**Synchronizing processes and resources part 2: producer/consumer**

(Author: Klaus G. Muller; kgmuller at users.sourceforge.net)

In producer/consumer synchronizations, producer processes make items available to consumer processes. Examples are a message sender and a message receiver, or two machines working on items in sequence. The synchronization here must ensure that the consumer process does not consume more items than have been produced. If necessary, the consumer process is blocked (must wait) if no item is available to be consumed. Producer and consumer processes are coupled by a buffer to allow asynchronous production and consumption. The buffer can be bounded (have a capacity limit) or unbounded (be able to store an unlimited number of items).

The structure of a producer/consumer synchronization with an unbounded buffer looks like this in SimPy:

|  |
| --- |
| **class** Producer(Process):     **def** produce(self):         **while** True:             **yield** hold,self,productionTime            *#produce an item*             *put item into buffer*             **if** consumer.passive():                       *#if consumer waiting for item,*                  reactivate(consumer)                   #       *wake it up*  **class** Consumer(Process):     **def** consume(self):         **while** True:             **if** **not** buffer:                                     #*if no item in buffer,*                 **yield** passivate,self                        #      *wait for item*             **while** buffer:                                     #*while items in buffer,*                 *take item out of buffer* **yield** hold,self,consumptionTime       #      *consume item*  buffer=[ ] producer=Producer() consumer=Consumer() |

**Note: this only works for the coordination/synchronization between one producer and one consumer.** With several consumers working in parallel, one consumer could erroneously try to take an item out of the buffer while another process is consuming the last item in the buffer.

Here is a simple yet complete model with producer/consumer synchronization with an unbounded buffer:

|  |
| --- |
| """  Simple producer/consumer model with unbounded buffer  """  from SimPy.Simulation import \*  import random  class Producer(Process):  def produce(self):  self.produced=0  while True:  yield hold,self,random.uniform(1.0,5.0) #produce  buffer.append(Item()) #put item into buffer  if cons.passive(): #reactivate consumer if passive  reactivate(cons)  print "%s item produced"%now()  self.produced+=1  class Consumer(Process):  def consume(self):  self.consumed=0  while True:  if not buffer: #if buffer empty, passivate  yield passivate,self  while buffer: #while items in buffer  buffer.pop(0) #take item out of buffer  self.consumed+=1 #consume item  print "%s item consumed"%now()  yield hold,self,random.uniform(1.0,4.9)  class Item:  pass  buffer=[]  produced=consumed=0  initialize()  prod=Producer("Producer")  activate(prod,prod.produce())  cons=Consumer("Consumer")  activate(cons,cons.consume())  simulate(until=20)  print "produced: %s, consumed: %s, in buffer: %s"\  %(prod.produced,cons.consumed,len(buffer))  **OUTPUT**:  1.18998162499 item produced 1.18998162499 item consumed 3.04978978053 item produced 4.67639404396 item consumed 7.91845737849 item produced 7.91845737849 item consumed 9.39241701538 item produced 10.7954940036 item produced 12.5224683604 item consumed 14.2282703847 item produced 14.2545615616 item consumed 16.3758414166 item consumed 17.4764134234 item produced 17.8687107056 item consumed 19.9345355896 item produced produced: 8, consumed: 7, in buffer: 1 |

Clearly, if the buffer access must be in mutual exclusion between producer putting and consumer getting, the mutual exclusion synchronization construct***yield*** *request/****yield*** *release* must be added in both processes. An example where this may be necessary is a sender process putting messages into shared memory from which a receiver process retrieves them.

What about buffers with limited capacity (bounded buffers)? Here, the producer process must not put items into a full buffer. The producer must wait until the consumer process has removed an item. The consumer must then wake up the producer.

Here is the general SimPy structure for producer/consumer synchronization with a bounded buffer:

|  |
| --- |
| **class Producer(Process):     def produce(self):         while True:             yield hold,self,productionTime            #produce an item             put item into buffer             if consumer.passive():                       #if consumer waiting for item,                  reactivate(consumer)                   #       wake it up  class Consumer(Process):     def consume(self):         while True:             if not buffer:                                     #if no item in buffer,                 yield passivate,self                        #      wait for item             while buffer:                                     #while items in buffer,                 take item out of buffer                 yield hold,self,consumptionTime       #      consume item  buffer=[ ] producer=Producer() consumer=Consumer()** |
| **class** Producer(Process):     **def** produce(self):         **while** True:             **if** len(buffer)==bufferSize:                       #*if buffer full,*                  **yield** passivate, self                           #     *wait for space to be freed*             **yield** hold**,**self**,**productionTime  #*produce an item             put item into buffer*             **if** consumer.passive():                 reactivate(consumer)  **class** Consumer(Process):     **def** consume(self):         **while** True:             **if not** buffer:                 **yield** passivate self             **while** buffer:                 *take item out of buffer*                 **if** producer.passive():                             #*if producer waiting for space,* reactivate(producer)                         #     *wake it up* **yield** hold,self,consumptionTime             #*consume item*  buffer=[ ] bufferSize=x consumer=Consumer() producer=Producer() |

Again, this is only a valid construction for one producer and one consumer.

Better, more flexible and robust solutions to the Producer/Consumer problem are possible with classes *Level* and *Store*, and the new **yield** statements '**yield** put' and '**yield** get', all introduced in SimPy 1.7.