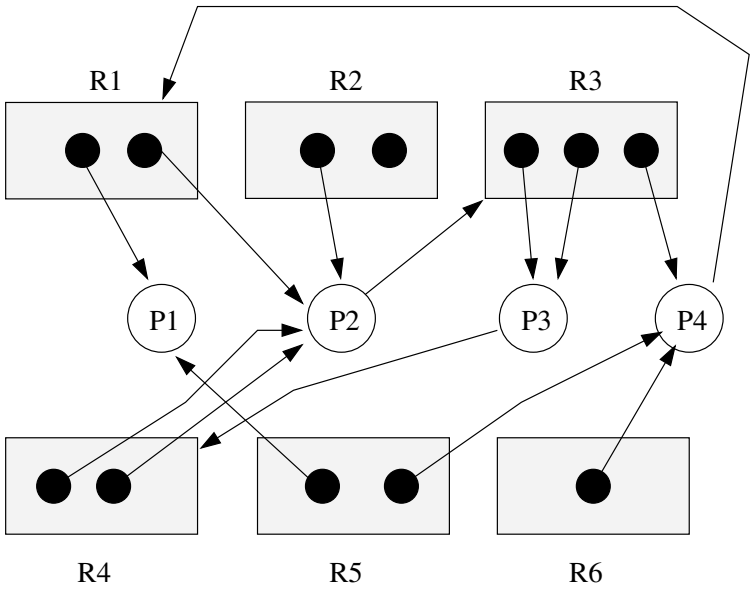
University of Waterloo
Midterm Examination #1 Model Solution
Spring, 2007

1. (6 marks)

Consider the following resource allocation graph, which is like the ones shown in the course notes and discussed in class. Is there a deadlock in this system? Answer "yes" or "no". If your answer is yes, indicate which processes are involved in the deadlock. If your answer is "no", indicate at least one sequence in which the processes could run to completion, e.g., "P1, then P2, then P3, then P4".



There is no deadlock. Processes can run to completion in the order P1,P4,P2,P3.

2. (10 marks)

Suppose that there are two processes, P_H and P_L, running in a system. Each process is single-threaded. The operating system's scheduler is preemptive and uses round-robin scheduling with a quantum of q time units. The scheduler supports two priority levels, `HIGH` and `LOW`. Processes at `LOW` priority will run only if there are no runnable `HIGH` priority processes.

Process P_H is a `HIGH` priority process. It behaves as described in the following pseudo-code:

```
while (TRUE) do
  compute for t_c time units
  block for t_b time units to wait for a resource
end while
```

That is, if this process were the only one running in the system, it would alternate between running for t_c units of time and blocking for t_b units of time. Assume that $t_c < q$.

Process P_L is a low priority process. This process runs forever, doing nothing but computation. That is, it never blocks waiting for a resource.

a. (3 marks)

For what percentage of the time will the low priority process P_L be running in this system? Express your answer in terms of t_b and t_c.

$$\frac{t_b}{t_b + t_c}$$

b. (7 marks)

Repeat part (a), but this time under the assumption that there are *two* `HIGH` priority processes (P_{H1} and P_{H2}) and one `LOW` priority process (P_L). Assume that each `HIGH` priority process waits for a different resource. Again, express your answer in terms of t_b and t_c. Your answer should be correct for all $t_b > 0$ and all $0 < t_c < q$.

$$\begin{cases} \frac{t_b - t_c}{t_b + t_c} & \text{if } t_c < t_b \\ 0 & \text{if } t_c \geq t_b \end{cases}$$

3. (14 marks)

Consider a version of the producer/consumer problem in which there are two buffers, `BufferA` and `BufferB`, rather than just one. `BufferA` has a capacity of N_A items. `BufferB` has a capacity of N_B items. Both buffers are initially empty.

Threads can use these buffers as follows:

- By calling the function `ProduceA`, a thread can add one item to `BufferA`. If `BufferA` is full, `ProduceA` should block until there is space available.
- By calling the function `ProduceB`, a thread can add one item to `BufferB`. If `BufferB` is full, `ProduceB` should block until there is space available.
- By calling `Consume`, a thread can remove one item from one of the buffers. `Consume` will always remove an item from `BufferA` if `BufferA` is not empty, otherwise it will remove an item from `BufferB`. If both `BufferA` and `BufferB` are empty, `Consume` should block until an item becomes available in either buffer, and then it should return that item.

In addition to the synchronization requirements described above, there are the following additional mutual exclusion requirements for these buffers:

- If any thread is running `ProduceA`, no other thread should be running `ProduceA` or `Consume`.
- If any thread is running `ProduceB`, no other thread should be running `ProduceB` or `Consume`.
- If any thread is running `Consume`, no other thread should be running `ProduceA`, `ProduceB`, or `Consume`.

Note that it is OK for `ProduceA` and `ProduceB` to run concurrently.

On the next page, you will find pseudo-code implementations of the `ProduceA`, `ProduceB`, and `Consume` functions. Your job is to use semaphores to enforce all of the synchronization requirements described above without causing threads to block unnecessarily.

Show how to enforce these requirements by **inserting semaphore `P` and `V` calls into the pseudo-code on the next page**. Do not make any changes to the pseudo-code other than inserting `P` and `V` calls in the correct places and in the correct order.

In addition, **for each semaphore that you use in your solution, you must provide a declaration of that semaphore**. Your declaration must explicitly state what type of semaphore (binary or counting) is being declared, as well as the initial value of that semaphore. For each semaphore that you declare, you should provide a brief (i.e., at most one sentence) comment to indicate the semaphore's intended purpose. For example, a declaration of a counting semaphore with initial value 5 might look something like this:

```
countingsemaphore foo(5);  /* tracks the number of widgets */
```

Place Semaphore Declarations Here

binarysemaphore mutexA(1)
binarysemaphore mutexB(1)
countingsemaphore itemsAB(0)
countingsemaphore freespaceA(N)
countingsemaphore freespaceB(N)

```
ProduceA(item I) {

  P(freespaceA)
  P(mutexA)

  insert I into BufferA

  V(mutexA)
  V(itemsAB)
}
```

```
ProduceB(item I) {

  P(freespaceB)
  P(mutexB)

    insert I into BufferB

  V(mutexB)
  V(itemsAB)
}
```

```
Consume() {

  P(itemsAB)
  P(mutexA)
  P(mutexB)

  if BufferA is not empty then

    remove item from Buffer A

    V(freespaceA)

  else

    remove item from Buffer B

    V(freespaceB)

  endif

  V(mutexB)
  V(mutexA)

  return item
}
```

4. (10 marks)

a. (2 marks)

Explain the distinction between exceptions and interrupts.

Interrupts are causes by devices (e.g., timer, disk), while exceptions are caused by an executing program.

b. (3 marks)

The address space of a NachOS process includes code, read-only data, initialized data, uninitialized data, and stack segments. From where does NachOS obtain the information that it it needs to determine the size of each of these segments?

The kernel obtains the sizes of the code, read-only data, initialized data, and uninitialized data sections from the executable file (NOFF file) that is used to initialize the process. The size of the stack segment is chosen by the kernel itself.

c. (3 marks)

What event(s) cause a processor to switch from unprivileged execution mode to privileged execution mode? What event(s) cause a processor to switch back from privileged execution mode to unprivileged execution mode?

Interrupts, exceptions and system calls cause the processor to switch from unprivileged execution mode to privileged execution mode. Returning from an interrupt, exception or system call causes execution mode to switch from privileged mode back to unprivileged mode.

d. (2 marks)

What is the distinction between a *ready* thread and a *blocked* thread?

A ready thread is able to run - it is waiting to be chosen by the scheduler. A blocked thread not able to run - it is waiting for an event to occur or a resource to become available.