Advanced Features of the SimPy Language

Norm Matloff

January 27, 2008
©2006-2008, N.S. Matloff

Contents

1 Overview

In this document we present several advanced features of the SimPy language. These will make your SimPy programming more convenient and enjoyable. In small programs, use of some of these features will produce a modest but worthwhile reduction on programming effort and increase in program clarity. In large programs, the savings add up, and can make a very significant improvement.

2 Use of SimPy's cancel() Function

In many simulation programs, a thread is waiting for one of two events; whichever occurs first will trigger a resumption of execution of the thread. The thread will typically want to ignore the other, later-occurring event. We can use SimPy's **cancel()** function to cancel the later event.

2.1 Example: Network Timeout

An example of this is in the program **TimeOut.py**. The model consists of a network node which transmits but also sets a **timeout** period, as follows: After sending the message out onto the network, the node waits for an acknowledgement from the recipient. If an acknowledgement does not arrive within a certain specified period of time, it is assumed that the message was lost, and it will be sent again. We wish to determine the percentage of attempted transmissions which result in timeouts.

The timeout period is assumed to be 0.5, and acknowledgement time is assumed to be exponentially distributed with mean 1.0. Here is the code:

```
#!/usr/bin/env python

# Introductory SimPy example to illustrate the modeling of "competing
# events" such as timeouts, especially using SimPy's cancel() method.  A
# network node sends a message but also sets a timeout period; if the
# node times out, it assumes the message it had sent was lost, and it
# will send again.  The time to get an acknowledgement for a message is
# exponentially distributed with mean 1.0, and the timeout period is
# 0.5.  Immediately after receiving an acknowledgement, the node sends
# out a new message.

# We find the proportion of messages which timeout.  The output should
# be about 0.61.

# the main classes are:

#   Node, simulating the network node, with our instance being Nd
#   TimeOut, simulating a timeout timer, with our instance being TO
#   Acknowledge, simulating an acknowledgement, with our instance being ACK

# overview of program design:

#   Nd acts as the main "driver," with a loop that continually creates
#   TimeOuts and Acknowledge objects, passivating itself until one of
#   those objects' events occurs; if for example the timeout occurs
#   before the acknowledge, the TO object will reactivate Nd and cancel
#   the ACK object's event, and vice versa

from SimPy.Simulation import *
from random import Random,expovariate
```

```

class Node(Process):
   def __init__(self):
      Process.__init__(self)
      self.NMsgs = 0  # number of messages sent
      self.NTimeOuts = 0  # number of timeouts which have occurred
      # ReactivatedCode will be 1 if timeout occurred, 2 ACK if received
      self.ReactivatedCode = None
   def Run(self):
      while 1:
         self.NMsgs += 1
         # set up the timeout
         G.TO = TimeOut()
         activate(G.TO,G.TO.Run())
         # set up message send/ACK
         G.ACK = Acknowledge()
         activate(G.ACK,G.ACK.Run())
         yield passivate,self
         if self.ReactivatedCode == 1:
            self.NTimeOuts += 1
         self.ReactivatedCode = None

class TimeOut(Process):
   TOPeriod = 0.5
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      yield hold,self,TimeOut.TOPeriod
      G.Nd.ReactivatedCode = 1
      reactivate(G.Nd)
      self.cancel(G.ACK)

class Acknowledge(Process):
   ACKRate = 1/1.0
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      yield hold,self,G.Rnd.expovariate(Acknowledge.ACKRate)
      G.Nd.ReactivatedCode = 2
      reactivate(G.Nd)
      self.cancel(G.TO)

class G:  # globals
   Rnd = Random(12345)
   Nd = Node()

def main():
   initialize()
   activate(G.Nd,G.Nd.Run())
   simulate(until=10000.0)
   print 'the percentage of timeouts was', float(G.Nd.NTimeOuts)/G.Nd.NMsgs

if __name__ == '__main__':  main()
```

The main driver here is a class **Node**, whose PEM code includes the lines

```
while 1:
   self.NMsgs += 1
   G.TO = TimeOut()
   activate(G.TO,G.TO.Run())
   G.ACK = Acknowledge()
   activate(G.ACK,G.ACK.Run())
   yield passivate,self
   if self.ReactivatedCode == 1:
      self.NTimeOuts += 1
   self.ReactivatedCode = None
```

The node creates an object **G.TO** of our **TimeOut** class, which will simulate a timeout period, and creates an object **G.ACK** of our **Acknowledge** class to simulate a transmission and acknowledgement. Then the node passivates itself, allowing **G.TO** and **G.ACK** to do their work. One of them will finish first, and then will call SimPy's **reactivate()** function to "wake up" the suspended node. The node senses whether it was a timeout or acknowledgement which woke it up, via the variable **ReactivatedCode**, and then updates its timeout count accordingly.

Here's what **TimeOut.Run()** does:

```
yield hold,self,TimeOut.TOPeriod
G.Nd.ReactivatedCode = 1
reactivate(G.Nd)
self.cancel(G.ACK)
```

It holds a random timeout time, then sets a flag in **Nd** to let the latter know that it was the timeout which occurred first, rather than the acknowledgement. Then it reactivates **Nd** and cancels **ACK**. **ACK** of course has similar code for handling the case in which the acknowledgement occurs before the timeout.

Note that in our case here, we want the thread to go out of existence when canceled. The **cancel()** function does not make that occur. It simply removes the pending events associated with the given thread. The thread is still there.

However, here the **TO** and **ACK** threads will go out of existence anyway, for a somewhat subtle reason:[1] Think of what happens when we finish one iteration of the **while** loop in **main()**. A new object of type **TimeOut** will be created, and then assigned to **G.TO**. That means that the **G.TO** no longer points to the old **TimeOut** object, and since nothing else points to it either, the Python interpreter will now **garbage collect** that old object.

2.2 Example: Machine with Breakdown

Here is another example of **cancel()**:

```
#!/usr/bin/env python

# JobBreak.py

# One machine, which sometimes breaks down.  Up time and repair time are
# exponentially distributed.  There is a continuing supply of jobs
# waiting to use the machine, i.e. when one job finishes, another
# immediately begins.  When a job is interrupted by a breakdown, it
# resumes "where it left off" upon repair, with whatever time remaining
# that it had before.

from SimPy.Simulation import *
from random import Random,expovariate

import sys

class G:  # globals
   CurrentJob = None
   Rnd = Random(12345)
   M = None  # our one machine

class Machine(Process):
```

[1]Thanks to Travis Grathwell for pointing this out.

```
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         UpTime = G.Rnd.expovariate(Machine.UpRate)
         yield hold,self,UpTime
         CJ = G.CurrentJob
         self.cancel(CJ)
         NewNInts = CJ.NInts + 1
         NewTimeLeft = CJ.TimeLeft - (now()-CJ.LatestStart)
         RepairTime = G.Rnd.expovariate(Machine.RepairRate)
         yield hold,self,RepairTime
         G.CurrentJob = Job(CJ.ID,NewTimeLeft,NewNInts,CJ.OrigStart,now())
         activate(G.CurrentJob,G.CurrentJob.Run())

class Job(Process):
   ServiceRate = None
   NDone = 0  # jobs done so far
   TotWait = 0.0  # total wait for those jobs
   NNoInts = 0  # jobs done so far that had no interruptions
   def __init__(self,ID,TimeLeft,NInts,OrigStart,LatestStart):
      Process.__init__(self)
      self.ID = ID
      self.TimeLeft = TimeLeft  # amount of work left for this job
      self.NInts = NInts  # number of interruptions so far
      # time this job originally started
      self.OrigStart = OrigStart
      # time the latest work period began for this job
      self.LatestStart = LatestStart
   def Run(self):
      yield hold,self,self.TimeLeft
      # job done
      Job.NDone += 1
      Job.TotWait += now() - self.OrigStart
      if self.NInts == 0:  Job.NNoInts += 1
      # start the next job
      SrvTm = G.Rnd.expovariate(Job.ServiceRate)
      G.CurrentJob = Job(G.CurrentJob.ID+1,SrvTm,0,now(),now())
      activate(G.CurrentJob,G.CurrentJob.Run())

def main():
   Job.ServiceRate = float(sys.argv[1])
   Machine.UpRate = float(sys.argv[2])
   Machine.RepairRate = float(sys.argv[3])
   initialize()
   SrvTm = G.Rnd.expovariate(Job.ServiceRate)
   G.CurrentJob = Job(0,SrvTm,0,0.0,0.0)
   activate(G.CurrentJob,G.CurrentJob.Run())
   G.M = Machine()
   activate(G.M,G.M.Run())
   MaxSimtime = float(sys.argv[4])
   simulate(until=MaxSimtime)
   print 'mean wait:', Job.TotWait/Job.NDone
   print '% of jobs with no interruptions:', \
      float(Job.NNoInts)/Job.NDone

if __name__ == '__main__': main()
```

Here we have one machine, with occasional breakdown, but we also keep track of the number of jobs done. See the comments in the code for details.

Here we have set up a class **Job**. When a new job starts service, an instance of this class is set up to model that job. If its service then runs to completion without interruption, fine. But if the machine breaks down in the midst of service, this instance of the **Job** class will be discarded, and a new instance will later be created when this job resumes service after the repair. In other words, each object of the class **Job** models one job to be done, but it can be either a brand new job or the resumption of an interrupted job.

Let's take a look at **Job.Run()**:

```
yield hold,self,self.TimeLeft
Job.NDone += 1
Job.TotWait += now() - self.OrigStart
if self.NInts == 0:  Job.NNoInts += 1
SrvTm = G.Rnd.expovariate(Job.ServiceRate)
G.CurrentJob = Job(G.CurrentJob.ID+1,SrvTm,0,now(),now())
activate(G.CurrentJob,G.CurrentJob.Run())
```

This looks innocuous enough. We hold for the time it takes to finish the job, then update our totals, and launch the next job. What is not apparent, though, is that we may actually never reach that second line,

```
Job.NDone += 1
```

The reason for this is that the machine may break down before the job finishes. In that case, what we have set up is that **Machine.Run()** will cancel the pending job completion event,

```
self.cancel(CJ)
```

simulate the repair of the machine,

```
RepairTime = G.Rnd.expovariate(Machine.RepairRate)
yield hold,self,RepairTime
```

and then create a new instance of **Job** which will simulate the processing of the remainder of the interrupted job (which may get interrupted too):

```
NewNInts = CJ.NInts + 1
NewTimeLeft = CJ.TimeLeft - (now()-CJ.LatestStart)
...
G.CurrentJob = Job(CJ.ID,NewTimeLeft,NewNInts,CJ.OrigStart,now())
activate(G.CurrentJob,G.CurrentJob.Run())
```

There are other ways of doing this, in particular by using SimPy's **interrupt()** and **interrupted()** functions, but we defer this to Section 3.

3 Job Interruption

SimPy allows one thread to interrupt another, which can be very useful.

3.1 Example: Machine Breakdown Again

In Section 2.2 we had a program **JobBreak.py**, which modeled a machine with breakdown on which we collected job time data. We presented that program as an example of **cancel()**. However, it is much more easily handeled via the function **interrupt()**. Here is a new version of the program using that function:

```python
#!/usr/bin/env python

# JobBreakInt.py:  illustration of interrupt() and interrupted()

# One machine, which sometimes breaks down.  Up time and repair time are
# exponentially distributed.  There is a continuing supply of jobs
# waiting to use the machine, i.e. when one job finishes, the next
# begins.  When a job is interrupted by a breakdown, it resumes "where
# it left off" upon repair, with whatever time remaining that it had
# before.

from SimPy.Simulation import *
from random import Random,expovariate

import sys

class G:  # globals
   CurrentJob = None
   Rnd = Random(12345)
   M = None  # our one machine

class Machine(Process):
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      from SimPy.Simulation import _e
      while 1:
         UpTime = G.Rnd.expovariate(Machine.UpRate)
         yield hold,self,UpTime
         self.interrupt(G.CurrentJob)
         RepairTime = G.Rnd.expovariate(Machine.RepairRate)
         yield hold,self,RepairTime
         reactivate(G.CurrentJob)

class Job(Process):
   ServiceRate = None
   NDone = 0  # jobs done so far
   TotWait = 0.0  # total wait for those jobs
   NNoInts = 0  # jobs done so far that had no interruptions
   NextID = 0
   def __init__(self):
      Process.__init__(self)
      self.ID = Job.NextID
      Job.NextID += 1
      # amount of work left for this job
      self.TimeLeft = G.Rnd.expovariate(Job.ServiceRate)
      self.NInts = 0  # number of interruptions so far
      # time this job originally started
      self.OrigStart = now()
      # time the latest work period began for this job
      self.LatestStart = now()
   def Run(self):
      from SimPy.Simulation import _e
      while True:
         yield hold,self,self.TimeLeft
         # did the job run to completion?
         if not self.interrupted(): break
         self.NInts += 1
         self.TimeLeft -= now() - self.LatestStart
         yield passivate,self  # wait for repair
         self.LatestStart = now()
      Job.NDone += 1
      Job.TotWait += now() - self.OrigStart
      if self.NInts == 0:  Job.NNoInts += 1
      # start the next job
      G.CurrentJob = Job()
      activate(G.CurrentJob,G.CurrentJob.Run())
```

```
def main():
   Job.ServiceRate = float(sys.argv[1])
   Machine.UpRate = float(sys.argv[2])
   Machine.RepairRate = float(sys.argv[3])
   initialize()
   G.CurrentJob = Job()
   activate(G.CurrentJob,G.CurrentJob.Run())
   G.M = Machine()
   activate(G.M,G.M.Run())
   MaxSimtime = float(sys.argv[4])
   simulate(until=MaxSimtime)
   print 'mean wait:', Job.TotWait/Job.NDone
   print '% of jobs with no interruptions:', \
      float(Job.NNoInts)/Job.NDone

if __name__ == '__main__': main()
```

The first key part of **Machine.Run()** is

```
yield hold,self,UpTime
self.interrupt(G.CurrentJob)
```

A call to **interrupt()** cancels the pending **yield hold** operation of its "victim," i.e. the thread designated in the argument. [2] A new artificial event will be created for the victim, with event time being the current simulated time, **now()**. The caller does not lose control of the CPU, and continues to execute, but when it hits its next **yield** statement (or **passivate()** etc.) and thus loses control of the CPU, the victim will probably be next to run, as its (new, artificial) event time will be the current time.

In our case here, at the time

```
self.interrupt(G.CurrentJob)
```

is executed by the **Machine** thread, the current job is in the midst of being serviced. The call interrupts that service, to reflect the fact that the machine has broken down. At this point, the current job's event is canceled, with the artificial event being created as above. The current job's thread won't run yet, and the **Machine** thread will continue. But when the latter reaches the line

```
yield hold,self,RepairTime
```

the **Machine** thread loses control of the CPU and the current job's thread runs. The latter executes

```
if not self.interrupted(): break
self.NInts += 1
self.TimeLeft -= now() - self.LatestStart
yield passivate,self  # wait for repair
```

The interruption will be sensed by **self.interrupted()** returning True. The job thread will then do the proper bookkeeping, and then passivate itself, waiting for the machine to come back up. When the latter event occurs, the machine's thread executes

```
reactivate(G.CurrentJob)
```

[2] The function **interrupt()** should not be called unless the thread to be interrupted is in the midst of **yield hold**.

to get the interrupted job started again.

Note that a job may go through multiple cycles of run, interruption, run, interruption, etc., depending on how many breakdowns the machine has during the lifetime of this job. This is the reason for the **while** loop in **Job.Run**():

```
while True:
   yield hold,self,self.TimeLeft
   # did the job run to completion?
   if not self.interrupted(): break
   self.NInts += 1
   self.TimeLeft -= now() - self.LatestStart
   yield passivate,self  # wait for repair
   self.LatestStart = now()
Job.NDone += 1
Job.TotWait += now() - self.OrigStart
...
```

In the job's final cycle (which could be its first), the **yield hold** will not be interrupted. In this case the call to **interrupted()** will inform the thread that it had *not* been interrupted. The loop will be exited, the final bookkeeping for this job will be done, and the next job will be started.

By the way, we did not have to have our instance variable **TimeLeft** in **Job**. SimPy's **Process** class has its own built-in instance variable **interruptLeft** which records how much time in the **yield hold** had been remaining at the time of the interruption.

3.2 Example: Network Timeout Again

Use of interrupts makes our old network node acknowledgement/timeout program **TimeOut.py** in Section 2.1 considerably simpler:

```
#!/usr/bin/env python

# TimeOutInt.py

# Same as TimeOut.py but using interrupts.  A network node sends a message
# but also sets a timeout period; if the node times out, it assumes the
# message it had sent was lost, and it will send again.  The time to get
# an acknowledgement for a message is exponentially distributed with
# mean 1.0, and the timeout period is 0.5.  Immediately after receiving
# an acknowledgement, the node sends out a new message.

# We find the proportion of messages which timeout.  The output should
# be about 0.61.

from SimPy.Simulation import *
from random import Random,expovariate

class Node(Process):
   def __init__(self):
      Process.__init__(self)
      self.NMsgs = 0  # number of messages sent
      self.NTimeOuts = 0  # number of timeouts which have occurred
   def Run(self):
      from SimPy.Simulation import _e
      while 1:
         self.NMsgs += 1
         # set up the timeout
         G.TO = TimeOut()
```

```
         activate(G.TO,G.TO.Run())
         # wait for ACK, but could be timeout
         yield hold,self,G.Rnd.expovariate(1.0)
         if self.interrupted():
            self.NTimeOuts += 1
         else:  self.cancel(G.TO)

class TimeOut(Process):
   TOPeriod = 0.5
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      from SimPy.Simulation import _e
      yield hold,self,TimeOut.TOPeriod
      self.interrupt(G.Nd)

class G:  # globals
   Rnd = Random(12345)
   Nd = Node()

def main():
   initialize()
   activate(G.Nd,G.Nd.Run())
   simulate(until=10000.0)
   print 'the percentage of timeouts was', float(G.Nd.NTimeOuts)/G.Nd.NMsgs

if __name__ == '__main__':  main()
```

Use of interrupts allowed us to entirely eliminate our old **ACK** class. Moreover, the code looks more natural now, as a timeout could be thought of as "interrupting" the node.

4 Interthread Synchronization

In our introductory SimPy document, in cases in which one thread needed to wait for some other thread to take some action,[3] we made use of **passivate()** and **reactivate()**. Those can be used in general, but more advanced constructs would make our lives easier.

For example, suppose many threads are waiting for the same action to occur. The thread which triggered that action would then have to call **reactivate()** on all of them. Among other things, this would mean we would have to have code which kept track of which threads were waiting. We could do that, but it would be nicer if we didn't have to.

In fact, actions like **yield waitevent** alleviate us of that burden. This makes our code easier to write and maintain, and easier to read.

4.1 Example: Yet Another Machine Breakdown Model

Below is an example, again modeling a machine repair situation. It is similar to **MachRep3.py** from our introductory document, but with R machines instead of two, and a policy that the repairperson is called if the number of operating machines falls below K.

[3]I've used the word *action* here rather than *event*, as the latter term refers to items in SimPy's internal event list, generated by **yield hold** operations. But this won't completely remove the confusion, as the SimPy keyword **waitevent** will be introduced below. But again, that term will refer to what I'm describing as *actions* here. The official SimPy term is a *SimEvent*.

```
#!/usr/bin/env python

# MachRep4.py

# SimPy example:  R machines, which sometimes break down.  Up time is
# exponentially distributed with rate UpRate, and repair time is
# exponentially distributed with rate RepairRate.  The repairperson is
# summoned when fewer than K of the machines are up, and reaches the
# site after a negligible amount of time.  He keeps repairing machines
# until there are none that need it, then leaves.

# usage:  python MachRep4.py R UpRate RepairRate K MaxSimTime

from SimPy.Simulation import *
from random import Random,expovariate

class G:  # globals
   Rnd = Random(12345)
   RepairPerson = Resource(1)
   RepairPersonOnSite = False
   RPArrive = SimEvent()

class MachineClass(Process):
   MachineList = []  # list of all objects of this class
   UpRate = None  # reciprocal of mean up time
   RepairRate = None  # reciprocal of mean repair time
   R = None  # number of machines
   K = None  # threshold for summoning the repairperson
   TotalUpTime = 0.0  # total up time for all machines
   NextID = 0  # next available ID number for MachineClass objects
   NUp = 0  # number of machines currently up
   # create an event to signal arrival of repairperson
   def __init__(self):
      Process.__init__(self)
      self.StartUpTime = None  # time the current up period started
      self.ID = MachineClass.NextID   # ID for this MachineClass object
      MachineClass.NextID += 1
      MachineClass.MachineList.append(self)
      MachineClass.NUp += 1  # start in up mode
   def Run(self):
      from SimPy.Simulation import _e
      while 1:
         self.StartUpTime = now()
         yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
         MachineClass.TotalUpTime += now() - self.StartUpTime
         MachineClass.NUp -= 1
         # if the repairperson is already onsite, just request him;
         # otherwise, check whether fewer than K machines are up
         if not G.RepairPersonOnSite:
            if MachineClass.NUp < MachineClass.K:
                  G.RPArrive.signal()
                  G.RepairPersonOnSite = True
            else: yield waitevent,self,G.RPArrive
         yield request,self,G.RepairPerson
         yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
         MachineClass.NUp += 1
         # if no more machines waiting for repair, dismiss repairperson
         if G.RepairPerson.waitQ == []:
            G.RepairPersonOnSite = False
         yield release,self,G.RepairPerson

def main():
   initialize()
   MachineClass.R = int(sys.argv[1])
   MachineClass.UpRate = float(sys.argv[2])
   MachineClass.RepairRate = float(sys.argv[3])
   MachineClass.K = int(sys.argv[4])
   for I in range(MachineClass.R):
```

```
      M = MachineClass()
      activate(M,M.Run())
   MaxSimtime = float(sys.argv[5])
   simulate(until=MaxSimtime)
   print 'proportion of up time was',  \
      MachineClass.TotalUpTime/(MachineClass.R*MaxSimtime)

if __name__ == '__main__':  main()
```

Here we make use of a new SimPy class, **SimEvent**:

```
RepairPersonOnSite = False
RPArrive = SimEvent()
```

We also set up a variable **RepairPersonOnSite** to keep track of whether the repairperson is currently available; more on this point below.

Here is the core code, executed when a machine goes down:

```
MachineClass.NUp -= 1
if not G.RepairPersonOnSite:
   if MachineClass.NUp < MachineClass.K:
         G.RPArrive.signal()
         G.RepairPersonOnSite = True
   else: yield waitevent,self,G.RPArrive
yield request,self,G.RepairPerson
```

If the repairperson is on site already, then we go straight to the **yield request** to queue up for repair. If the repairperson is not on site, and the number of working machines has not yet dropped below K, our machine executes **yield waitevent** on our action **G.RPArrive**, which basically passivates this thread. If on the other hand our machine's failure does make the number of working machines drop below K, we execute the **signal()** function, which reactivates all the machines which had been waiting.

Again, all of that could have been done via explicit **passivate()** and **reactivate()** calls, but it's much more convenient to let SimPy do that work for us, behind the scenes.

One of the member variables of **SimEvent** is **occurred**, which of course is a boolean variable stating whether the action has occurred yet. Note that as soon as a wait for an event finishes, this variable reverts to False. This is why we needed a separate variable above, **G.RepairPersonOnSite**.

4.2 Which Comes First?

In general thread terminology, we say that we **post** a signal when we call **signal()**. One of the issues to resolve when you learn any thread system concerns what happens when a signal is posted before any waits for it are executed. In many thread systems, that posting will be completely ignored, and subsequent waits will thus last forever, or at least until another signal is posted. This obviously can cause bugs and makes programming more difficult.

In SimPy it's the opposite: If a signal is posted first, before any waits are started, subsequent waits will return immediately. That was not an issue in this program, but it's important to keep in mind in general.

4.3 Waiting for Whichever Action Comes First

You can also use **yield waiteventj** to wait for several actions, producing a “whichever comes first” operation. To do this, instead of using the form

`yield waitevent,self` *action_name*

use

`yield waitevent,self` *tuple_or_list_of_action_names*

Then whenever a signal is invoked on any one of the specified actions occurs, all actions queued will be reactivated.

4.4 The yield queueevent Operation

This works just like **yield waitevent**, but when the signal is invoked, only the action at the head of the queue will be reactivated.

5 Advanced Use of the Resource Class

The default queuing **discipline**, i.e. priority policy, for the **Resource** class is First Come, First Served (FCFS). The alternative is to assign different priorities to threads waiting for the resource, which you do by the named argument **qType**. For example,

```
R = Resource(8,qType=PriorityQ)
```

creates a resource **R** with eight service units, the queue for which has priorities assigned. The priorities are specified in the **yield request** statement. For instance,

```
yield request,self,R,88
```

requests to use the resource **R**, with priority 88. The priorities are user-defined.

5.1 Example: Network Channel with Two Levels of Service

Below is an example of a model in which we use the non-FCFS version of **Resource**. Here we have a shared network channel on which both video and data are being transmitted. The two types of traffic act in complementary manners:

- We can tolerate a certain percentage of lost video packets, as small loss just causes a bit of jitter on the screen. But we can’t have any noticeable delay.
- We can tolerate a certain increase in delay for data packets. We won’t care about or even notice a small increase in delay. But we can’t lose packets.

Accordingly,

- We discard video packets that are too "old," with threshold being controlled by the design parameter L explained in the comments in the program below.
- We don't discard data packets.

For a fixed level of data traffic, we can for example use simulation to study the tradeoff arising from our choice of the value of L. Larger L means more lost video packets but smaller delay for data, and vice versa.

Here is the program:

```
#!/usr/bin/env python

# QoS.py:  illustration of non-FCFS priorities in Resource class

# Communications channel, shared by video and data.  Video packets
# arrive every 2.0 amount of time, and have transmission time 1.0.  Data
# packet interarrivals are exponentially distributed with rate DArrRate,
# and their transmission time is uniformaly distributed on {1,2,3,4,5}.
# Video packets have priority over data packets but the latter are not
# pre-emptable.  A video packet is discarded upon arrival if it would be
# sent L or more amount of time late.

# usage:  python QoS.py DArrRate L MaxSimTime

from SimPy.Simulation import *
from random import Random,expovariate

class G:  # globals
   Rnd = Random(12345)
   Chnl = None  # our one channel
   VA = None  # our one video arrivals process
   DA = None  # our one video arrivals process

class ChannelClass(Resource):
   def __init__(self):
      # note arguments to parent constructor:
      Resource.__init__(self,capacity=1,qType=PriorityQ)
      # if a packet is currently being sent, here is when transmit will end
      self.TimeEndXMit = None
      self.NWaitingVid = 0  # number of video packets in queue

class VidJob(Process):
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      Lost = False
      # if G.Chnl.TimeEndXMit is None, then no jobs in the system
      # now, so this job will start right away (handled below);
      # otherwise:
      if G.Chnl.TimeEndXMit != None:
         # first check for loss
         TimeThisPktStartXMit = G.Chnl.TimeEndXMit + G.Chnl.NWaitingVid
         if TimeThisPktStartXMit - now() > VidArrivals.L:
            Lost = True
            VidArrivals.NLost += 1
            return
      G.Chnl.NWaitingVid += 1
      yield request,self,G.Chnl,1  # higher priority
      G.Chnl.NWaitingVid -= 1
      G.Chnl.TimeEndXMit = now() + 0.999999999999
      yield hold,self,0.999999999999  # to avoid coding "ties"
```

```
      G.Chnl.TimeEndXMit = None
      yield release,self,G.Chnl

class VidArrivals(Process):
   L = None  # threshold for discarding packet
   NArrived = 0  # number of video packets arrived
   NLost = 0  # number of video packets lost
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         yield hold,self,2.0
         VidArrivals.NArrived += 1
         V = VidJob()
         activate(V,V.Run())

class DataJob(Process):
   def __init__(self):
      Process.__init__(self)
      self.ArrivalTime = now()
   def Run(self):
      yield request,self,G.Chnl,0  # lower priority
      XMitTime = G.Rnd.randint(1,6) - 0.000000000001
      G.Chnl.TimeEndXMit = now() + XMitTime
      yield hold,self,XMitTime
      G.Chnl.TimeEndXMit = None
      DataArrivals.NSent += 1
      DataArrivals.TotWait += now() - self.ArrivalTime
      yield release,self,G.Chnl

class DataArrivals(Process):
   DArrRate = None  #  data arrival rate
   NSent = 0  # number of video packets arrived
   TotWait = 0.0  # number of video packets lost
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         yield hold,self,G.Rnd.expovariate(DataArrivals.DArrRate)
         D = DataJob()
         activate(D,D.Run())

# def ShowStatus():
#    print 'time', now()
#    print 'current xmit ends at', G.Chnl.TimeEndXMit
#    print 'there are now',len(G.Chnl.waitQ), 'in the wait queue'
#    print G.Chnl.NWaitingVid, 'of those are video packets'

def main():
   initialize()
   VidArrivals.L = float(sys.argv[1])
   DataArrivals.DArrRate = float(sys.argv[2])
   G.Chnl = ChannelClass()
   G.VA = VidArrivals()
   activate(G.VA,G.VA.Run())
   G.DA = DataArrivals()
   activate(G.DA,G.DA.Run())
   MaxSimtime = float(sys.argv[3])
   simulate(until=MaxSimtime)
   print 'proportion of video packets lost:', \
      float(VidArrivals.NLost)/VidArrivals.NArrived
   print 'mean delay for data packets:', \
      DataArrivals.TotWait/DataArrivals.NSent

if __name__ == '__main__':  main()
```

We have chosen to make a subclass of **Resource** for channels. In doing so, we do have to be careful when our subclass' constructor calls **Resource**'s constructor:

```
Resource.__init__(self,capacity=1,qType=PriorityQ)
```

The named argument **capacity** is the number of resource units, which is 1 in our case. I normally don't name it in my **Resource** calls, as it is the first argument and thus doesn't need to be named, but in this case I've used the name for clarity. And of course I've put in the **qType** argument.

Here is where I set the priorities:

```
yield request,self,G.Chnl,1  # video
...
yield request,self,G.Chnl,0  # data
```

I chose the values 1 and 0 arbitrarily. Any values would have worked, as long as the one for video was higher, to give it a higher priority.

Note that I have taken transmission times to be 0.000000000001 lower than an integer, so as to avoid "ties," in which a transmission would end exactly when messages might arrive. This is a common issue when **yield hold** times are integers.

5.2 Example: Call Center

This program simulates the operation of a call-in advice nurse system, such as the one in Kaiser Permanente. The key issue here is that the number of servers (nurses) varies through time, as the policy here is to take nurses off the shift when the number of callers is light, and to add more nurses during periods of heavy usage.

```
#!/usr/bin/env python

# CallCtr.py:  simulation of call-in advice nurse system

#  patients call in, with exponential interarrivals with rate Lambda1;
#  they queue up for a number of advice nurses which varies through time
#  (initially MOL); service time is exponential with rate Lambda2; if the
#  system has been empty (i.e. no patients in the system, either being
#  served or in the queue) for TO amount of time, the number of nurses
#  is reduced by 1 (but it can never go below 1); a new TO period is then
#  begun; when a new patient call comes in, if the new queue length is
#  at least R the number of nurses is increased by 1, but it cannot go
#  above MOL; here the newly-arrived patient is counted in the queue
#  length

# usage:

#   python CallCtr.py MOL, R, TO, Lambda1, Lambda2, MaxSimtime, Debug

from SimPy.Simulation import *
from random import Random,expovariate
import sys
import PeriodicSampler

# globals
class G:
```

```
   Rnd = Random(12345)
   NrsPl = None  # nurse pool

class NursePool(Process):
   def __init__(self,MOL,R,TO):
      Process.__init__(self)
      self.Rsrc = Resource(capacity=MOL,qType=PriorityQ)  # the nurses
      self.MOL = MOL  # maximum number of nurses online
      self.R = R
      self.TO = TO
      self.NrsCurrOnline = 0  # current number of nurses online
      self.TB = None  # current timebomb thread, if any
      self.Mon = Monitor()  # monitors numbers of nurses online
      self.PrSm = PeriodicSampler.PerSmp(1.0,self.Mon,self.MonFun)
      activate(self.PrSm,self.PrSm.Run())
   def MonFun(self):
      return self.NrsCurrOnline
   def Wakeup(NrsPl,Evt):  # wake nurse pool manager
      reactivate(NrsPl)
      # state the cause
      NrsPl.WakingEvent = Evt
      if G.Debug:  ShowStatus(Evt)
   def StartTimeBomb(self):
      self.TB = TimeBomb(self.TO,self)
      activate(self.TB,self.TB.Run())
   def Run(self):
      self.NrsCurrOnline = self.MOL
      # system starts empty, so start timebomb
      self.StartTimeBomb()
      # this thread is a server, usually sleeping but occasionally being
      # wakened to handle an event:
      while True:
         yield passivate,self  # sleep until an event occurs:
         if self.WakingEvent == ’arrival’:
            # if system had been empty, cancel timebomb
            if PtClass.NPtsInSystem == 1:
               self.cancel(self.TB)
               self.TB = None
            else:  # check for need to expand pool
               # how many in queue, including this new patient?
               NewQL = len(self.Rsrc.waitQ) + 1
               if NewQL >= self.R and self.NrsCurrOnline < self.MOL:
                  # bring a new nurse online
                  yield release,self,self.Rsrc
                  self.NrsCurrOnline += 1
            continue  # go back to sleep
         if self.WakingEvent == ’departure’:
            if PtClass.NPtsInSystem == 0:
               self.StartTimeBomb()
            continue  # go back to sleep
         if self.WakingEvent == ’timebomb exploded’:
            if self.NrsCurrOnline > 1:
               # must take 1 nurse offline
               yield request,self,self.Rsrc,100
               self.NrsCurrOnline -= 1
            self.StartTimeBomb()
            continue  # go back to sleep

class TimeBomb(Process):
   def __init__(self,TO,NrsPl):
      Process.__init__(self)
      self.TO = TO  # timeout period
      self.NrsPl = NrsPl  # nurse pool
      self.TimeStarted = now()  # for debugging
   def Run(self):
      yield hold,self,self.TO
      NursePool.Wakeup(G.NrsPl,’timebomb exploded’)
```

```
class PtClass(Process):  # simulates one patient
   SrvRate = None  # service rate
   NPtsInSystem = 0
   Mon = Monitor()
   def __init__(self):
      Process.__init__(self)
      self.ArrivalTime = now()
   def Run(self):
      # changes which trigger expansion or contraction of the nurse pool
      # occur at arrival points and departure points
      PtClass.NPtsInSystem += 1
      NursePool.Wakeup(G.NrsPl,'arrival')
      # dummy to give nurse pool thread a chance to wake up, possibly
      # change the number of nurses, and reset the timebomb:
      yield hold,self,0.00000000000001
      yield request,self,G.NrsPl.Rsrc,1
      if G.Debug:  ShowStatus('service starts')
      yield hold,self,G.Rnd.expovariate(PtClass.SrvRate)
      yield release,self,G.NrsPl.Rsrc
      PtClass.NPtsInSystem -= 1
      Wait = now() - self.ArrivalTime
      PtClass.Mon.observe(Wait)
      NursePool.Wakeup(G.NrsPl,'departure')

class ArrivalClass(Process):  # simulates patients arrivals
   ArvRate = None
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         yield hold,self,G.Rnd.expovariate(ArrivalClass.ArvRate)
         Pt = PtClass()
         activate(Pt,Pt.Run())

def ShowStatus(Evt):  # for debugging and code verification
   print
   print Evt, 'at time', now()
   print G.NrsPl.NrsCurrOnline, 'nurse(s) online'
   print PtClass.NPtsInSystem, 'patient(s) in system'
   if G.NrsPl.TB:
      print 'timebomb started at time', G.NrsPl.TB.TimeStarted
   else:  print 'no timebomb ticking'

def main():
   MOL = int(sys.argv[1])
   R = int(sys.argv[2])
   TO = float(sys.argv[3])
   initialize()
   G.NrsPl = NursePool(MOL,R,TO)
   activate(G.NrsPl,G.NrsPl.Run())
   ArrivalClass.ArvRate = float(sys.argv[4])
   PtClass.SrvRate = float(sys.argv[5])
   A = ArrivalClass()
   activate(A,A.Run())
   MaxSimTime = float(sys.argv[6])
   G.Debug = int(sys.argv[7])
   simulate(until=MaxSimTime)
   print 'mean wait =',PtClass.Mon.mean()
   print 'mean number of nurses online =',G.NrsPl.Mon.mean()

if __name__ == '__main__': main()
```

Since the number of servers varies through time, we cannot use the SimPy **Resource** class in a straightforward manner, as that class assumes a fixed number of servers. However, by making use of that class' priorities capability, we can achieve the effect of a varying number of servers. Here we make use of an idea from a page on the SimPy Web site, `http://simpy.sourceforge.net/changingcapacity.htm`.

The way this works is that we remove a server from availability by performing a **yield request** with a very high priority level, a level higher than is used for any real request. In our case here, a real request is done via the line

```
yield request,self,G.NrsPl.Rsrc,1
```

with priority 1. By contrast, in order to take one nurse off the shift, we perform

```
yield request,self,self.Rsrc,100
self.NrsCurrOnline -= 1
```

The high priority ensures that this bogus "request" will prevail over any real one, with the effect that the nurse is taken offline. Note, though, that existing services are not pre-empted, i.e. a nurse is not removed from the shift in the midst of serving someone.

Note the necessity of the line

```
self.NrsCurrOnline -= 1
```

The **n** member variable of SimPy's **Resource** class, which records the number of available resource units, would not tell us here how many nurses are available, because some of the resource units are held by the bogus "requests" in our scheme here. Thus we need a variable of our own, **NrsCurrOnline**.

As you can see from the call to **passivate()** in **NursePool.Run()**, the thread **NursePool.Run()** is mostly dormant, awakening only when it needs to add or delete a nurse from the pool. It is awakened for this purpose by the patient and "timebomb" classes, **PtClass** and **TimeBomb**, which call this function in **NursePool**:

```
def Wakeup(NrsPl,Evt):
   reactivate(NrsPl)
   NrsPl.WakingEvent = Evt
```

It wakes up the **NursePool** thread, which will then decide whether it should take action to change the size of the nurse pool, based on the argument **Evt**.

For example, when a new patient call arrives, generated by the **ArrivalClass** thread, the latter creates a **PtClass** thread, which simulates that one patient's progress through the system. The first thing this thread does is

```
NursePool.Wakeup(G.NrsPl,'arrival')
```

so as to give the **NursePool** thread a chance to check whether the pool should be expanded.

We also have a **TimeBomb** class, which deals with the fact that if the system is devoid of patients for a long time, the size of the nurse pool will be reduced. After the given timeout period, this thread awakenens the **NursePool** thread with the event 'timebomb exploded'.

By the way, since **activate()** requires that its first argument be a class instance rather than a class, we are forced to create an instance of **NursePool**, **G.NrsPl**, even though we only have one nurse pool. That leads to the situation we have with the function **NursePool.Wakeup()** being neither a class method nor an instance method.

Note the use of monitors, including in our **PeriodicSampler** class.

I have included a function **ShowStatus()** to help with debugging, and especially with verification of the program. Here is some sample output:

```
timebomb exploded at time 0.5
6 nurse(s) online
0 patient(s) in system
timebomb started at time 0

arrival at time 0.875581049552
5 nurse(s) online
1 patient(s) in system
timebomb started at time 0.5

service starts at time 0.875581049552
5 nurse(s) online
1 patient(s) in system
no timebomb ticking

departure at time 1.19578373243
5 nurse(s) online
0 patient(s) in system
no timebomb ticking

timebomb exploded at time 1.69578373243
5 nurse(s) online
0 patient(s) in system
timebomb started at time 1.19578373243

timebomb exploded at time 2.19578373243
4 nurse(s) online
0 patient(s) in system
timebomb started at time 1.69578373243

timebomb exploded at time 2.69578373243
3 nurse(s) online
0 patient(s) in system
timebomb started at time 2.19578373243

timebomb exploded at time 3.19578373243
2 nurse(s) online
0 patient(s) in system
timebomb started at time 2.69578373243

timebomb exploded at time 3.69578373243
1 nurse(s) online
0 patient(s) in system
timebomb started at time 3.19578373243
```