University of Waterloo
Midterm Examination Sample Solution

Spring, 2014

1. (6 total marks)

Consider the application code below, which uses the `fork` and `waitpid` system calls.

```
int i;

dofork() {
  pid_t pid; int status;
  i++;
  pid = fork();
  if (pid == 0) { /* child */
    printf("%d",i);
    return;
  } else { /* parent */
    i--;
    waitpid(pid,&status,0);
    printf("%d",i);
    return;
  }
}

int main() {
  i = 0;
  dofork();
  dofork();
}
```

a. (2 marks) How many processes, in total, will be created when this program is run, including the original parent process?

Four processes.

b. (2 marks) Show the output that will be produced when this program runs. Note that the statement

```
printf("%d",i);
```

will print the value of integer variable `i`. You should show the combined output of all processes. Your answer should be a single sequence of numbers.

1 2 1 0 1 0

c. (2 marks) Suppose that the call to `waitpid` is removed from the `dofork` function, and the modified program is then run. Give an example of output that could be produced by this modified program, but that could not be produced by the original, unmodified program. Again, your answer should be a single sequence of numbers.

The sequence must include at exactly two 0's, three 1's, one 2, and nothing else. In addition

- at least one 1 must precede the 2, and
- at least one 0 must precede the last 1

This is one possible correct answer: 1 1 0 0 2 1

2. (10 total marks)

In this question, you are asked to consider a type of `lock` with a richer interface than that used by locks in OS/161. In addition to `lock_acquire` and `lock_release`, there is an additional interface function called `lock_try_acquire`, which is described in part (a) of this question.

You should assume that the `lock` structure is defined as follows:

```
struct lock {
  char *lk_name;
  struct wchan *wchan;
  struct spinlock spinlock;
  volatile struct thread *holder;
};
```

a. (5 marks) In the space below, implement the `lock_try_acquire` function, which has the following prototype:

```
int lock_try_acquire(struct lock *lock);
```

If the lock is free, `lock_try_acquire` should acquire the lock on behalf of the calling thread, and should return 1 to indicate success. If the lock is not free, `lock_try_acquire` should do nothing, and should return 0 to indicate failure. Thus, this function is similar to `lock_acquire` except that it must never cause the calling thread to block. Like `lock_acquire` and `lock_release`, `lock_try_acquire` must be an atomic operation.

```
int lock_try_acquire(struct lock *lock) {
  spinlock_acquire(&lock->spinlock);
  if (lock->holder == NULL) {
     lock->holder = curthread;
     spinlock_release(&lock->spinlock);
     return 1;
  } else {
     spinlock_release(&lock->spinlock);
     return 0;
  }
```

b. (5 marks) Suppose that a set of threads is using multiple locks for synchronization. In most cases, threads will acquire and release one lock at a time, using `lock_acquire` and `lock_release`. However, in some situations it is necessary for a thread to acquire two locks simultaneously. Write a function called `acquire_two_locks` that a thread can call to lock two different locks. The function prototype should be as follows:

```
void acquire_two_locks(struct lock *L1, struct lock *L2)
```

Before calling this function, the calling thread must not hold either lock. When this call returns, the calling thread should hold both of the specified locks.

Your implementation of `acquire_two_locks` must satisfy the following requirements:

1. The only synchronization primitives that your implementation may use are the two locks that are passed as input parameters. Your implementation may call `lock_acquire`, `lock_release`, and/or `lock_try_acquire` on those locks. (You may use `lock_try_acquire` here even if you did not implement it for part (a).) It may not use any other synchronization primitives, and it may not use wait channels or test-and-set instructions directly.
2. Your implementation must never "hold and wait". That is, it must never block or spin while it is holding one of the locks.

Keep your implementation as simple as possible. Overly long or complex solutions may be penalized.

```
void acquire_two_locks(struct lock *L1, struct lock *L2) {
  lock_acquire(L1);
  while ( lock_try_acquire(L2) == 0 ) {
     lock_release(L1);
     lock_acquire(L1);
  }
  return;
}
```

3 (8 total marks)

In the cat and mouse simulation from Assignment 1, a single master thread is responsible for creating one cat thread for each simulated cat, and one mouse thread for each simulated mouse. After creating the cat and mouse threads, the master thread waits for all of the cats and all of the mice to finish their simulations. To do this, it uses a single semaphore, called `CatMouseWait`, which has an initial value of 0. The master thread uses the following code to wait for all of the cats and all of the mice to finish

```
for(i=0;i<(NumCats+NumMice);i++) {
  P(CatMouseWait);
}
```

Here, `NumCats` and `NumMice` are global variables representing the numbers of cats and mice. Each cat and each mouse, when it finishes its simulation, executes

```
V(CatMouseWait);
```

to indicate that it has finished.

Your task is to re-implement this mechanism, using locks and/or condition variables instead of semaphores. You may use as many locks, condition variables, and global variables as you need in your solution. However, you may not use semaphores, spinlocks, or any other synchronization primitives.

a. (2 marks) In the space below, declare any lock(s), condition variable(s), and shared global variable(s) you will need in your solution. Be sure to indicate an initial value for any shared global variable(s).

```
struct lock *mutex;
struct cv *cv;
volatile int num_animals = NumCats + NumMice;
```

b. (3 marks) In the space below, show the code that should be used by the master thread to wait for all cats and mice to finish. This should use the variable(s), lock(s) or condition variable(s) you declared in part (a).

```
lock_acquire(mutex);
while (num_animals > 0) {
   cv_wait(cv,mutex);
}
lock_release(mutex);
```

c. (3 marks) In the space below, show the code that should be used by each cat and mouse thread to indicate that it has finished. This should use the variable(s), lock(s) or condition variable(s) you declared in part (a). Cats and mice must use the same code.

```
lock_acquire(mutex);
num_animals--;
if (num_animals == 0) {
   cv_signal(cv,mutex);
}
lock_release(mutex);
```

4 (6 total marks) The following assembly language pseudo-code shows how the load linked (`ll`) and store conditional (`sc`) instructions can be used together to test-and-set a lock. In this code, `&lock` represents the address of the lock variable. The comments remind you how the `ll` and `sc` instructions behave.

```
  // load the value 1 into register R1
li R1,1
  // load the value of the lock variable into register R0 */
ll R0,&lock
  // if the value of the lock variable has not changed since the ll
  // instruction, store the value in R1 into the lock variable and
  // set the value in R1 to 1 to indicate success.   Otherwise,
  // do not change the value of the lock variable and set the value
  // of R1 to 0 to indicate failure.
sc R1,&lock
```

Suppose that a thread T executes these instructions as part of a call to `spinlock_acquire`. Immediately after T executes the `sc` instructions, there are four possible situations, depending on the values in the registers `R0` and `R1`.

The table below lists these four possible situations. For each situation, indicate which of the following statements is true:

- T holds the lock.
- Some thread other than T holds the lock.
- No thread holds the lock.
- Not possible to determine whether the lock is held.

Indicate your answers by writing the correct statement in each box. The same statement may appear in more than one box.

Value of R0	Value of R1	Statement
0	0	Not possible to determine whether the lock is held.
0	1	T holds the lock.
1	0	Not possible to determine whether the lock is held.
1	1	Some thread other than T holds the lock.

5 (6 total marks) Suppose that two different types of processes, crunchers and talkers, run in a system. The system has a single processor, and it uses preemptive round-robin scheduling, with a scheduling quantum of q time units.

Crunchers never block. When they are chosen to run by the scheduler, they will run until they are preempted. Talkers, on the other hand, continuously output characters, as illustrated by the following pseudo-code:

```
while (true) {
  /* output a character */
}
```

Each time a talker outputs a character, it blocks for b time units while the character is being output, before becoming ready again. Assume that the actual execution time (time spent in the "running" state) for each iteration of the talker is very small - much smaller than b.

Answer each of the following questions about this system. Express your answers in terms of b and q. Assume that $b < q$.

a. (2 marks) Suppose that there is one talker process in the system, and no other processes. How long will it take the talker to output 100 characters?

$100b$

b. (2 marks) Suppose that there is one talker and one cruncher running in the system. How much time will elapse before the talker outputs 100 characters?

$100q$

c. (2 marks) Suppose that one talker and k ($k > 0$) crunchers are running in the system. How much time will elapse before the talker outputs 100 characters? Express your answer in terms of k, b and q.

$100kq$

6 (8 total marks) Suppose threads in a concurrent program share access to two different FIFO queues of data, QueueA and QueueB. There are two functions that threads use to move data items between the two queues:

- `AtoB()`: this function dequeues one item from QueueA and enqueues that item onto QueueB
- `BtoA()`: this function dequeues one item from QueueB and enqueues that item onto QueueA

Suppose that, initially, QueueA contains N data items, and QueueB is empty.

On the next page, you will find skeleton implementations of `AtoB()` and `AtoB()`. Your job is to modify these functions by inserting semaphore operations to ensure that certain synchronization requirements are satisfied. These requirements are as follows:

1. `dequeue()` should never be run on an empty queue.
2. At most one thread at a time should be using each queue. For example, if one thread is running `dequeue()` on QueueA, no other thread should be running `dequeue()` or `enqueue()` on QueueA.
3. Items must be enqueued onto QueueB in the same order that they are dequeued from QueueA. Similarly, items must be enqueued onto QueueA in the same order that they are dequeued from QueueB.
4. Threads must never deadlock.

In addition, it must be possible, at least in some situations, for different threads to use different queues at the same time. In particular, a solution that uses a single semaphore to lock both queues is not acceptable.

Add semaphore operations (P and V) to the skeleton code on the next page so that these synchronization requirements will be satisfied. Do not use any synchronization primitives other than semaphores in your solution. Do not make any changes to the skeleton code other than inserting calls to semaphore operations.

Declare Semaphores Here:

```
Semaphores:
   Acount (initial value N)
   Bcount (initial value 0)
   Amutex (initial value 1)
   Bmutex (initial value 1)
   AtoBmutex (initial value 1)
   BtoAmutex (initial value 1)
```

```
void AtoB() {
    P(Acount);
    P(AtoBmutex);
    P(Amutex);
  /* dequeue item from QueueA */
  x = dequeue(QueueA);
    V(Amutex);
    P(Bmutex);
  /* enqueue the dequeued
     item onto QueueB */
  enqueue(x,QueueB);
    V(Bmutex);
    V(AtoBmutex);
    V(Bcount);
}
```

```
void BtoA() {
    P(Bcount);
    P(BtoAmutex);
    P(Bmutex);
  /* dequeue item from QueueB */
  x = dequeue(QueueB);
    V(Bmutex);
    P(Amutex);
  /* enqueue the dequeued
     item onto QueueA */
  enqueue(x,QueueA);
    V(Amutex);
    V(BtoAmutex);
    V(Acount);
}
```

7 (16 total marks)

a. (3 total marks) Which of the following effects does a MIPS `syscall` instruction have when it is executed? (Note: we are interested only in the effects of this single instruction, not the effects of any code that runs after this instruction.) Circle all that apply.

- [x] the current value of the program counter is saved
- [ ] a trap frame is saved
- [x] the processor switches to privileged execution mode
- [ ] the current thread stops running
- [ ] a timer interrupt occurs
- [ ] an error code is returned to the application
- [x] the value of the program counter is changed

b. (2 total marks) Neither threads that are in the ready state nor threads that are in the blocked state are running. What is the difference between these two states?

A ready thread is runnable, and will run again as soon as the scheduler chooses it to run. A blocked thread will not become runnable again until a another process wakes it up (via a call to `wchan_wakeone` or `wchan_wakeall`).

c. (2 total marks) Explain why disabling interrupts may not enforce mutual exclusion on a multiprocessor machine.

Disabling interrupts on one processor will not prevent a thread running on another processor from entering the critical section.

d. (1 total marks) If a thread avoids "holding and waiting", then that thread will never be involved in a deadlock. True or false?

False. (The thread may block waiting for something held by another thread that is deadlocked.)

e. (1 total marks) If all threads avoid "holding and waiting", then no thread will ever be involved in a deadlock. True or false?

True.

f. (3 total marks) We have identified three types of events that cause execution control to transfer from an application program to the kernel. What are those three types of events?

System calls, interrupts, and exceptions.

g. (2 total marks) Suppose that a process P calls `waitpid` and blocks because the process it is waiting for is still running. At the time that P blocks, how many trap frames will be on P's thread's stacks, and which stack, or stacks, will those trap frames be found on? Briefly justify your answer.

One trap frame, which will be on P's thread's kernel stack. The trap frame will have been created when P's thread switched to kernel mode to handle the `waitpid` system call.

h. (2 total marks) For the same situation described in part (g), how many switch frames will be on P's thread's stacks, and which stack, or stacks, will those switch frames be found on? Briefly justify your answer.

One switch frame, which will be on P's thread's kernel stack. The switch frame will have been created when P's thread blocked, and the thread library switched execution to a different thread.