Producer Consumer Problem

Introduction

This algorithm uses a monitor to solve the producer/consumer (or bounded-buffer) problem.

Algorithm

```
1   buffer: monitor ;
2   var     slots: array [0..n-1] of item;
3                  count, in, out: integer;
4                  notempty, notfull: condition;
5   procedure entry deposit(data: item);
6   begin
7           if count = n then
8                  notfull.wait ;
9           slots[in] := data;
10          in := in + 1 mod n;
11          count := count + 1;
12          notempty.signal ;
13  end ;
14  procedure entry extract(var data: item);
15  begin
16          if count = 0 then
17                 notempty.wait ;
18          data := slots[out];
19          out := out + 1 mod n;
20          count := count – 1;
21          notfull.signal ;
22  end ;
23  begin
24          count := 0; in := 0; out := 0;
25  end .
```

Comments

lines 2-4	Here, `slots` is the actual buffer, `count` the number of items in the buffer, and `in` and `out` the indices of the next element of `slots` where a deposit is to be made or from which an extraction is to be made. There are two conditions we care about: if the buffer is not full (represented by the condition variable `notfull`), and if the buffer is not empty (represented by the condition variable `notempty`).
line 5	The keyword **`entry`** means that this procedure may be called from outside the monitor. It is called by placing the name of the monitor first, then a period, then the function name; so, `buffer.deposit(…)`.
lines 7-8	This code checks to see if there is room in the buffer for a new item. If not, the process blocks on the condition `notfull`; when some other process does extract an element from the buffer, then there will be room and that process will signal on the condition `notfull`, allowing the blocked one to proceed. Note that while blocked on this condition, other processes may access procedures within the monitor.
lines 9-11	This code actually deposits the item into the buffer. Note that the monitor guarantees mutual exclusion.
line 12	Just as a producer will block on a full buffer, a consumer will block on an empty one. This indicates to any such consumer process that the buffer is no longer empty, and unblocks exactly one of them. If there are no blocked consumers, this is effectively a no-op.

line 14	As with the previous procedure, this is called from outside the monitor by `buffer.extract(…)`.
lines 16-17	This code checks to see if there is any unconsumed item in the buffer. If not, the process blocks on the condition *notempty*; when some other process does deposit an element in the buffer, then there will be something for the consumer to extract and that producer process will signal on the condition *notempty*, allowing the blocked one to proceed. Note that while blocked on this condition, other processes may access procedures within the monitor.
lines 18-20	This code actually extracts the item from the buffer. Note that the monitor guarantees mutual exclusion.
line 21	Just as a consumer will block on an empty buffer, a producer will block on a full one. This indicates to any such producer process that the buffer is no longer full, and unblocks exactly one of them. If there are no blocked producers, this is effectively a no-op.
lines 23-25	This is the initialization part.