

Project 2 – Solution to Producer/Consumer and Dining Philosophers

----- ProducerConsumer -----

```
-- This code implements the consumer-producer task.  There are several
-- "producers", several "consumers", and a single shared buffer.
--
-- The producers are named "A", "B", "C", etc.  Each producer is a thread which
-- will loop 5 times.  For each iteration, the producer thread will add its
-- character to a shared buffer.  For example, "Producer-B" will add 5 "B"s to
-- the shared buffer.  Since the 5 producer threads will run concurrently, the
-- characters will be added in an unpredictable order.  Regardless of the order,
-- however, there will be five "A"s, five "B"s, five "C"s, etc.
--
-- There are several consumers.  Each consumer is a thread which executes an
-- infinite loop.  During each iteration of its loop, a consumer will remove
-- whatever character is next in the buffer and will print it.
--
-- The shared buffer is a FIFO queue of characters.  The producers put characters
-- in one end and the consumers take characters out the other end.  Think of a
-- section of steel pipe.  The capacity of the buffer is limited to BUFFER_SIZE
-- characters.
--
-- This code illustrates the mechanisms required to synchronize the producers,
-- consumers, and the shared buffer.  Consumers must wait if the buffer is empty.
-- Producers must wait if the buffer is full.  Furthermore, the buffer is a shared
-- data structure.  (The buffer is implemented as an array with pointers to the
-- next position to add or remove characters.)  No two threads are allowed to
-- access these pointers simultaneously, or else errors may result.
--
-- To perform the synchronization, three semaphores are used.  The semaphore
-- called "bufferContents" is used to count the number of elements in the buffer.
-- It is used to force consumers to wait when the buffer is empty.  The
-- semaphore called "bufferSpaceLeft" is used to count the number of free spaces
-- left in the buffer.  It is used to make producers wait when the buffer is full.
-- The mutex called "bufferLock" is used as a lock to make sure that only
-- one thread at a time accesses the shared buffer.
--
-- To document what is happening, each producer will print a line when it adds
-- a character to the buffer.  The line printed will include the buffer contents
-- along with the name of the poducer.  Also, each time a consumer removes a
-- character from the buffer, it will print a line, showing the buffer contents
-- after the removal, along with the name of the consumer thread.  Each line of
-- output is formatted so that you can see the buffer growing and shrinking.  By
-- reading the output vertically, you can also see what each thread does.
--
-- The output itself can also be regarded as a shared resource.  In order to
-- ensure that all printing is done at the time the buffer is modified, the
-- print statements are done while the "bufferLock" is held.  Since only one
-- thread at a time can hold the "bufferLock", we are assured that several
-- consecutive print statements will be executed as a group, without output from
-- other threads being interleaved.

const
    BUFFER_SIZE = 5

var
    buffer: array [BUFFER_SIZE] of char
    bufferSize: int = 0
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

```
bufferNextIn: int = 0
bufferNextOut: int = 0
bufferContents: Semaphore = new Semaphore
bufferSpaceLeft: Semaphore = new Semaphore
bufferLock: Mutex = new Mutex
thArray: array [8] of Thread = new array of Thread { 8 of new Thread }

function ProducerConsumer ()

    buffer = new array of char {BUFFER_SIZE of '?'}
    bufferLock.Init ()
    bufferContents.Init (0)
    bufferSpaceLeft.Init (BUFFER_SIZE)
    print ("      ")

    thArray[0].Init ("Consumer-1          |      ")
    thArray[0].Fork (Consumer, 1)

    thArray[1].Init ("Consumer-2          |      ")
    thArray[1].Fork (Consumer, 2)

    thArray[2].Init ("Consumer-3          |      ")
    thArray[2].Fork (Consumer, 3)

    thArray[3].Init ("Producer-A          ")
    thArray[3].Fork (Producer, 1)

    thArray[4].Init ("Producer-B          ")
    thArray[4].Fork (Producer, 2)

    thArray[5].Init ("Producer-C          ")
    thArray[5].Fork (Producer, 3)

    thArray[6].Init ("Producer-D          ")
    thArray[6].Fork (Producer, 4)

    thArray[7].Init ("Producer-E          ")
    thArray[7].Fork (Producer, 5)

    ThreadFinish ()
endFunction

function Producer (myId: int)
    var
        i: int
        c: char = intToChar ('A' + myId - 1)
    for i = 1 to 5
        -- Perform synchronization
        bufferSpaceLeft.Wait ()
        bufferLock.Lock ()
        -- Add c to the buffer
        buffer [bufferNextIn] = c
        bufferNextIn = (bufferNextIn + 1) % BUFFER_SIZE
        bufferSize = bufferSize + 1
        -- Print a line showing the state
        PrintBuffer (c)
        -- Perform synchronization
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

```
        bufferContents.Signal()
        bufferLock.Unlock()
    endFor
endFunction

function Consumer (myId: int)
    var
        c: char
    while true
        -- Perform synchronization...
        bufferContents.Wait()
        bufferLock.Lock()
        -- Remove next character from the buffer
        c = buffer [bufferNextOut]
        bufferNextOut = (bufferNextOut + 1) % BUFFER_SIZE
        bufferSize = bufferSize - 1
        -- Print a line showing the state
        PrintBuffer (c)
        -- Perform synchronization...
        bufferSpaceLeft.Signal()
        bufferLock.Unlock()
    endWhile
endFunction

function PrintBuffer (c: char)
    --
    -- This method prints the buffer and what we are doing to it.  Each
    -- line should have
    --     <buffer> <threadname> <character involved>
    -- We want to print the buffer as it was *before* the operation;
    -- however, this method is called *after* the buffer has been modified.
    -- To achieve the right order, we print the operation first, skip to
    -- the next line, and then print the buffer.  Assuming we start by
    -- printing an empty buffer first, and we are willing to end the output
    -- in the middle of a line, this prints things in the desired order.
    --
    var
        i, j: int
    -- Print the thread name, which tells what we are doing.
    print (" ")
    print (currentThread.name) -- Will include right number of spaces after name
    printChar (c)
    nl ()
    -- Print the contents of the buffer.
    j = bufferNextOut
    for i = 1 to bufferSize
        printChar (buffer[j])
        j = (j + 1) % BUFFER_SIZE
    endFor
    -- Pad out with blanks to make things line up.
    for i = 1 to BUFFER_SIZE-bufferSize
        printChar (' ')
    endFor
endFunction
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

----- Dining Philosophers -----

```
-- This code is an implementation of the Dining Philosophers problem.  Each
-- philosopher is simulated with a thread.  Each philosopher thinks for a while
-- and then wants to eat.  Before eating, he must pick up both his forks.
-- After eating, he puts down his forks.  Each fork is shared between
-- two philosophers and there are 5 philosophers and 5 forks arranged in a
-- circle.
--
-- Since the forks are shared, access to them is controlled by a monitor
-- called "ForkMonitor".  The monitor is an object with two "entry" methods:
--     PickupForks (phil)
--     PutDownForks (phil)
-- The philosophers are numbered 0 to 4 and each of these methods is passed an integer
-- indicating which philosopher wants to pickup (or put down) the forks.
-- The call to "PickUpForks" will wait until both of his forks are
-- available.  The call to "PutDownForks" will never wait and may also
-- wake up threads (i.e., philosophers) who are waiting.
--
-- Each philosopher is in exactly one state: HUNGRY, EATING, or THINKING.  Each time
-- a philosopher's state changes, a line of output is printed.  The output is organized
-- so that each philosopher has column of output with the following code letters:
--     E      -- eating
--     .      -- thinking
--     blank  -- hungry (i.e., waiting for forks)
-- By reading down a column, you can see the history of a philosopher.
--
-- The forks are not modeled explicitly.  A fork is only picked up
-- by a philosopher if he can pick up both forks at the same time and begin
-- eating.  To know whether a fork is available, it is sufficient to simply
-- look at the status's of the two adjacent philosophers.  (Another way to state
-- the problem is to forget about the forks altogether and stipulate that a
-- philosopher may only eat when his two neighbors are not eating.)

enum HUNGRY, EATING, THINKING
var
    mon: ForkMonitor
    philosopher: array [5] of Thread = new array of Thread {5 of new Thread }

function DiningPhilosophers ()

    print ("Plato\n")
    print ("    Sartre\n")
    print ("        Kant\n")
    print ("            Nietzsche\n")
    print ("                Aristotle\n")

    mon = new ForkMonitor
    mon.Init ()
    mon.PrintAllStatus ()

    philosopher[0].Init ("Plato")
    philosopher[0].Fork (PhilosphizeAndEat, 0)

    philosopher[1].Init ("Sartre")
    philosopher[1].Fork (PhilosphizeAndEat, 1)
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

```
philosopher[2].Init ("Kant")
philosopher[2].Fork (PhilosphizeAndEat, 2)

philosopher[3].Init ("Nietzsche")
philosopher[3].Fork (PhilosphizeAndEat, 3)

philosopher[4].Init ("Aristotle")
philosopher[4].Fork (PhilosphizeAndEat, 4)

endFunction

function PhilosphizeAndEat (p: int)
--
-- The parameter "p" identifies which philosopher this is.
-- In a loop, he will think, acquire his forks, eat, and
-- put down his forks.
--
var
  i: int
for i = 1 to 7
  -- Now he is thinking
  mon. PickupForks (p)
  -- Now he is eating
  mon. PutDownForks (p)
endFor
endFunction

class ForkMonitor
  superclass Object
  fields
    monitorLock: Mutex          -- The monitor lock
    status: array [5] of int     -- For each philosopher: HUNGRY, EATING, or
    THINKING
    startEating: array [5] of Condition -- Signaled when eating can begin
  methods
    Init ()
    PickupForks (p: int)        -- An external "entry" method
    PutDownForks (p: int)       -- An external "entry" method
    CheckAboutEating (p: int)   -- Internal to the monitor
    PrintAllStatus ()
endClass

behavior ForkMonitor

method Init ()
--
-- Initialize so that all philosophers are THINKING. Also create
-- the monitor lock and the 5 condition variables.
--
var i: int
status = new array of int { 5 of THINKING }
startEating = new array [5] of Condition { 5 of new Condition }
for i = 0 to 4
  startEating[i].Init ()
endFor
monitorLock = new Mutex
monitorLock.Init ()
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

```
endMethod

method PickupForks (p: int)
--
-- This method is called when philosopher 'p' is wants to eat.
-- Change his status to HUNGRY and then see if he can begin eating.
-- If he was not able to begin immediately, then this thread must
-- wait.
--
monitorLock.Lock ()
status [p] = HUNGRY
self.PrintAllStatus ()
self.CheckAboutEating (p)
if status [p] != EATING
    startEating [p].Wait (& monitorLock)
endif
monitorLock.Unlock ()
endMethod

method PutDownForks (p: int)
--
-- This method is called when the philosopher 'p' is done eating.
-- Change his status. Also, this might make it possible for his
-- left and right neighbors to begin eating, so check on them.
--
monitorLock.Lock ()
status [p] = THINKING
self.PrintAllStatus ()
self.CheckAboutEating ((p+1) % 5)
self.CheckAboutEating ((p-1) % 5)
monitorLock.Unlock ()
endMethod

method CheckAboutEating (p: int)
--
-- See if the p-th philosopher should begin eating. He should begin
-- if he is HUNGRY and if his left and right neighbors are not eating.
-- If so, change his status to EATING. Also, it could be that this
-- philosopher's thread was waiting; signal that thread so he can
-- resume execution.
--
if status [p] == HUNGRY &&
    status [(p+1) % 5] != EATING &&
    status [(p-1) % 5] != EATING
    status [p] = EATING
    self.PrintAllStatus ()
    startEating [p].Signal (& monitorLock)
endif
endMethod

method PrintAllStatus ()
--
-- Print a single line showing the status of all philosophers.
--     '.' means thinking
--     ' ' means hungry
--     'E' means eating
-- Note that this method is internal to the monitor. Thus, when
```

Project 2 – Solution to Producer/Consumer and Dining Philosophers

```
-- it is called, the monitor lock will already have been acquired
-- by the thread. Therefore, this method can never be re-entered,
-- since only one thread at a time may execute within the monitor.
-- Consequently, printing is safe. This method calls the "print"
-- routine several times to print a single line, but these will all
-- happen without interruption.
--
var
  p: int
for p = 0 to 4
  switch status [p]
  case HUNGRY:
    print (" ")
    break
  case EATING:
    print ("E ")
    break
  case THINKING:
    print (". ")
    break
  endSwitch
endFor
nl ()
endMethod
```