COMP 530 Introduction to Operating Systems

Fall 2017  
Kevin Jeffay

Worksheet 8, September 27

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

1. Consider the following producer/consumer system solution from the lecture that uses the “baton passing” general semaphore implementation that was also presented in lecture (see slides 9 and 27 from the lecture on semaphores):

struct semaphore

*mutex* : binary\_semaphore

*numWaiting* : integer

*queue* : system\_queue

*value* : integer

end struct

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

|  |  |  |
| --- | --- | --- |
|  | process *Producer*  begin  loop  *<produce a character “c”>*  *emptyBuffers.down()*  *buf[nextIn] := c*  *nextIn := nextIn+1* mod *n*  *fullBuffers.up()*  end loop  end *Producer* | process *Consumer*  begin  loop  *fullBuffers.down()*  *data := buf[nextOut]*  *nextOut := nextOut+1* mod *n*  *emptyBuffers.up()*  *<consume “data”>*  end loop  end *Consumer* |

Assume all the above code executes on an operating system using round-robin scheduling with a quantum that is very large compared to the time required to produce or consume a character.

Simulate the execution of this producer/consumer system and semaphore implementation and characterize the set of possible interleavings of the execution of the producer and consumer processes. That is, is it possible for the producer and consumer to interleave their execution in arbitrary ways, or do stylized patterns of process execution or interleaving result?

Producer produces N items in the long quantum. Consumer takes one item off the full buffer after calling down. Then calls up to empty Buffer which passes empty Buffer baton to the producer (because the producer was waiting in down on the empty buffers). Consumer busy waits on binary mutex get-set in down, because the producer has the binary mutex. This wastes most of the consumer’s quantum. After quantum expires, goes back to producer.

1. Recall the correct semaphore implementations from Worksheet 7.

|  |  |  |
| --- | --- | --- |
| *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* |

|  |  |  |
| --- | --- | --- |
| *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* |

Perform the simulation exercise you did in problem 1 except now use these semaphore implementations instead of the implementation from the lecture notes. That is, use the producer/consumer code in problem 1 with each of these two semaphore implementations (*i.e.*, perform two separate sets of simulations, one for each of the correct semaphore implementation above), and characterize the possible set of interleavings of the execution of the producer/consumer processes that result.

Do you get a different set of process interleavings depending on which semaphore implementation is used in your simulation? If so, is one set of possible interleavings more or less appealing to you than the other?

©

In up you only increment the value of the semaphore when there is not waiting process.

Better than baton passing producer can produce all buffers, the consumer can consume all buffers

In baton passing, once the buffer was full, consumer had to wait for the producer to do something to continue. If the quantum was small there is no coupling between producer and consumer.

Three correct semaphore implementations. Certain aesthetics etc are proffered.

Use this page for extra space (if needed) to answer question 2.

Use this page for extra space (if needed) to answer question 2.