COMP 530 Introduction to Operating Systems

Fall 2017  
Kevin Jeffay

Worksheet 7, September 25

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Your Name: |  | You worked with: |  | +1/blank/-1: |  |
|  | Aaron Zhang |  | Brennan Proudfoot |  | +1 |  |
|  |  |  | John Espenhahn |  | +1 |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. Consider the following four (4) pseudo-code implementations of general semaphores spread out over the next four pages. All use the following semaphore data type (although not all implementations use all the fields of the struct):

struct semaphore

*mutex* : binary\_semaphore

*delay* : binary\_semaphore

*numWaiting* : integer

*queue* : system\_queue

*value* : integer

end struct

Are any of the following implementations correct? For each implementation that you think is correct (if any) give the best argument that you can that the implementation is correct (keep your argument to no more than 1 page of text). For each implementation that you think is incorrect (if any), demonstrate a set of interleavings of processes that call *up* and *down* that leads to some error condition. Make sure your description of any problem you discover in a semaphore implementation is clear and complete! For example, for semaphore implementations that you suspect are incorrect, consider using a producer/consumer system that uses the semaphore implementation could lead to an incorrect execution of an otherwise producer/consumer system. For semaphore implementations that you believe are correct, compare the implementation to those in the lecture notes that have already been “certified” as correct.

In answering this question you should assume an underlying round-robin process scheduler and assume that all semaphores used in an implementation have the appropriate initial value.

|  |  |  |
| --- | --- | --- |
| *a*) | *down*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  while (*sem.value* = 0) then  *sem.numWaiting* += 1  DISABLE\_INTS  insert\_queue(*sem.queue*, *running*)  *next* := remove\_queue(*readyQueue*)  sem.mutex.up*b*()  dispatch(*next*)  ENABLE\_INTS  sem.mutex.down*b*()  end while  *sem.value* := *sem.value* - 1  sem.mutex.up*b*()  end *down* | *up*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  *sem.value* := *sem.value* + 1  if (*sem.numWaiting* > 0) then  *next* := remove\_queue(*sem.queue*)  *sem.numWaiting* -= 1  DISABLE\_INTS  insert\_queue(*readyQueue, next*)  ENABLE\_INTS  end if  sem.mutex.up*b*()  end *up* |

This is correct. You can reason by correctness by only looking at down. You just performed a test before you call down(). When down() is called, the caller is effectively starting over. Because you are testing before you call, you are guaranteed you are never decrementing below zero.

Basic paradigm of a context switch is you always 1. Disable interrupts 2. Do a dispatch 3. Enable interrupts

|  |  |  |
| --- | --- | --- |
| *b*) | down(var sem : semaphore)  begin  sem.mutex.downb()  if (sem.value = 0) then  sem.numWaiting += 1  sem.mutex.upb()  sem.delay.downb()  else  sem.value := sem.value - 1  sem.mutex.upb()  end if  end down | up(var sem : semaphore)  begin  sem.mutex.downb()  if (sem.numWaiting > 0) then  sem.numWaiting -= 1  sem.delay.upb()  else  sem.value := sem.value + 1  end if  sem.mutex.upb()  end up |

This does not work.

Binary semaphore is stateless.

If the binary semaphore is already one, you are going to lose the effect of calling up(). Consecutive up operations will have no effect, need to add state by adding queues.

|  |  |  |
| --- | --- | --- |
| *c*) | *down*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  if (*sem.value* = 0) then  *sem.numWaiting* += 1  DISABLE\_INTS  insert\_queue(*sem.queue*, *running*)  *next* := remove\_queue(*readyQueue*)  sem.mutex.up*b*()  dispatch(*next*)  ENABLE\_INTS  else  *sem.value* := *sem.value* - 1  sem.mutex.up*b*()  end if  end *down* | *up*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  if (*sem.numWaiting* > 0) then  *sem.numWaiting* -= 1  *next* := remove\_queue(*sem.queue*)  DISABLE\_INTS  insert\_queue(*readyQueue, next*)  ENABLE\_INTS  else  *sem.value* := *sem.value* + 1  end if  sem.mutex.up*b*()  end *up* |

This is correct

Baton passing from the lecture notes. All you do in down is decrement and then leave, why do all the headache in down of doing all the baton passing.

Compare the execution of this code to the baton passing code, the steps are the same just done in a different process.

|  |  |  |
| --- | --- | --- |
| *d*) | *down*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  if (*sem.value* = 0) then  *sem.numWaiting* += 1  DISABLE\_INTS  insert\_queue(*sem.queue*, *running*)  *next* := remove\_queue(*readyQueue*)  sem.mutex.up*b*()  dispatch(*next*)  ENABLE\_INTS  *sem.mutex.downb()//remove*  end if  *sem.value* := *sem.value* - 1  sem.mutex.up*b*()  end *down* | *up*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  *sem.value* := *sem.value* + 1  if (*sem.numWaiting* > 0) then  *sem.numWaiting* -= 1  *next* := remove\_queue(*sem.queue*)  DISABLE\_INTS  insert\_queue(*readyQueue, running*)  sem.mutex.up*b*()//remove  dispatch(next)  ENABLE\_INTS  else  sem.mutex.up*b*()  end if  end *up* |

The issue is that we dispatch the waiting process rather than placing it inside the ready queue.

No, the solution suffers from a race condition. Once you wake up from dispatch, nothing can guarantee once you enable interrupts nothing can be effected.

This code can be fixed by baton passing