CSC 369H1 S
OPERATING SYSTEMS

UNIVERSITY OF TORONTO
Spring 2007

Midterm Test

NO AIDS ALLOWED

Please **PRINT** in answering the following requests for information:

Family Name: ___________________________________

Given Names: ___________________________________

Student Number: |_ _ _| |_ _ _| |_ _ _|

Login (@cdf): _____________________

Notes to students:

1. This test lasts for 50 minutes and consists of 46 marks. Budget your time accordingly.
2. This test has 5 questions and 8 pages (including this one and an extra space page at the back); Check that you have all pages before starting.
3. **Write in pen. No pencils. Really, I mean it.**
4. Write your answers on this "question and answer" paper, in the spaces provided. Be concise. In general, the amount of space provided is an upper bound on the "size" of answer that is expected. If necessary, use space where available and provide explicit pointers.
5. State your assumptions and show your intermediate work, where appropriate.
6. Do **not** go beyond here until instructed to do so. Write your student number at the top of each succeeding page once you get going.

Question	Marks
1	/10
2	/6
3	/ 10
4	/ 10
5	/ 10
Total	/ 46

../continued

1. [10 marks] Short Answer

(a) A running process may stop execution for several reasons. Briefly describe two circumstances where the process is **not** moved to the ready state when it stops running.
Three possibilities are (only two were needed):
i) thread made exit() system call and is moved to the Exit state when it stops running.
ii) thread is waiting (e.g. for device on I/O operation, or for synchronization object like a lock) and is moved to Blocked state when it stops running.
iii) thread was killed by parent OR had a fatal exception and was killed by OS, moving it to Exit state when it stops running.

(b) In the OS/161 process id management subsystem, the process table stores pointers to pidinfo structs which hold status information about each thread. Explain why a separate pidinfo structure is used, rather than storing the status information in the thread struct.
Status info, specifically the "exited" state and exit code must be retained after thread exits in case the parent wants to retrieve the exit code (i.e., calls thread_join, perhaps via waitpid). If we use a separate data structure, only a small amount of space is needed for this state while the much larger thread structure can be deleted.

(c) How is a system call different from a normal C language function call?
Normal function calls transfer control (with a branch) directly to the destination and execute in the same address space. System calls transfer control indirectly using special hardware support, including a trap that switches to kernel mode and jumps to a handler routine, which finally dispatches the desired system call.

(d) Binary semaphores provide the same synchronization power as locks. Why bother providing a separate lock interface
The semantics of locks are different from semaphores, e.g. a lock should only be released by the same process that previously acquired it, while with semaphores one process may execute the P() and another the V() operation. Having a different interface allows us to check for usage errors and makes it easier to understand the code, which is valuable because synchronization is hard.

(e) Can a user-level thread scheduler be pre-emptive? If so, describe how. If not, discuss why not.
Either answer is acceptable, with suitable explanation.
Yes. The user-level thread library must arrange for a periodic signal from the OS (say, once per second) and the signal handler must call the user-level thread scheduler to force a context switch.
No. There is no periodic clock interrupt visible to a process at user-level. Requesting a signal from the OS is probably too infrequent and won't work if the user-level threads themselves need to use the same signal to do their work.

../continued

2. [6 marks] Spinlocks

In the lecture notes, we showed a spinlock that used the atomic `test-and-set` instruction. We also claimed that a spinlock could be implemented using the atomic `swap` instruction, which swaps the contents of two 1-byte memory locations.

Implement a spinlock using `swap` by writing the `lock_acquire` and `lock_release` functions. The definition of a lock structure, and the prototype of the swap function is given below. The lock is available if `status` is 0 and held if `status` is 1.

```
struct lock {
     char status;
}

void swap(char *a, char *b);

lock_acquire(struct lock *l) {
     /* YOUR SOLUTION HERE */

     char status = 1;
     while (status)
          swap(&status, &l->status);

}

lock_release(struct lock *l) {
     /* YOUR SOLUTION HERE */

     l->status = 0;

     /* Also ok, but unnecessary: */
      * char status = 0;
      * swap(&status, &l->status);
      */

}
```

../continued

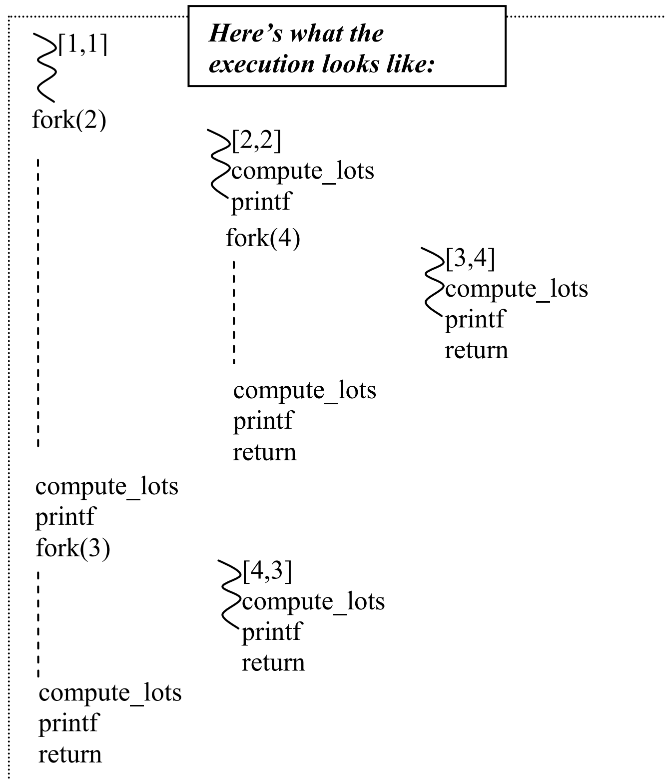
3. [10 marks] Scheduling

Assume a modified `fork()` system call, `fork(pri)`, that allows a child process to be created with a new scheduling priority different from that of the parent. Assume also a `getpri()` system call that returns the base priority of the calling process.

Assume that the first process has pid 1, that pids increase by one for each new process created, and that higher numbers represent higher priorities. Show what happens (order of process execution and output) when the program on the right is run with:

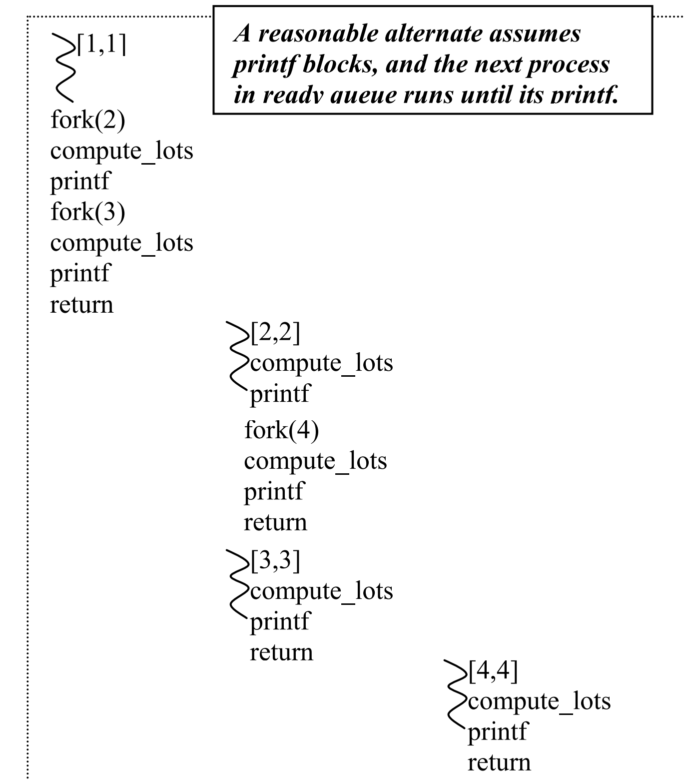
```
int main() {
    int mypid = getpid();
    int mypri = getpri();
    if (fork(mypri+1) == 0) {
        mypid = getpid();
        mypri = getpri();
    }
    compute_lots();
    print("[%d,%d] ", mypid, mypri);
    if (fork(mypri+2) == 0) {
        mypid = getpid();
        mypri = getpri();
    }
    compute_lots();
    print("[%d,%d] ", mypid, mypri);
    return 0;
}
```

(a)[4 pts] Pre-emptive priority scheduling



Output: [2,2] [3,4] [2,2] [1,1] [4,3] [1,1]

(b)[4 pts] FCFS scheduling



Output: [1,1] [1,1] [2,2] [2,2] [3,3] [4,4]

(c)[2pts] Assume that compute_lots() crunches numbers for a *really* long time. Would the output produced in (a) change if the scheduler dynamically adjusted priorities to prevent starvation? Why, or why not? *No. All processes do the same work in compute_lots and would have their priorities changed at the same rate. When only p1 and p2 exist, p2 has higher priority, but it decays during compute_lots until p1 can run. But p1 then calls compute_lots and **its** priority decays until p2 can run again! Even with adjustments, the relative priorities over time should stay the same. (but it depends a lot on exactly how priorities are adjusted and what happens when 2 processes have the same priority)*

../continued

4. [10 marks] Memory Management

Describe how memory fragmentation occurs in:

(a) Fixed partitioning systems:

Internal fragmentation happens when a process is allocated to a partition that is larger than it needs.

(b) Dynamic partitioning systems:

External fragmentation occurs when free memory is broken into many small chunks, which are too small to use, as partitions are created dynamically at the size a process needs and freed in the same fashion.

For each scenario, state whether *compaction* can be used to reduce fragmentation, and why or why not.

(a) dynamic partitioning with run-time address binding

Yes. Compaction can reduce fragmentation by moving allocated partitions together and coalescing the resulting free space. This requires that we be able to ***relocate*** *partitions, which requires dynamic address translation, which is possible when we have run-time address binding.*

(b) dynamic partitioning with load-time address binding

No. If addresses are bound to memory locations at load time, we cannot later relocate the partition, which is required for compaction. Once the program begins execution we can no longer identify all the addresses that would need to be translated after relocation.

(c) C library heap management routines (e.g. malloc/free)

No. There is no support for run-time address translation at user-level within a C program. Once malloc returns a pointer to a chunk of memory in the heap, we can't move that chunk to another part of the heap since we have no way of updating the pointer that was returned to the caller.

../continued

5. [10 Marks] Airline Reservations

Reserving a flight is an *atomic transaction* that typically involves reserving seats on each segment, or *leg*, of the trip. For example, a trip from Toronto to Lisbon has 4 legs: (Toronto -> NYC), (NYC -> Lisbon), (Lisbon -> NYC), (NYC -> Toronto).

Your job is to write a function `make_reservation()` which takes as input a linked list of legs and reserves a seat on each leg (by decrementing the available seats on that leg), if possible. The input list is in the order that legs will be taken during the trip. The function returns either `RESERVED` on success or `ENOSEATS` if it fails. Many threads (i.e., travel agents) may call `make_reservation` concurrently for different trips.

Your solution should use *two-phase locking* together with a *deadlock prevention* strategy. Do not worry about failures, other than failure to reserve a seat. Pseudo-code is acceptable. The linked list of flight segments is defined by the leg_s structure below:

```
struct leg_s {
    int flt_no;             /* Unique flight number of this leg */
    int seats_left;         /* Seats remaining on this leg */
    struct lock *flt_lock;  /* Guard access to seats_left */
    struct leg_s *next;     /* Next leg in trip */
};
```

(a)[2 pts] What property of concurrent transactions is guaranteed by two-phase locking?

Conflict serializability – conflicting transactions appear to complete in some serial order.

(b)[2 pts] Describe (in English) which deadlock prevention strategy you prefer for this problem, the deadlock condition that it prevents, and how it would work in this case.

Prevent circular wait by imposing an order on lock acquisition. In this case, we can sort the list of legs by flight number and require locks to be acquired from lowest flight number to highest flight number.

(c)[6 pts] On the following page, implement `make_reservation()`. You may assume helper function(s) that implement the deadlock prevention strategy you described in (b).

../continued

```
int make_reservation(struct leg_s *legs) {
    struct leg_s *tmp;

    sort_legs_by_flight_number(&legs); /* helper to get legs sorted */
    tmp = legs;
    while(tmp != NULL) {
        lock_acquire(tmp->flt_lock);
        if (tmp->seats_left > 0) {
            tmp->seats_left--;
            tmp = tmp->next;
        } else {
            /* failed to get seat, roll back earlier legs */
            struct leg_s *undo = legs;
            while (undo != tmp) {
                undo->seats_left++;
                lock_release(undo->flt_lock);
                undo = undo->next;
            }
            lock_release(tmp->flt_lock);
            return ENOSEATS;
        }
    }
    /* Success. Unlock everything */
    tmp = legs;
    while (tmp != NULL) {
        lock_release(tmp->flt_lock);
        tmp = tmp->next;
    }
    return RESERVED;

}
```

../continued

Extra space. Please indicate clearly which question(s) you are answering here, if any.

Total marks = (46)

End of test