University of Waterloo
Midterm Examination #1 Model Solution
Spring, 2009

1. (12 total marks)

Show how to implement condition variables using semaphores. For the purposes of this question, you may **not** use `thread_sleep`, `thread_wakeup`, or any of the other thread library functions. Follow the detailed instructions for each part of this question.

a. (2 marks) Show what fields your `cv` structure will have by completing the type definition below. The only types of fields you may include in this structure are semaphore pointers (`struct semaphore *`) or integers or both. For the purposes of this question, ignore the `cv name` field which is found in the OS/161 `cv` structure.

Keep your solution simple. Unnecessarily complex solutions may be penalized.

```
struct cv {
  struct semaphore *waitsem;
  volatile int waitcount;
};
```

b.(3 marks) Show how the fields in your semaphore structure are initialized by completing the implementation of `cv_create`, which is shown below. Recall that the signature of the semaphore creation function is:

```
struct semaphore *sem_create(const char *name, int initial_count);
```

Include error-handling code, as necessary, in your answer.

```
struct cv *
cv_create() {
  struct cv *cv;
  cv = (struct cv *)kmalloc(sizeof(struct cv));
  if (cv == NULL) { return NULL; }
  /* initialized the fields of the cv structure here */
  cv->waitcount = 0;
  cv->waitsem = sem_create(``waitsem'',0);
  if ( cv->waitsem == NULL ) {
     kfree(cv);
     return(NULL);
  }
  return(cv);
}
```

c. (4 marks) Implement `cv_wait` using your `cv` structure. Make sure that you implement the correct behaviour for `cv_wait`, as discussed in class. You may use `lock_acquire` and/or `lock_release` on the parameter `lock` if needed. Remember: do not use `thread_sleep` or or any other thread library functions. In addition, your `cv_wait` should not use busy waiting - you must use semaphores to provide any blocking that is needed.

```
void
cv_wait(struct cv *cv, struct lock *lock)
{
  int spl;
  spl = splhigh();
  cv->waitcount++;
  lock_release(lock);
  P(cv->waitsem);
  lock_acquire(lock);
  splx(spl);
}
```

d. (3 marks) Implement `cv_signal` using your `cv` structure. Make sure that you implement the correct behaviour for `cv_signal`, as discussed in class. You may use `lock_acquire` and/or `lock_release` on the parameter `lock` if needed. Remember: do not use `thread_wakeup` or or any other thread library functions.

```
void
cv_signal(struct cv *cv, struct lock *lock)
{
  int spl;
  spl = splhigh();
  if ( cv->waitcount > 0 ) {
    cv->waitcount--;
    V(cv->waitsem);
  }
  splx(spl);
}
```

2. (10 marks)

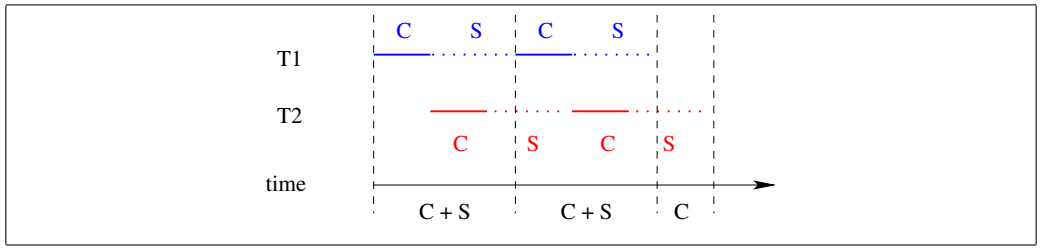
Suppose that there are K threads in a system that uses preemptive round-robin scheduling with a scheduling quantum of Q milliseconds. The system has a single processor. Each thread runs a function which behaves as follows

```
for i from 1 to N do
  compute for C milliseconds
  sleep for S milliseconds
end
```

At the end of its `for` loop, a thread is finished and it exits. During the "compute" part of each of its iterations, a thread is runnable (running or ready to run). During the "sleep" part of each of its iterations, a thread is blocked. For both parts of this question assume that $C < Q$ and $C < S$.

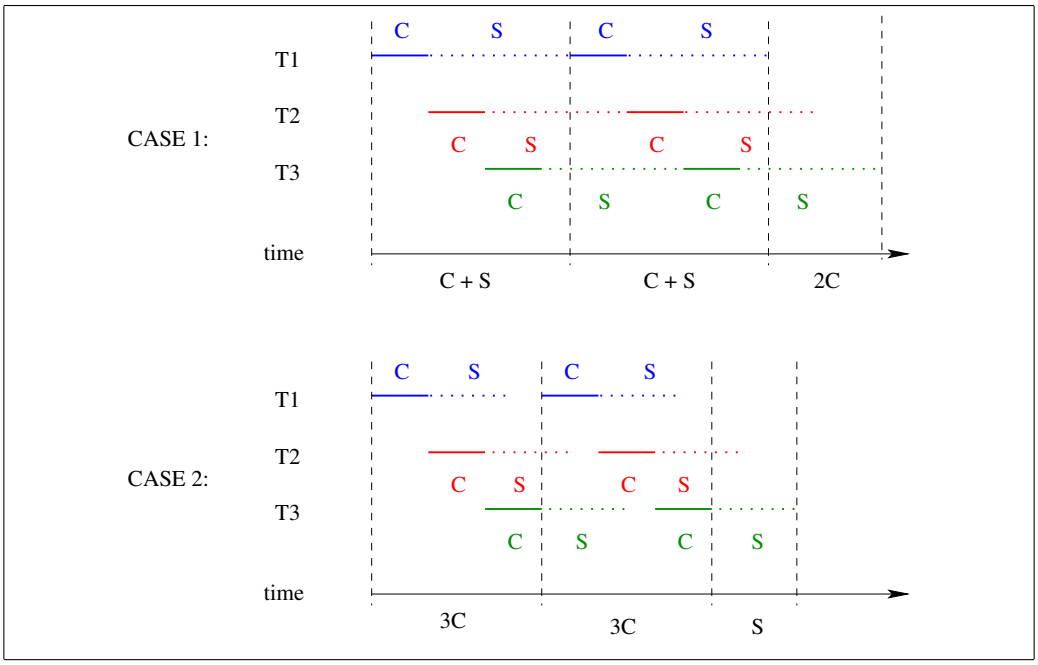
a. (4 marks) For this part of the question, suppose that $K = 2$. Suppose that both of the threads are created at time $t = 0$. At what time will both of the threads be finished? Express your answer in terms of Q, N, C, and S, as necessary.

$N(C + S) + C$



b. (6 marks) For this part of the question suppose that $K = 3$. Assuming that all of the threads are created at $t = 0$, at what time will all of the threads be finished? Express your answer in terms of Q, N, C, and S, as necessary.

CASE 1 ($2C \leq S$): $N(C + S) + 2C$
CASE 2 ($2C > S$): $3NC + S$



3. (12 total marks)

Consider a small computer system with 16-bit virtual addresses and 16-bit physical addresses. The page size in this system is 256 (2^8) bytes. Suppose that there are two processes, P_1 and P_2, running in this system, with page tables as shown below. For the purposes of this question, assume that all of the page table entries shown in the tables are valid.

P_1's page table

Page Number	Frame Number
0	0x10
1	0x14
2	0x4a
3	0x05
4	0x22
5	0x21
6	0x1a
7	0x58

P_2's page table

Page Number	Frame Number
0	0x3b
1	0x01
2	0x0b
3	0x38
4	0x30
5	0x2c

a. (3 marks)
For each of the following virtual addresses from P_1's virtual address space, indicate the physical address to which it corresponds. Give your answers in hexadecimal. If the specified virtual address is not part of the virtual address space of P_1, write "NO TRANSLATION" instead.

- `0x034a` → *0x054a*
- `0x1004` → *NO TRANSLATION*
- `0x0022` → *0x1022*

b. (3 marks)
For each of the following *physical* addresses, indicate which process's virtual address space maps to

that physical address, and indicate which specific virtual address maps there. If neither process has a virtual address space that maps to the given physical address, write "NO MAPPING" instead.

- `0x2222` → *0x0422 Process* P_1
- `0x38ff` → *0x03ff Process* P_2
- `0x0123` → *0x0123 Process* P_2

c. (2 marks)

What is the size (in bytes) of P_1's virtual address space?

$$8(2^8) = 2^{11} = 2048$$

d. (2 marks)

What is the maximum number of page table entries that might be needed for a page table in this system?

$$\frac{2^{16}}{2^8} = 2^8 = 256$$

e. (2 marks)

Suppose that the kernel is creating a new process P_3 to run in the system, in addition to P_1 and P_2. The new process should have a virtual address space of size 1024 (2^{10}) bytes. Assume that the system has the maximum amount (2^{16} bytes) of physical memory. Show a legitimate page table for P_3, similar to the ones shown previously, that might be created by the kernel. Many such page tables are possible - you only need to show one of them.

The table should have four entries. Any frame numbers that do not appear in the tables for P_1 or P_2 can be used to fill in the entries. For example:

Page Number	Frame Number
0	0x00
1	0x02
2	0x03
3	0x04

4. (6 marks)

a. (2 marks)

What is one *advantage* of Peterson's mutual exclusion algorithm over mutual exclusion provided by a spin lock using a test-and-set instruction?

Peterson's algorithm does not require an atomic synchronization instruction, like test-and-set. Atomic loads and stores are sufficient. Also, starvation is not possible under Peterson's algorithm.

b. (2 marks)

Briefly explain the difference between `thread_yield` and `thread_sleep`.

After yielding, a running thread becomes *ready* and is eligible to run again as soon as it is selected by the scheduler. After sleeping, a running thread is *blocked* and is not eligible to run again until it has been awakened by a subsequent call to `thread_wakeup`.

c. (2 marks)

In OS/161, how do applications pass system call parameters to the kernel?

System call parameters are passed by placing them into the usual parameter passing registers (A0, A1, A2, ...) prior to the `syscall` instruction.