difference between starvation and deadlock? A short answer (two sentences tops) should suffice.

Starvation implies that a thread cannot make progress because other threads are using resources it needs. Starvation can be recovered from if, for example, the other processes finish. Deadlock is a circular wait without preemption that can never be recovered from.

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?

No – this does not solve the Dining Philosopher's problem. If diners sit down in a circle ordered by their last name, they can still enter into a deadlock situation.

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

Answer: (d) Copy-on-write – the old TLB entry will point to the old physical address.

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.

Easiest answer: Change the size of each page to be larger: for example, from 4 KB to 64 KB. More complex answers: Introduce extra registers or somehow separate physical processes into different physical address ranges.

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

Answer: (b)

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.

Time	PH ready queue	PL ready queue	Waiting	Next proc to run
0	-	A(10), B(5), C(25)	-	A for 10 ms
10	-	B(5), C(25)	A	B for 5 ms
15	-	C(25)	A, B	C for 20 ms
35	A(10)	C(5)	B	A for 5 ms
40	-	C(5), A(5)	B	C for 5 ms
45	B(5)	A(5)	C	B for 5 ms
50	C(25)	A(5)	B	C for 5 ms
55	-	A(5), C(20)	B	A for 5 ms
60	-	C(20)	A, B	C for 20 ms
80	A(10), B(5)	-	C	A for 5 ms
85	B(5), C(25)	A(5)	-	B for 5 ms
90	C(25)	A(5)	B	C for 5 ms
95	-	A(5), C(20)	B	A for 5 ms
100	-	C(20)	A, B	C for 20 ms

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.

<i>time (ms)</i>	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Proc A	R	R	W	W	W	W	PH	R	PL	PL	PL	R	W	W	W	W	R	PL	PL	R
Proc B	PL	PL	R	W	W	W	W	W	W	R	W	W	W	W	W	W	PH	R	W	W
Proc C	PL	PL	PL	R	R	R	R	PL	R	W	R	PL	R	R	R	R	W	PH	R	PL

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

100%, since the CPU is used the whole time.

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

Waiting time is the time that a process spends in the WAIT state. Proc A wait time = 40 ms; Proc B wait time = 70 ms; Proc C wait time = 10 ms. Total wait time = 40 + 70 + 10 ms = 120 ms. Average over 3 processes = 120/3 = 40 ms.

2e (2 pts) What is the average response time for processes under this schedule?

Response time is the time that a process spends in the READY state. Proc A ready time = 30 ms; Proc B ready time = 15 ms; Proc C ready time = 35 ms. Total ready time = 30 + 15 + 35 = 80 ms. Average over 3 processes = 80/3 = 26.666 ms.

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 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.

```

/* 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. **dining_party() forgets to signal table_ready condvar when leaving the table.**
2. **host() thread needs to wait for either a party entering the restaurant or a table becoming available. The way the loop in host() is written, a party has to come in for anyone on the waiting list to be seated.**
3. **Deadlock with reservations() and host() – both threads grab the table_list_lock and waiting_list lock, but they do so in different orders, leading to deadlock.**
4. **host() does not lock table_list_lock before waiting on it.**

Your name:

14

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

Answer: Don't hold lock for table list and waiting list at the same time, or lock them in the other order, to avoid the deadlock with `host()` thread.

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:

<i>1 bit</i>	<i>1 bit</i>	<i>1 bit</i>	<i>1 bit</i>	<i>12 bits</i>
Valid	Readable	Writeable	Executable	Frame number

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.)

Your name:

0x00	da 0b 6a ff	0xa0	6b d5 1f 4c	0x140	13 ef 40 4e
0x04	9f 07 7a 8e	0xa4	93 12 4c 1f	0x144	f4 a1 2c 6a
0x08	42 47 b8 2e	0xa8	72 be bc b4	0x148	ed 6e 6e 5c
0x0c	b4 38 60 85	0xac	f2 bc 4e 89	0x14c	ce a2 8b 8f
0x10	0f 1e 21 9e	0xb0	58 19 a0 2d	0x150	58 ae fe b5
0x14	e3 8e 2b 36	0xb4	7f f2 ab 03	0x154	1f a3 0d 82
0x18	f1 98 97 6e	0xb8	1a 01 d2 e1	0x158	1a af 41 78
0x1c	c6 17 d2 41	0xbc	84 01 4b 80	0x15c	e5 9e cb 97
0x20	34 bc e0 07	0xc0	19 ba 67 3f	0x160	fd cd ee f6
0x24	c0 12 d0 04	0xc4	78 82 75 95	0x164	1d e0 0e 18
0x28	50 11 f0 00	0xc8	84 52 00 08	0x168	da 4c 36 a2
0x2c	0b 3f 30 1a	0xcc	30 b5 17 dd	0x16c	49 0c 20 9d
0x30	11 81 d0 72	0xd0	f2 ba 51 0a	0x170	8b 42 47 f3
0x34	b6 ff f2 4c	0xd4	bc 04 ed 19	0x174	ac 3e e7 89
0x38	53 f0 13 31	0xd8	57 84 dd 3f	0x178	78 49 5d ba
0x3c	e6 30 d2 71	0xdc	5a 2a 3d 4e	0x17c	34 43 4d 8e
0x40	f3 ce a4 28	0xe0	b9 a7 89 d8	0x180	c8 96 d8 40
0x44	16 31 61 7c	0xe4	6c d2 d8 65	0x184	56 78 69 ea
0x48	96 7b d6 5b	0xe8	3a b3 87 81	0x188	90 37 16 89
0x4c	38 ab 42 06	0xec	58 ea f6 78	0x18c	f2 89 34 f1
0x50	e3 bb ef e6	0xf0	d6 94 50 24	0x190	19 e4 26 16
0x54	c0 5a 18 25	0xf4	1a f4 90 74	0x194	17 90 91 a0
0x58	64 0f aa ac	0xf8	c5 ee 69 84	0x198	1d ad 95 de
0x5c	0b a1 c6 d4	0xfc	b8 ba a8 92	0x19c	af 87 3d d7
0x60	95 95 b9 be	0x100	80 04 80 01	0x1a0	fe 42 dd 41
0x64	72 d2 07 b6	0x104	00 10 00 00	0x1a4	52 d2 0b 3a
0x68	c7 1a a3 ca	0x108	00 03 80 07	0x1a8	44 06 28 55
0x6c	0b c2 01 f1	0x10c	00 a7 80 02	0x1ac	31 f3 b4 ee
0x70	c9 f7 f8 88	0x110	80 03 00 41	0x1b0	31 56 3d 23
0x74	cc 12 c2 50	0x114	00 16 80 06	0x1b4	8c d1 19 c6
0x78	6b f7 30 7b	0x118	00 03 80 00	0x1b8	fa 69 8b 2f
0x7c	05 ca 78 2b	0x11c	80 05 00 09	0x1bc	75 16 cb f7
0x80	ad 6c 01 22	0x120	89 2d c6 d5	0x1c0	32 5a 6d 2c
0x84	44 16 6e 48	0x124	b2 d3 53 58	0x1c4	73 23 20 04
0x88	02 ce 70 3e	0x128	8a be 9b 09	0x1c8	02 55 bb 8d
0x8c	5b fd ab 63	0x12c	e1 7a d9 12	0x1cc	62 ee 09 20
0x90	3a d1 2b ca	0x130	22 9c 18 67	0x1d0	62 db d1 5c
0x94	d0 df 45 e6	0x134	b5 37 23 8f	0x1d4	2b b7 42 2b
0x98	b0 d7 14 06	0x138	b6 94 c6 d8	0x1d8	eb 3c 2f ec
0x9c	2d 15 31 1a	0x13c	cc f7 3f e8	0x1dc	2c 0f 44 29

Question 4a (1 pt). What is the size of each page in bytes?

Answer: with 16 entries in the top level and 256 entries in the 2nd level, that's 4 bits for the primary page number and 8 bits for the secondary page number, leaving 4 bits for the offset, for a total page size of 16 bytes.

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

Answer: The PFN is 12 bits wide, so we can have up to 4096 pages, of 16 bytes each = 65536 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?)

Index	Valid	PFN
0	V	0x4
1	V	0x1
2	-	0x10
3	-	0x0
4	-	0x3
5	V	0x7
6	-	0xa7
7	V	0x2
8	V	0x3
9	-	0x41
10	-	0x16
11	V	0x6
12	-	0x3
13	V	0x0
14	V	0x5
15	-	0x9

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.)

Virt addr	Valid?	Readable?	Writeable?	Executable?	Physical address
0x702d	V	R	-	-	0x012d
0x60bb	-	-	-	-	invalid
0x27f3	-	-	-	-	invalid
0x7006	-	-	-	-	invalid
0x7015	V	R	W	-	0x0075