COMP 530 Introduction to Operating Systems

Fall 2017  
Kevin Jeffay

Worksheet 14, October 23

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Your Name: |  | You worked with: |  | +1/blank/-1: |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. Someone once observed that in Hoare monitors, the *signal* operation is usually the last statement that is executed in a monitor function/method. For example, in the monitor lecture I gave you an example of a Hoare monitor with entries *deposit* and *remove* to solve the producer/consumer problem. The body of this monitor appears below. In this code you can see that in fact the *signal* statement is the last executable statement in each monitor function/method.

The person who made this observation went on to develop a variant of Hoare monitors wherein the only place a *signal* operation could be performed was as the last executable statement in a monitor function/method. This change only effected where signaling could be performed within a monitor. In particular, this change had no effect on the semantics of synchronization. (Note: We’re not saying that a *signal* must be the last statement in a monitor function/method; only that if a function/method in a monitor contains a *signal* statement, then the *signal* statement must be the last executable statement in the function/method.)

*a*) Given that this change does not affect the semantics of synchronization, what advantage, if any, is there to restricting the use of the *signal* primitive to only appear as the last executable statement in a method of a Hoare monitor?

*b*) The pseudo code for implementing a Hoare monitor that was presented in lecture is presented on this page and the next page. Either modify (edit) or rewrite this code in the space below to reflect any changes to the implementation that are made possible by the rule that signal operations can only performed as the last statement in a procedure exported by the monitor.

|  |  |
| --- | --- |
| struct conditionVar  queue : systemQueue  numWaiting : integer := 0  end struct | var  monitorCodeMutex : binarySem := 1  monitorBusy : Boolean := FALSE  entryQueue : systemQueue  urgentQueue : systemQueue  numWaiters : integer := 0  numSignalers : integer := 0 |

function wakeAWaiter() : boolean

begin

var waiter : process\_ID

if(numSignalers > 0) then

numSignalers -= 1

waiter := remove\_queue(urgentQueue)

elseif(numWaiters > 0) then

numWaiters -= 1

waiter := remove\_queue(entryQueue)

else

return(FALSE)

end if

DISABLE\_INTS

insert\_queue(readyQueue, waiter)

ENABLE\_INTS

return(TRUE)

end wakeAWaiter

procedure EnterMonitor()

begin

var next : process\_ID

monitorCodeMutex.down*b*()

if (monitorBusy) then

numWaiters += 1

insert\_queue(entryQueue, running)

DISABLE\_INTS

next := remove\_queue(readyQueue)

monitorCodeMutex.up*b*()

dispatch(next)

ENABLE\_INTS

else

monitorBusy := TRUE

monitorCodeMutex.up*b*()

end if

end EnterMonitor

procedure ExitMonitor()

begin

var waiterAwoken : boolean := FALSE

monitorCodeMutex.down*b*()

waiterAwoken := wakeAWaiter()

if (!waiterAwoken) then

monitorBusy := FALSE

end if

monitorCodeMutex.up*b*()

end ExitMonitor

procedure Wait(cv : conditionVar)

begin

var next : process\_ID

cv.numWaiting += 1

insert\_queue(cv.queue, running)

monitorCodeMutex.down*b*()

if (!wakeAWaiter()) then

monitorBusy := FALSE

end if

monitorCodeMutex.up*b*()

DISABLE\_INTS

next := remove\_queue(readyQueue)

dispatch(next)

ENABLE\_INTS

end Wait

procedure Signal(cv : conditionVar)

begin

var next := process\_ID

if (cv.numWaiting > 0) then

cv.numWaiting -= 1

numSignalers += 1

insert\_queue(urgentQueue, running)

next := remove\_queue(cv.queue)

DISABLE\_INTS

dispatch(next)

ENABLE\_INTS

end if

end Signal

2. Consider the following implementation of a general semaphore. Is the implementation correct? If so, give a brief sketch of a correctness argument (arguing correctness either from first principles or by an equivalence argument to an alternate implementation previously shown to be correct). If you believe the implementation is incorrect demonstrate a scenario in which an error occurs.

struct sem

*mutex* : binary\_semaphore /\* Assume initialized to 1 \*/

*balance*  : integer

*queue* : system\_queue

*value* : integer /\* Assume ‘value’ and ‘balance’ both \*/

end struct /\* initialized to ‘k’ for some k ≥ 0 \*/

|  |  |
| --- | --- |
| *down*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  *sem.balance* := *sem.balance* – 1  if (*sem.balance* < 0) then  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*()  endif  end *down* | *up*(var *sem* : semaphore)  begin  var *next* : processId  sem.mutex.down*b*()  *sem.balance* := *sem.balance* + 1  if (*sem.balance* ≤ 0) then  *next* := remove\_queue(*sem.queue*)  DISABLE\_INTS  insert\_queue(*readyQueue, next*)  ENABLE\_INTS  else  *sem.value* := *sem.value* + 1  endif  sem.mutex.up*b*()  end *up* |