Introduction to Discrete-Event Simulation and the SimPy Language

Norm Matloff

November 7, 2006

Contents

1 What Is Discrete-Event Simulation?

Consider simulation of some system which evolves through time. There is a huge variety of such applications. One can simulate a weather system, for instance. A key point, though, is that in that setting, the events being simulated would be **continuous**, meaning for example that if we were to graph temperature against time, the curve would be continuous, no breaks.

By contrast, suppose we simulate the operation of a warehouse. Purchase orders come in and are filled, reduced inventory, but inventory is replenished from time to time. Here a typical variable would be the inventory itself, i.e. the number of items currently in stock for a given product. If we were to graph that number against time, we would get what mathematicians call a **step function**, i.e. a set of flat line segments with breaks between them. The events here—decreases and increases in the inventory—are discrete variables, not continuous ones.

2 World Views in Discrete-Event Simulation Programming

Simulation programming can often be difficult—difficult to write the code, and difficult to debug. The reason for this is that it really is a form of parallel programming, with many different activities in progress simultaneously, and parallel programming can be challenging.

For this reason, many people have tried to develop separate simulation **languages**, or at least simulation **paradigms** (i.e. programming styles) which enable to programmer to achieve clarity in simulation code. Special simulation languages have been invented in the past, notably SIMULA, which was invented in the 1960s and has significance today in that it was the language which invented the concept of object-oriented programmg which is so popular today. However, the trend today is to simply develop simulation *libraries* which can be called from ordinary languages such as C++, instead of inventing entire new languages.[1] So, the central focus today is on the programming paradigms, not on language. In this section we will present an overview of the three major discrete-event simulation paradigms.

[1]These libraries are often called "languages" anyway, and I will do so too.

2.1 The Activity-Oriented Paradigm

Let us think of simulating a queuing system. The time between arrivals of jobs, and the time needed to serve a job, will be continuous random variables, possibly having exponential or other continuous distributions.

Under the **activity-oriented paradigm**, we would break time into tiny increments. If for instance the mean interarrival time were, say 20 seconds, we might break time into increments of size 0.001. At each time point, our code would look around at all the activities, e.g. currently-active service jobs, and check for the occurrence of events, e.g. completion of service.

Let SimTime represent current simulated time. Our simulation code in the queue example above would look something like this:

```
QueueLength = 0
NJobsServed = 0
SumResidenceTimes = 0
ServerBusy = false
generate NextArrivalTime  // random # generation
NIncrements = MaxSimTime / 0.001
for SimTime = 1*0.001 to NIncrements*0.001 do
   if SimTime = NextArrivalTime then
      QueueLength++
      generate NextArrivalTime  // random # generation
      if not ServerBusy then
         ServerBusy = true
         jobobject.ArrivalTime = SimTime
         generate ServiceFinishedtime
         currentjob = jobobject
         add jobobject to queue
         QueueLength--
   else
      if SimTime = ServiceFinishedtime then
         NJobsServed++
         SumResidenceTimes += SimTime - currentjob.ArrivalTime
         if QueueLength > 0 then
            generate ServiceFinishedtime  // random # generation
            QueueLength--
         else
            ServerBusy = false
print out SumResidenceTimes / NJobsServed
```

2.2 The Event-Oriented Paradigm

Clearly, an activity-oriented simulation program is going to be very slow to execute. Most time increments will produce no change to the system at all, i.e. no new arrivals to the queue and no completions of service by the server. Thus the activity checks will be wasted processor time. This is a big issue, because in general simulation code often needs a very long time to run.

Inspection of the above pseudocode, though, shows a way to dramatically increase simulation speed. Instead of having time "creep along" so slowly, why not take a "shortcut" to the next event? What we could do is something like the following:

Instead of having the simulated time advance via the code

```
for SimTime = 1*0.001 to NIncrements*0.001 do
```

we could advance simulated time directly to the time of the next event:

```
if ServerBusy and NextArrivalTime < ServiceFinishedtime or
   not ServerBusy then
      SimTime = NextArrivalTime
else
   SimTime = ServiceFinishedtime
```

(The reason for checking ServerBusy is that ServiceFinishedtime will be undefined if ServerBusy is false.)

The entire pseudocode would then be

```
QueueLength = 0
NJobsServed = 0
SumResidenceTimes = 0
ServerBusy = false
generate NextArrivalTime
SimTime = 0.0;
while (1) do
   if ServerBusy and NextArrivalTime < ServiceFinishedtime or
      not ServerBusy then
         SimTime = NextArrivalTime
   else
      SimTime = ServiceFinishedtime
   if SimTime > MaxSimTime then break
   if SimTime = NextArrivalTime then
      QueueLength++
      generate NextArrivalTime
      if not ServerBusy then
         ServerBusy = true
         jobobject.ArrivalTime = SimTime
         currentjob = jobobject
         generate ServiceFinishedtime
         QueueLength--
   else  // the case SimTime = ServiceFinishedtime
      NJobsServed++
      SumResidenceTimes += SimTime - currentjob.ArrivalTime
      if QueueLength > 0 then
         generate ServiceFinishedtime
         QueueLength--
      else
         ServerBusy = false
print out SumResidenceTimes / NJobsServed
```

The **event-oriented** paradigm formalizes this idea. We store an **event set**, which is the set of all pending events. In our queue example above, for instance, there will always be at least one event pending, namely the next arrival, and sometimes a second pending event, namely the completion of a service. Our code above simply inspects the scheduled event times of all pending events (again, there will be either one or two of them in our example here), and updates SimTime to the minimum among them.

In the general case, there may be many events in the event set, but the principle is still the same—in each iteration of the **while** loop, we update SimTime to the minimum among the scheduled event times. Note also that in each iteration of the **while** loop, a new event is generated and added to the set; be sure to look at the pseudocode above and verify this.

Thus a major portion of the execution time for the program will consist of a find-minimum operation within the event set. Accordingly, it is desirable to choose a data structure for the set which will facilitate this operation, such as a heap-based **priority queue**. In many event-oriented packages, though, the event set is implemented simply as a linearly-linked list. This will be sufficiently efficient as long as there usually aren't too many events in the event set; again, in the queue example above, the maximum size of the event set is 2.

Again, note the contrast between this and continuous simulation models. The shortcut which is the heart of the event-oriented paradigm was only possible because of the discrete nature of system change. So this paradigm is not possible in models in which the states are continuous in nature.

The event-oriented paradigm was common in the earlier years of simulation, used in packages in which code in a general-purpose programming language such as C called functions in a simulation library. It still has some popularity today. Compared to the main alternative, the **process-oriented** paradigm, the chief virtues of the event-oriented approach are:

- Ease of implementation. The process-oriented approach requires something like **threads**, and in those early days there were no thread packages available. One needed to write one's own threads mechanisms, by writing highly platform-dependent assembly-language routines for stack manipulation.
- Execution speed. The threads machinery of process-oriented simulation really slows down execution speed (even if user-level threads are used).
- Flexibility. If for example one event will trigger two others, it is easy to write this into the application code.

2.3 The Process-Oriented Paradigm

Here each simulation activity is modeled by a **process**. The idea of a process is similar to the notion by the same name in Unix, and indeed one could write process-oriented simulations using Unix processes. However, these would be inconvenient to write, difficult to debug, and above all they would be slow.

As noted earlier, the old process-oriented software such as SIMULA and later CSIM were highly platform-dependent, due to the need for stack manipulation. However, these days this problem no longer exists, due to the fact that modern systems include threads packages (e.g. pthreads in Unix, Java threads, Windows threads and so on). Threads are sometimes called "lightweight" processes.

If we were to simulate a queuing system as above, but using the process-oriented paradigm, we would have two threads, one simulating the arrivals and the other simulating the operation of the server. Those would be the application-specific threads (so NumActiveAppThreads = 2 in the code below), and we would also have a general thread to manage the event set.

Our arrivals thread would look something like

```
NumActiveAppThreads++
while SimTime < MaxSimTime do
   generate NextArrivalTime
   add an arrival event for time NextArrivalTime to the event set
   sleep until wakened by the event-set manager
   jobobject.ArrivalTime = SimTime
   add jobobject to the machine queue
thread exit
```

The server thread would look something like

```
NumActiveAppThreads++
while SimTime < MaxSimTime do
   sleep until QueueLength > 0
   while QueueLength > 0 do
      remove queue head and assign to jobobject
      QueueLength--
      generate ServiceFinishedtime
      add a service-done event for time ServiceFinishedtime to the event set
      sleep until wakened by the event-set manager
      SumResidenceTimes += SimTime - jobobject.ArrivalTime
      NJobsServed++
thread exit
```

The event set manager thread would look something like

```
while SimTime < MaxSimTime do
   sleep until event set is nonempty
   delete the minimum-time event E from the event set
   update SimTime to the time scheduled for E
   wake whichever thread had added E to the event set
thread exit
```

The main() program would look something like this:

```
QueueLength = 0
NJobsServed = 0
SumResidenceTimes = 0
ServerBusy = false
start the 3 threads
sleep until all 3 threads exit
print out SumResidenceTimes / NJobsServed
```

Note that the event set manager would be library code, while the other modules shown above would be application code.

Two widely used process-oriented packages are C++SIM, available at `http://cxxsim.ncl.ac.uk` and SimPy, available at `http://simpy.sourceforge.net`.

The process-oriented paradigm produces more modular code. This is probably easier to write and easier for others to read. It is considered more elegant, and is the more popular of the two main world views today.

3 Introduction to the SimPy Simulation Language

SimPy (rhymes with "Blimpie" is a public-domain package for process-oriented discrete-event simulation. It is written in, and called from, Python. I like the clean manner in which it is designed, and the use of Python generators—and for that matter, Python itself—is a really strong point. If you haven't used Python before, you can learn enough about it to use SimPy quite quickly; see my quick tutorial on Python, at my Python tutorials page, `http://heather.cs.ucdavis.edu/~matloff/python.html`.

Instead of using threads, as is the case for most process-oriented simulation packages, SimPy makes novel use of Python's generators capability.[2] Generators allow the programmer to specify that a function can be prematurely exited and then later re-entered at the point of last exit, enabling **coroutines**, meaning functions that alternate execution with each other. The exit/re-entry points are marked by Python's **yield** keyword. Each new call to the function causes a resumption of execution of the function at the point immediately following the last yield executed in that function. As you will see below, that is exactly what we need for discrete-event simulation.

For convenience, I will refer to each coroutine (or, more accurately, each instance of a coroutine), as a **thread**.[3]

3.1 How to Obtain and Install SimPy

You will need to have Python version 2.3 or better.

Download SimPy from SimPy's Sourceforge site, `http://simpy.sourceforge.net`.

Create a directory, say **/usr/local/SimPy**.[4] You need to at least put the code files **Simulation.** and **__init__.** in that directory, and I will assume here that you also put in the test and documentation subdirectories which come with the package, say as subdirectories of **/usr/local/SimPy**.

You'll need that directory to be in your Python path, which is controlled by the PYTHONPATH environment variable. Set this in whatever manner your OS/shell sets environment variable. For example, in a **csh**/UNIX environment, type

```
setenv PYTHONPATH /usr/local/
```

Modify accordingly for **bash**, Windows, etc.

One way or the other, you need to be set up so that Python finds the library files correctly. Both the SimPy example programs and our example programs here include lines like

```
from SimPy.Simulation import *
```

[2] Python 2.2 or better is required. See my Python generators tutorial at the above URL if you wish to learn about generators, but you do not need to know about them to use SimPy.

[3] This tutorial does not assume the reader has a background in threads programming. In fact, readers who do have that background will have to unlearn some of what they did before, because our threads here will be non-preemptive, unlike the preemptive type one sees in most major threads packages.

[4] My instructions here will occasionally have a slight Unix orientation, but it should be clear how to make the small adjustments needed for other platforms.

which instructs the Python interpreter to look for the module Simulation in the package SimPy. Given the setting of PYTHONPATH above, Python would look in **/usr/local/** for a directory **SimPy**, i.e. look for a directory **/usr/local/SimPy**, and then look for **Simulation.py** and **__init__.py** (or their **.pyc** compiled versions) within that directory.

Test by copying **testSimPy** from that directory to some other directory and then running

```
python testSimPy.py
```

Some graphical windows will pop up, and after you remove them, a message like "Run 54 tests..." will appear.

3.2 SimPy Overview

Here are the major SimPy classes which we will cover in this introduction:[5]

- **Process**: simulates an entity which evolves in time, e.g. one job which needs to be served by a machine; we will refer to it as a thread, even though it is not a formal Python thread
- **Resource**: simulates something to be queued for, e.g. the machine

Here are the major SimPy operations/function calls we will cover in this introduction:

- **activate()**: used to mark a thread as runnable when it is first created
- **simulate()**: starts the simulation
- **yield hold**: used to indicate the passage of a certain amount of time within a thread; **yield** is a Python operator whose first operand is a function to be called, in this case a code for a function which performs the hold operation in the SimPy library
- **yield request**: used to cause a thread to join a queue for a given resource (and start using it immediately if no other jobs are waiting for the resource)
- **yield release**: used to indicate that the thread is done using the given resource, thus enabling the next thread in the queue, if any, to use the resource
- **yield passivate**: used to have a thread wait until "awakened" by some other thread
- **reactivate()**: does the "awakening" of a previously-passivated thread
- **cancel()**: cancels all the events associated with a previously-passivated thread

Here is how the flow of control goes from one function to another:

- When the main program calls **simulate()** the main program blocks. The simulation itself then begins, and the main program will not run again until the simulation ends.

[5]Others will be covered in our followup tutorial at `AdvancedSimpy.pdf`.

- Anytime a thread executes yield, that thread will pause. SimPy's internal functions will then run, and will restart some thread (possibly the same thread).
- When a thread is finally restarted, its execution will resume right after whichever yield statement was executed last in this thread.

Note that **activate()**, **reactivate()** and **cancel** do NOT result in a pause to the calling function. Such a pause occurs only when **yield** is invoked. Those with extensive experience in threads programming (which, as mentioned, we do NOT assume here) will recognize this the **non-preemptive** approach to threads. In my opinion, this is a huge advantage, for two reasons:

- Your code is not cluttered up with a lot of lock/unlock operations.
- Execution is deterministic, which makes both writing and debugging the program much easier.

(A disadvantage is that SimPy, in fact Python in general, cannot run in a parallel manner on multiprocessor machines.)

3.3 Introduction to Using SimPy

We will demonstrate the usage of SimPy by presenting three variations on a machine-repair model. In each case, we are modeling a system consisting of two machines which are subject to breakdown, but with different repair patterns:

- **MachRep1.py**: There are two repairpersons, so that both machines can be repaired simultaneously if they are both down at once.
- **MachRep2.py**: Here there is only one repairperson, so if both machines are down then one machine must queue for the repairperson while the other machine is being repaired.
- **MachRep3.py**: Here there is only one repairperson, and he/she is not summoned until both machines are down.

In all cases, the up times and repair times are assumed to be exponentially distributed with means 1.0 and 0.5, respectively. Now, let's look at the three programs.[6]

3.3.1 MachRep1.py: Our First SimPy Program

Here is the code:

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

# MachRep1.py

# Introductory SimPy example:  Two machines, which sometimes break down.
```

[6]You can make your own copies of these programs by downloading the raw **.tex** file for this tutorial, and then editing out the material other than the program you want.

```
# Up time is exponentially distributed with mean 1.0, and repair time is
# exponentially distributed with mean 0.5.  There are two repairpersons,
# so the two machines can be repaired simultaneously if they are down
# at the same time.

# Output is long-run proportion of up time.  Should get value of about
# 0.66.

import SimPy.Simulation
import random

class G:  # global variables
   Rnd = random.Random(12345)

class MachineClass(SimPy.Simulation.Process):
   UpRate = 1/1.0  # reciprocal of mean up time
   RepairRate = 1/0.5  # reciprocal of mean repair time
   TotalUpTime = 0.0  # total up time for all machines
   NextID = 0  # next available ID number for MachineClass objects
   def __init__(self):
      SimPy.Simulation.Process.__init__(self)  # required
      self.UpTime = 0.0  # amount of work this machine has done
      self.StartUpTime = 0.0  # time the current up period started
      self.ID = MachineClass.NextID   # ID for this MachineClass object
      MachineClass.NextID += 1
   def Run(self):
      while 1:
         # record current time, now(), so can see how long machine is up
         self.StartUpTime = SimPy.Simulation.now()
         # hold for exponentially distributed up time
         UpTime = G.Rnd.expovariate(MachineClass.UpRate)
         yield SimPy.Simulation.hold,self,UpTime
         # update up time total
         MachineClass.TotalUpTime += SimPy.Simulation.now() - self.StartUpTime
         RepairTime = G.Rnd.expovariate(MachineClass.RepairRate)
         # hold for exponentially distributed repair time
         yield SimPy.Simulation.hold,self,RepairTime

def main():
   SimPy.Simulation.initialize()  # required
   # set up the two machine processes
   for I in range(2):
      # create a MachineClass object
      M = MachineClass()
      # register thread M, executing M's Run() method,
      SimPy.Simulation.activate(M,M.Run())
   # run until simulated time 10000
   MaxSimtime = 10000.0
   SimPy.Simulation.simulate(until=MaxSimtime)
   print "the percentage of up time was", \
      MachineClass.TotalUpTime/(2*MaxSimtime)

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

First, some style issues:

- My style is to put all global variables into a Python class, which I usually call **G**. See my Python tutorial if you wish to know my reasons.
- In order to be able to use debugging tools, I always define a function **main()** which is my "main" program, and include the line

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

Again, see my Python tutorial if you wish to know the reasons.

- In this first SimPy example, I am using the "wordier" form of Python's **import** facility:

  ```
  import SimPy.Simulation
  ```

  This leads to rather cluttered code, such as

  ```
  SimPy.Simulation.simulate(until=MaxSimtime)
  ```

  instead of

  ```
  simulate(until=MaxSimtime)
  ```

  The latter could be used had we done the **import** via

  ```
  from SimPy.Simulation import *
  ```

  But in this first SimPy program, I wanted to clearly distinguish SimPy's functions from the others. The same holds for the functions in the Python library **random**. So, in this program, we use long names.

Let's look at **main()**. Since we are simulating two machines, we create two objects of our **MachineClass** class. These will be the basis for our two machine threads. Here **MachineClass** is a class which I wrote, as a subclass of SimPy's built-in class **Process**.

By calling SimPy's **activate()** function on the two instances of **MachineClass**, we tell SimPy to create a thread for each of them, which will execute the **Run()** function for their class. This puts them on SimPy's internal "ready" list of threads which are ready to run.

The call to SimPy's **simulate()** function starts the simulation. The next statement, the print, won't execute for quite a while, since it won't be reached until the call to **simulate()** returns, and that won't occur until the end of the simulation.

Python allows **named arguments** in function calls,[7], and this feature is used often in the SimPy library. For example, SimPy's **simulate()** function has many arguments, one of which is named **until**.[8] In our call here, we have only specified the value of **until**, omitting the values of the other arguments. That tells the Python interpreter that we accept whatever default values the other arguments have, but we want the argument **until** to have the value 10000.0. That argument has the meaning that we will run the simulation for a simulated time span of duration 10000.0.

In general, I'll refer to the functions like **MachineClass.Run()** in this example) as the **process execution method** (PEM). (Functions in Python are called *methods*.)

The object **G.Rnd** is an instance of the **Random** class in the **random** module of the Python library. This will allow us to generate random numbers, the heart of the simulation. We have arbitrarily initialized the seed to 12345.

[7] See my Python tutorial.

[8] Look in the file **Simulation.py** of the SimPy library to see the entire code for **simulate()**.

Since we are assuming up times and repair times are exponentially distributed, our code calls the function **random.Random.expovariate()**. Its argument is the reciprocal of the mean.[9] Here we have taken the mean up time and repair times to be 1.0 and 0.5, respectively, just as an example.

Note too that Python's **random** class contains a variety of random number generators. To see what is available, get into interactive mode in Python and type

```
>>> import random
>>> dir(random)
```

To find out what the functions do, use Python's online help facility, e.g.

```
>>> help(random.expovariate)
```

The call to SimPy's **initialize()** function is required for all SimPy programs.

Now, let's look at **MachineClass**. First we define two class variables,[10] **TotalUpTime** and **NextID**. As the comment shows, **TotalUpTime** will be used to find the total up time for all machines, so that we can eventually find out what proportion of the time the machines are up. Be sure to make certain you understand why **TotalUpTime** must be a class variable rather than an instance variable.

Next, there is the class' constructor function, **__init__()**.[11] Since our class here, **MachineClass**, is a subclass of the SimPy built-in class **Process**, the first thing we must do is call the latter's constructor; our program will not work if we forget this (it will also fail if we forget the argument **self** in either constructor).

Finally, we set several of the class' instance variables, explained in the comments. Note in particular the ID variable. You should always put in some kind of variable like this, not necessarily because it is used in the simulation code itself, but rather as a debugging aid.

If you have experience with pre-emptive thread systems, note that we did NOT need to protect the line

```
MachineClass.NextID += 1
```

with a lock variable. This is because a SimPy thread retains control until voluntarily relinquishing it via a **yield**. Our thread here will NOT be interrupted in the midst of incrementing **MachineClass.NextID**.

Now let's look at the details of **Machine.Run()**, where the main action of the simulation takes place.

The SimPy function **now()** yields the current simulated time. We are starting this machine in up mode, i.e. no failure has occurred yet. Remember, we want to record how much of the time each machine is up, so we need to have a variable which shows when the current up period for this machine began. With this in mind, we had our code **self.StartUpTime = SimPy.Simulation.now()** record the current time, so that later the code

```
MachineClass.TotalUpTime += SimPy.Simulation.now() - self.StartUpTime
```

[9] You might think the mean would be a more natural form for the argument, but the reciprocal has physical meaning too, which we will discuss later in our unit which reviews the laws of probability for continuous random variables.

[10] If you are not familiar with the general object-oriented programming terms **class variable** and **instance variable**, see my Python tutorial.

[11] Some programmers consider this to be a bit different from a constructor function, but I'll use that term here.

will calculate the duration of this latest uptime period, and add it to our running total.

Again, make sure you understand why **StartUpTime** needs to be an instance variable rather than a class variable.

A point to always remember about simulation programming is that you must constantly go back and forth between two mental views of things. On the one hand, there is what I call the "virtual reality" view, where you are imagining what would happen in the real system you are simulating. On the other hand, there is the "nuts and bolts programming" view, in which you are focused on what actual program statesments do. With these two views in mind, let's discuss the lines

```
UpTime = G.Rnd.expovariate(MachineClass.UpRate)
yield SimPy.Simulation.hold,self,UpTime
```

First, from a "virtual reality" point of view, what the **yield** does is simulate the passage of time, specifically, **UpTime** amount of time, while the machine goes through an up period, at the end of which a breakdown occurs.

Now here's the "nuts and bolts programming" point of view: Python's **yield** construct is a like a **return**, as it does mean an exit from the function and the passing of a return value to the caller. In this case, that return value is the tuple (SimPy.Simulation.hold,self,UpTime). Note by the way that the first element in that tuple is in SimPy cases always the name of a function in the SimPy library. The difference between **yield** and **return** is that the "exit" from the function is only temporary. The SimPy internals will later call this function again, and instead of starting at the beginning, it will "pick up where it left off." In other words, the statement

```
yield SimPy.Simulation.hold,self,UpTime
```

will cause a temporary exit from the function but later we will come back and resume execution at the line

```
MachineClass.TotalUpTime += SimPy.Simulation.now() - self.StartUpTime
```

The term "yield" alludes to the fact that this thread physically relinquishes control of the Python interpreter. Execution of this thread will be suspended, and another thread will be run. Later, after simulated time has advanced to the end of the up period, control will return to this thread, resuming exactly where the suspension occurred.

The second yield,

```
RepairTime = G.Rnd.expovariate(MachineClass.RepairRate)
yield SimPy.Simulation.hold,self,RepairTime
```

works similarly, suspending execution of the thread for a simulated exponentially-distributed amount of time to simulate the repair time.

In other words, the **while** loop within **MachineClass.Run()** simulates a repeated cycle of up time, down time, up time, down time, ... for this machine.

It is very important to understand how control transfers back and forth among the threads. Say for example that machine 0's first uptime lasts 1.2 and its first downtime lasts 0.9, while for machine 1 the corresponding times are 0.6 and 0.8. The simulation of course starts at time 0.0. Then here is what will happen:

- The two invocations of **activate()** in **main()** cause the two threads to be added to the “runnable” list maintained by the SimPy internals.
- The invocation of **simulate()** tells SimPy to start the simulation. It will then pick a thread from the “runnable” list and run it. We cannot predict which one it will be, but let’s say it’s the thread for machine 0.
- The thread for machine 0 will generate the value 1.2, then yield. SimPy’s internal event list will now show that the thread for machine 0 is suspended until simulated time 0.0+1.2 = 1.2. This thread will be moved to SimPy’s “suspended” list.
- The thread for machine 1 (the only available choice at this time) will now run, generating the value 0.6, then yielding. SimPy’s event list will now show that the thread for machine 0 is waiting until time 0.6. The “runnable” list will be empty now.
- SimPy advances the simulated time clock to the earliest event in the event list, which is for time 0.6. It removes this event from the event list, and then resumes the thread corresponding to the 0.6 event, i.e. the thread for machine 1.
- The latter generates the value 0.8, then yields. SimPy’s event list will now show that the thread for machine 0 is waiting until time 0.6+0.8 = 1.4.
- SimPy advances the simulated time clock to the earliest event in the event list, which is for time 1.2. It removes this event from the event list, and then resumes the thread corresponding to the 1.2 event, i.e. the thread for machine 0.
- Etc.

When the simulation ends, control returns to the line following the call to **simulate()** where the result is printed out:

```
print "the percentage of up time was", Machine.TotalUpTime/(2*MaxSimtime)
```

3.3.2 MachRep2.py: Introducing the Resource Class

Here is the code:

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

# MachRep2.py

# SimPy example:  Variation of MachRep1.py.  Two machines, but sometimes
# break down.  Up time is exponentially distributed with mean 1.0, and
# repair time is exponentially distributed with mean 0.5.  In this
# example, there is only one repairperson, so the two machines cannot be
# repaired simultaneously if they are down at the same time.

# In addition to finding the long-run proportion of up time as in
# Mach1.py, let’s also find the long-run proportion of the time that a
# given machine does not have immediate access to the repairperson when
# the machine breaks down.  Output values should be about 0.6 and 0.67.

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

```

class G:  # globals
   Rnd = Random(12345)
   # create the repairperson
   RepairPerson = Resource(1)

class MachineClass(Process):
   TotalUpTime = 0.0  # total up time for all machines
   NRep = 0  # number of times the machines have broken down
   NImmedRep = 0  # number of breakdowns in which the machine
                  # started repair service right away
   UpRate = 1/1.0  # breakdown rate
   RepairRate = 1/0.5  # repair rate
   # the following two variables are not actually used, but are useful
   # for debugging purposes
   NextID = 0  # next available ID number for MachineClass objects
   NUp = 0  # number of machines currently up
   def __init__(self):
      Process.__init__(self)
      self.StartUpTime = 0.0  # time the current up period stated
      self.ID = MachineClass.NextID   # ID for this MachineClass object
      MachineClass.NextID += 1
      MachineClass.NUp += 1  # machines start in the up mode
   def Run(self):
      while 1:
         self.StartUpTime = now()
         yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
         MachineClass.TotalUpTime += now() - self.StartUpTime
         # update number of breakdowns
         MachineClass.NRep += 1
         # check whether we get repair service immediately
         if G.RepairPerson.n == 1:
            MachineClass.NImmedRep += 1
         # need to request, and possibly queue for, the repairperson
         yield request,self,G.RepairPerson
         # OK, we've obtained access to the repairperson; now
         # hold for repair time
         yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
         # release the repairperson
         yield release,self,G.RepairPerson

def main():
   initialize()
   # set up the two machine processes
   for I in range(2):
      M = MachineClass()
      activate(M,M.Run())
   MaxSimtime = 10000.0
   simulate(until=MaxSimtime)
   print 'proportion of up time:', MachineClass.TotalUpTime/(2*MaxSimtime)
   print 'proportion of times repair was immediate:', \
      float(MachineClass.NImmedRep)/MachineClass.NRep

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

This model includes a queuing element. A typical (but not universal) way to handle that in SimPy is to add an object of the SimPy class **Resource**:

```
RepairPerson = Resource(1)
```

with the “1” meaning that there is just one repairperson. Then in **MachineClass.Run()** we do the following when an uptime period ends:

```
yield request,self,G.RepairPerson
yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
yield release,self,G.RepairPerson
```

Here is what those **yield** lines do:

- The first **yield** requests access to the repairperson. This will return immediately if the repairperson is not busy now. Otherwise, this thread will be suspended until the repairperson is free, at which time the thread will be resumed.
- The second **yield** simulates the passage of time, representing the repair time.
- The third **yield** releases the repairperson. If another machine had been in the queue, awaiting repair—with its thread suspended, having executing the first **yield**—it would now attain access to the repairperson, and its thread would now execute the second **yield**.

Suppose for instance the thread simulating machine 1 reaches the first **yield** slightly before the thread for machine 0 does. Then the thread for machine 1 will immediately go to the second **yield**, while the thread for machine 0 will be suspended at the first **yield**. When the thread for machine 1 finally executes the third **yield**, then SimPy's internal code will notice that the thread for machine 0 had been queued, waiting for the repairperson, and would now reactivate that thread.

Note the line

```
if G.RepairPerson.n == 1:
```

Here **n** is a member variable in SimPy's class **Resource**. It gives us the number of items in the resource currently free. In our case here, it enables us to keep a count of how many breakdowns are lucky enough to get immediate access to the repairperson. We later use that count in our output.

The same class contains the member variable **waitQ**, which is a Python list which contains the queue for the resource. This may be useful in debugging, or if you need to implement a special priority discipline other than the ones offered by SimPy.

Another member variable is **activeQ**, which is a list of threads which are currently using units of this resource.

3.3.3 MachRep3.py: Introducing Passivate/Reactivate Operations

Here's the code:

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

# MachRep3.py

# SimPy example:  Variation of Mach1.py, Mach2.py.  Two machines, but
# sometimes break down.  Up time is exponentially distributed with mean
# 1.0, and repair time is exponentially distributed with mean 0.5.  In
# this example,there is only one repairperson, and she is not summoned
# until both machines are down.  We find the proportion of up time.  It
# should come out to about 0.45.
```

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

class G:  # globals
   Rnd = Random(12345)
   RepairPerson = Resource(1)

class MachineClass(Process):
   MachineList = []  # list of all objects of this class
   UpRate = 1/1.0
   RepairRate = 1/0.5
   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
   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):
      while 1:
         self.StartUpTime = now()
         yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
         MachineClass.TotalUpTime += now() - self.StartUpTime
         # update number of up machines
         MachineClass.NUp -= 1
         # if only one machine down, then wait for the other to go down
         if MachineClass.NUp == 1:
            yield passivate,self
         # here is the case in which we are the second machine down;
         # either (a) the other machine was waiting for this machine to
         # go down, or (b) the other machine is in the process of being
         # repaired
         elif G.RepairPerson.n == 1:
            reactivate(MachineClass.MachineList[1-self.ID])
         # now go to repair
         yield request,self,G.RepairPerson
         yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
         MachineClass.NUp += 1
         yield release,self,G.RepairPerson

def main():
   initialize()
   for I in range(2):
      M = MachineClass()
      activate(M,M.Run())
   MaxSimtime = 10000.0
   simulate(until=MaxSimtime)
   print 'proportion of up time was', MachineClass.TotalUpTime/(2*MaxSimtime)

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

Recall that in this model, the repairperson is not summoned until both machines are down. We add a class variable **MachineClass.NUp** which we use to record the number of machines currently up, and then use it in the following code, which is executed when an uptime period for a machine ends:

```
if MachineClass.NUp == 1:
   yield passivate,self
elif G.RepairPerson.n == 1:
   reactivate(MachineClass.MachineList[1-self.ID])
```

We first update the number of up machines, by decrementing **MachineClass.NUp**. Then if we find that there is still one other machine remaining up, this thread must suspend, to simulate the fact that this broken machine must wait until the other machine goes down before the repairperson is summoned. The way this suspension is implemented is to invoke **yield** with the operand **passivate**. Later the other machine's thread will execute the **reactivate()** statement on this thread, "waking" it.

But there is a subtlety here. Suppose the following sequence of events occur:

- machine 1 goes down
- machine 0 goes down
- the repairperson arrives
- machine 0 starts repair[12]
- machine 0 finishes repair
- machine 1 starts repair
- machine 0 goes down again

The point is that when the thread for machine 0 now executes

```
if MachineClass.NUp == 1:
```

the answer will be no, since **MachineClass.NUp** will be 0. Thus this machine should not passivate itself. But it is not a situation in which this thread should waken the other one either. Hence the need for the **elif** condition.

3.3.4 MMk.py

Here is an alternate way to handle queues, by writing one's own code to manage them. Though for most situations in which entities queue for a resource we make use of the SimPy's **Resource** class, there are some situations in which we want finer control. For instance, we may wish to set up a special priority scheme, or we may be modeling a system in which the number of resources varies with time.[13]

We thus need to be able to handle resource management "on our own," without making use of the **Resource** class. The following program shows how we can do this, via **passivate()** and **reactivate()**:

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

# simulates NMachines machines, plus a queue of jobs waiting to use them

# usage:  python MMk.py NMachines ArvRate SrvRate MaxSimtime

from SimPy.Simulation import *
```

[12]You might argue that machine 1 should be served first, but we put nothing in our code to prioritize the order of service.

[13]One way to do this with **Resource** is to use fake **yield request** and **yield release** statements, with the effect of reducing and increasing the number of servers. However, this must be done carefully. See a discussion of this on the SimPy Web site, at `http://simpy.sourceforge.net/changingcapacity.htm`.

```
from random import Random,expovariate

# globals
class G:
   Rnd = Random(12345)

class MachineClass(Process):
   SrvRate = None  # reciprocal of mean service time
   Busy = []  # busy machines
   Idle = []  # idle machines
   Queue = []  # queue for the machines
   NDone = 0  # number of jobs done so far
   TotWait = 0.0  # total wait time of all jobs done so far, including
                  # both queuing and service times
   def __init__(self):
      Process.__init__(self)
      MachineClass.Idle.append(self)  # starts idle
   def Run(self):
      while 1:
         # "sleep" until this machine awakened
         yield passivate,self
         MachineClass.Idle.remove(self)
         MachineClass.Busy.append(self)
         # take jobs from the queue as long as there are some there
         while MachineClass.Queue != []:
            # get the job
            J = MachineClass.Queue.pop(0)
            # do the work
            yield hold,self,G.Rnd.expovariate(MachineClass.SrvRate)
            # bookkeeping
            MachineClass.NDone += 1
            MachineClass.TotWait += now() - J.ArrivalTime
         MachineClass.Busy.remove(self)
         MachineClass.Idle.append(self)

class JobClass:
   def __init__(self):
      self.ArrivalTime = now()

class ArrivalClass(Process):
   ArvRate = None
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         # wait for arrival of next job
         yield hold,self,G.Rnd.expovariate(ArrivalClass.ArvRate)
         J = JobClass()
         MachineClass.Queue.append(J)
         # any machine ready?
         if MachineClass.Idle != []:
            reactivate(MachineClass.Idle[0])

def main():
   NMachines = int(sys.argv[1])
   ArrivalClass.ArvRate = float(sys.argv[2])
   MachineClass.SrvRate = float(sys.argv[3])
   initialize()
   for I in range(NMachines):
      M = MachineClass()
      activate(M,M.Run())
   A = ArrivalClass()
   activate(A,A.Run())
   MaxSimtime = float(sys.argv[4])
   simulate(until=MaxSimtime)
   print MachineClass.TotWait/MachineClass.NDone

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

3.3.5 SMP.py

Here is another example, this one modeling a **multiprocessor** computer system, i.e. one with many CPUs.

Here we see more use of SimPy's request and release capabilities. One thing to pay particular attention to is the fact that a processor needs at one point to have possession (**yield request**) of two things at once.

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

# SMP.py

# SimPy example:  Symmetric multiprocessor system.  Have m processors
# and m memory modules on a single shared bus.  The processors read from
# and write to the memory modules via messages sent along this shared
# bus.  The key word here is "shared"; only one entity (processor or
# memory module) can transmit information on the bus at one time.

# When a processor generates a memory request, it must first queue for
# possession of the bus.  Then it takes 1.0 amount of time to reach the
# proper memory module.  The request is queued at the memory module, and
# when finally served, the service takes 0.6 time.  The memory module
# must then queue for the bus.  When it acquires the bus, it sends the
# response (value to be read in the case of a read request,
# acknowledgement in the case of a write) along the bus, together with
# the processor number.  The processor which originally made the request
# has been watching the bus, and thus is able to pick up the response.

# When a memory module finishes a read or write operation, it will not
# start any other operations until it finishes sending the result of the
# operation along the bus.

# For any given processor, the time between the completion of a previous
# memory request and the generation of a new request has an exponential
# distribution.  The specific memory module requested is assumed to be
# chosen at random (i.e. uniform distribution) from the m modules.
# While a processor has a request pending, it does not generate any new
# ones.

# The processors are assumed to act independently of each other, and the
# requests for a given processor are assumed independent through time.
# Of course, more complex assumptions could be modeled.

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

class Processor(Process):
   M = int(sys.argv[1])  # number of CPUs/memory modules
   InterMemReqRate = 1.0/float(sys.argv[2])
   NDone = 0  # number of memory requests completed so far
   TotWait = 0.0  # total wait for those requests
   WaitMem = 0
   NextID = 0
   def __init__(self):
      Process.__init__(self)
      self.ID = Processor.NextID
      Processor.NextID += 1
   def Run(self):
      while 1:
         # generate a memory request
         yield hold,self,expovariate(Processor.InterMemReqRate)
         self.StartWait = now()  # start of wait for mem request
         # acquire bus
         yield request,self,G.Bus
         # use bus
         yield hold,self,1.0
```

```
            # relinquish bus
            yield release,self,G.Bus
            self.Module = G.Rnd.randrange(0,Processor.M)
            # go to memory
            self.StartMemQ = now()
            yield request,self,G.Mem[self.Module]
            if now() > self.StartMemQ:
               Processor.WaitMem += 1
            # simulate memory operation
            yield hold,self,0.6
            # memory sends result back to requesting CPU
            yield request,self,G.Bus
            yield hold,self,1.0
            # done
            yield release,self,G.Bus
            yield release,self,G.Mem[self.Module]
            Processor.NDone += 1
            Processor.TotWait += now() - self.StartWait

# globals
class G:
   Rnd = Random(12345)
   Bus = Resource(1)
   CPU = []  # array of processors
   Mem = []  # array of memory modules

def main():
   initialize()
   for I in range(Processor.M):
      G.CPU.append(Processor())
      activate(G.CPU[I],G.CPU[I].Run())
      G.Mem.append(Resource(1))
   MaxSimtime = 10000.0
   simulate(until=MaxSimtime)
   print 'mean residence time', Processor.TotWait/Processor.NDone
   print 'prop. wait for mem', float(Processor.WaitMem)/Processor.NDone

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

3.3.6 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.

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 sets up a timeout by creating an object **G.TO** of our **TimeOut** class, and sets up a transmission and acknowledgement by creating an object **G.ACK** of our **Acknowledge** class. Then the node passivates itself, allowing **G.TO** and **G.ACK** to do their work. One of them will finish first, and then 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:[14] 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.

You should notice some differences about this example from the machine-repair models we looked at earlier:

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

- Rather than creating and activating all the threads before the simulation starts, here most of our threads are created "on the fly," as the simulation progresses.
- The functions **Acknowledge.Run()** and **TimeOut.Run()** don't consist of **while** loops. Each thread does one thing, and then exits.[15]

This is a common pattern.

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, 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):
      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):
```

[15]Or is canceled.

```
      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**. Each object of this type models one job to be done. 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, simulate the repair of the machine and then create a new instance of **Job** which will simulate the processing of the remainder of the interrupted job.

There are other ways of doing this, in particular by using SimPy's **interrupt()** and **interrupted()** functions, but again, we defer this to a separate document in `http://heather.cs.ucdavis.edu/~matloff/156/PLN`.

3.3.7 Note These Restrictions

Some PEMs may be rather lengthy, and thus you will probably want to apply top-down program design and break up one monolithic PEM into smaller functions. In other words, you may name your PEM **Run()**, and then have **Run()** in turn call some smaller functions. This is of course highly encouraged. However, you must make sure that you do not invoke **yield** in those subprograms; it must be used only in the PEM itself. Otherwise the Python interpreter would lose track of where to return the next time the PEM were to resume execution.

Also, make sure NOT to invoke **yield** from within **main()** or some other function not associated with a call to **activate()**.

3.3.8 Other SimPy Features

Advanced features of SimPy will be discussed in separate documents in `http://heather.cs.ucdavis.edu/~matloff/156/PLN`.

3.4 SimPy Data Collection and Display

SimPy provides the class **Monitor** to make it more convenient to collect data for your simulation output. It is a subclass of the Python list type.

3.4.1 Introduction to Monitors

For example, suppose you have a variable **X** in some line in your SimPy code and you wish to record all values **X** takes on during the simulation. Then you would set up an object of type **Monitor**, say named **XMon**, in order to remind yourself that this is a monitor for **X**. Each time you have a value of **X** to record, you would have a line like

```
XMon.observe(X)
```

which would add the value, and the current simulated time, to the list in **XMon**. (So, **XMon**'s main data item is a list of pairs.)

The **Monitor** class also includes member functions that operate on the list. For example, you can compute the mean of **X**:

```
print 'the mean of X was', XMon.mean()
```

For example, we could apply this to the program **MMk.py** in Section 3.3.4. Here are code excerpts where we would make changes (look for lines referring to **WaitMon**):

```
class MachineClass(Process):
   ...
   TotWait = 0.0
   WaitMon = Monitor()
```

```
def __init__(self):
   ...
def Run(self):
   while 1:
   ...
      while MachineClass.Queue != []:
         J = MachineClass.Queue.pop(0)
         yield hold,self,G.Rnd.expovariate(MachineClass.SrvRate)
         Wait = now() - J.ArrivalTime
         MachineClass.WaitMon.observe(Wait)
...
MaxSimtime = float(sys.argv[4])
simulate(until=MaxSimtime)
print MachineClass.WaitMon.mean()
```

There is a function **Monitor.var()** for the variance too.

Note, though, that means are often not meaningful, no pun intended. To get a better understanding of queue wait times, for instance, you may wish to plot a histogram of the wait times, rather than just computing their mean. This is possible, via the function **Monitor.histogram**, which finds the bin counts and places them into a data structure which can then be displayed using SimPy's SimPlot package.

Indeed, since monitors collect all the data, you can write your own routines (or better, subclasses of **Monitor**, to find quantiles, etc.

3.4.2 Time Averages

Suppose in the example above we wished to find the long-run queue length. Before addressing how to do this, let's first ask what it really means.

Suppose we record every queue length that occurs in our simulation run, and take the average of those numbers. Would that be what we want? No, because it doesn't account for the time duration of each of those numbers. If for instance the queue had length 5 for long periods of time but had length 2 for shorter times, clearly we should not give the 5 and the 2 equal weights. We need to factor the durations into our weighting.

Say for instance the queue lengths were as follows: 2 between times 0.0 and 1.4, 3 between times 1.4 and 2.1, 2 between times 2.1 and 4.9, and 1 between 4.9 and 5.3. Then the average would be

$$(2 \times 1.4 + 3 \times 0.7 + 2 \times 2.8 + 1 \times 0.4)/5.3 = 2.06 \tag{1}$$

Another way to look at it would be to think of observing the system at regular time intervals, say 1.0, 2.0, 3.0 etc. Let Q_i denote the queue length observed at time i. Then we could define the long-run average queue length as

$$\lim_{n \to \infty} \frac{Q_1 + ... + Q_n}{n} \tag{2}$$

This actually is consistent with (1), in the long run.

3.4.3 The Function Monitor.timeAverage()

The function **Monitor.timeAverage()** computes time-value product averages for us, very convenient. Each time the queue changes length, you would call **Monitor.observe()** with the current queue length as argument, resulting in **Monitor** recording the length and the current simulated time (from **now()**).

In our little numerical example which led to (1), when the simulation ends, at time 5.3, the monitor will consist of this list of pairs: [[0.0,2], [1.4,3], [2.1,2], [4.9,1]] The function **timeAverage()** would then compute the value 2.06, as desired.

3.4.4 But I Recommend That You Not Use This Function

You should be careful, though. Properly keeping track of when to call **timeAverage()** is a bit delicate. Also, this function only gives you a mean, not variances or other statistics.

Thus I recommend that you simply set up another thread whose sole purpose is to add periodic sampling to estimate (2). This is simpler, more general and more flexible. To that end, here is a function you can use:

```
# PeriodicSampler.py

# creates a thread for periodic sampling, e.g. to be used for long-run
# queue length; the arguments Per, Mon and Fun are the sampling period,
# the monitor to be used, and the function to be called to get the data
# to be recorded

from SimPy.Simulation import *

class PerSmp(Process):
   def __init__(self,Per,Mon,Fun):
      Process.__init__(self)
      self.Per = Per
      self.Mon = Mon
      self.Fun = Fun
   def Run(self):
      while 1:
         yield hold,self,self.Per
         Data = self.Fun()
         self.Mon.observe(Data)
```

Here the argument **Per** allows us to sample with whatever frequency we like. A higher rate gives us more statistical accuracy (due to taking more samples), while a lower rate means a somewhat faster program.

Note the need for the function argument **Fun**. We need to tell **PerSmp** what data item to record. If we had made the argument that data, then we'd only get the first value of that data (probably 0 or None), rather than the changing values over time.

Here is an example of use:

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

# PerSmpExample.py--illustration of usage of the PerSmp class

# single-server queue, with interarrival and service times having
# uniform distributions on (0,1) and (0,0.5), respectively

from SimPy.Simulation import *
```

```
from random import Random,uniform
import sys
from PeriodicSampler import PerSmp

class G:  # globals
   Rnd = Random(12345)
   S = None  # our one disk

class Srvr(Resource):
   def __init__(self):
      Resource.__init__(self)
      self.QMon = Monitor()  # monitor queue lengths
      self.PrSm = PerSmp(1.0,self.QMon,self.SMonFun)
      activate(self.PrSm,self.PrSm.Run())
   def SMonFun(self):  # for PerSmp
      return len(self.waitQ)

class Job(Process):
   def __init__(self):
      Process.__init__(self)
      self.ArrivalTime = now()
   def Run(self):
      yield request,self,G.S
      yield hold,self,G.Rnd.uniform(0,0.5)
      yield release,self,G.S

class Arrivals(Process):
   def __init__(self):
      Process.__init__(self)
   def Run(self):
      while 1:
         yield hold,self,G.Rnd.uniform(0,1)
         J = Job()
         activate(J,J.Run())

def main():
   initialize()
   A = Arrivals()
   activate(A,A.Run())
   G.S = Srvr()
   MaxSimtime = 10000.0
   simulate(until=MaxSimtime)
   print 'mean queue length:',G.S.QMon.mean()

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

3.4.5 Little's Rule

Little's Rule says,

$$\text{mean queue length} = \text{arrival rate} \times \text{mean wait}$$

For First Come, First Served queues, an informal proof goes along the following lines: Imagine that you have just gotten to the head of the queue and have started service, with a wait of 5 minutes, and that the arrival rate is 2 jobs per minute. During your 5-minute wait, there would be an average of $5 \times 2 = 10$ jobs arriving, thus an average of 10 jobs behind you now in the queue, i.e. the mean queue length should be 10. Little's Rule has been formally proved in quite broad generality, including for non-FCFS priority policies.

The point is that if your simulation program is finding the mean wait anyway, you can get the mean queue length from it via Little's Rule, without any extra code.

3.5 Another Example: Call Center

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

#  patients call in, with exponential interarrivals with rate Lambda1;
#  they queue up for a number of advice nurses which varies through time
#  (initially 1); 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 K; here the newly-arrived patient is counted in the queue
#  length

# usage:

#   python PhoneCenter.py K, 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)
      # the nurses:
      self.Rsrc = Resource(capacity=MOL,qType=PriorityQ)
      self.Mon = Monitor()  # monitor numbers of nurses online
      self.PrSm = PeriodicSampler.PerSmp(1.0,self.Mon,self.MonFun)
      activate(self.PrSm,self.PrSm.Run())
      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
   def MonFun(self):
      return self.NrsCurrOnline
   def Run(self):
      # want to start with only 1 nurse online, so take MOL-1 offline
      for I in range(self.MOL-1):
         yield request,self,self.Rsrc,100
      self.NrsCurrOnline = 1
      # queue starts empty, so start timebomb
      self.TB = TimeBomb(self.TO,self)
      activate(self.TB,self.TB.Run())
      # 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':
            # did this patient encounter an empty system?
            if self.TB:
               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:
               # start new timebomb
               self.TB = TimeBomb(self.TO,self)
               activate(self.TB,self.TB.Run())
            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
            # start new timebomb
            self.TB = TimeBomb(self.TO,self)
            activate(self.TB,self.TB.Run())
            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
      self.NrsPl.WakingEvent = 'timebomb exploded'
      if G.Debug:  ShowStatus('timebomb exploded')
      reactivate(self.NrsPl)

class PtClass(Process):
   SrvRate = None  # service rate
   NPtsServed = 0  # total number of patients served so far
   TotWait = 0.0  # total wait time of all patients served so far
   NPtsInSystem = 0  # for debugging
   def __init__(self):
      Process.__init__(self)
      self.ArrivalTime = now()
   def Wakeup(self,Evt):  # wake nurse pool manager
      reactivate(G.NrsPl)
      # state the cause
      G.NrsPl.WakingEvent = Evt
   def Run(self):
      # changes which trigger expansion or contraction of the nurse pool
      # occur at arrival points and departure points
      PtClass.NPtsInSystem += 1
      if G.Debug:  ShowStatus('arrival')
      self.Wakeup('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('srv start')
      yield hold,self,G.Rnd.expovariate(PtClass.SrvRate)
      yield release,self,G.NrsPl.Rsrc
      PtClass.NPtsInSystem -= 1
      if G.Debug:  ShowStatus('srv done')
      PtClass.NPtsServed += 1
      Wait = now() - self.ArrivalTime
      PtClass.TotWait += Wait
      self.Wakeup('departure')

class ArrivalClass(Process):
   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
   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():
   K = int(sys.argv[1])
   R = int(sys.argv[2])
   TO = float(sys.argv[3])
   initialize()
   G.NrsPl = NursePool(K,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.TotWait/PtClass.NPtsServed
   print 'mean number of nurses online =',G.NrsPl.Mon.mean()

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

3.6 Debugging SimPy Programs

As with any other type of programming, do yourself a big favor and use a debugging tool, rather than just adding **print** statements. See my debugging slide show for general tips on debugging, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`, and I have some points on Python debugging in particular in my introductory Python tutorial, available at my Python tutorials page, `http://heather.cs.ucdavis.edu/~matloff/python.html`.

This section then provides debugging tips specific to simulation programming, especially with SimPy.

3.6.1 Checking Your Simulation Program's Correctness

In simulation situations, we typically do not have good test cases to use to check our code. After all, the reason we are simulating the system in the first place is because we don't know the quantity we are finding via simulation.

So, in simulation contexts, the only way to really check whether your code is correct is to use your debugging tool to step through the code for a certain amount of simulated time, verifying that the events which occur jibe with the model being simulated.

What I recommend is that you add a special function to your code, named something like **ShowStatus()**, which will print out all the current information. You should then have your debugging tool automatically call this function every time you hit a breakpoint.

I also recommend doing these checks first starting at time 0.0, and later again at some fairly large time, say at the halfway point of the total amount of time you wish to simulate (i.e. half of the variable **MaxSimTime** in our examples above). The latter is important, as some bugs only show up after the simulation has been running for a long time.

3.6.2 PDB: Primitive, But a Must-Know

Python comes with its own debugger, PDB. It's very primitive, but it can be made to work well, and it is the basis for other more sophisticated debugging tools. In addition, some of my general remarks on SimPy debugging will be presented in this section. So, this section is "must reading." I assume here that you are familiar with the material on PDB in the appendix on debugging in my Python tutorial.

Know How Control Transfers in SimPy Programs:

Your ability to debug SimPy programs will be greatly enhanced by having some degree of familiarity with SimPy's internal operations. You should review the overview section of this SimPy tutorial, concerning how control transfers among various SimPy functions, and always keep this in mind. Consider for example what happens when you execute your code in PDB, and reach a line like

```
yield hold,self,Rnd.expovariate(ArrvRate)
```

Let's see what will now happen with the debugging tool. First let's issue PDB's **n** (”next”) command, which skips over function calls, so as to skip over the call to **expovariate()**. We will still be on the **yield** line:

```
(Pdb) n
--Return--
> /usr/home/matloff/Tmp/tmp6/HwkIII1.py(14)Run()->(1234,  yield hold,self,Rnd.expovariate(ArrvRate)
```

If we were to issue the **n** command again, the **hold** operation would be started, which causes us to enter SimPy's **holdfunc()** method:

```
(Pdb) n
> /usr/local/SimPy/Simulation.py(388)holdfunc()
-
. holdfunc(a):
```

This presents a problem. We don't want to traipse through all that SimPy internals code.

One way around this would be to put breakpoints after every **yield**, and then simply issue the continue command, **c**, each time we hit a **yield**.

Another possibility would be to use the debugger's command which allows us to exit a function from within. In the case of PDB, this is the **r** (”return”) command. We issue the command twice:

```
(Pdb) r
--Return--
> /usr/local/SimPy/Simulation.py(389)holdfunc()->None
-> a[0][1]._hold(a)
(Pdb) r
> /usr/home/matloff/Tmp/tmp6/HwkIII1.py(29)Run()->(1234, , 0.45785058071658913)
-> yield hold,self,Rnd.expovariate(ExpRate)
```

Ah, there, we're finally out of that bewildering territory.

Always Know What (Simulated) Time It Is:

Again, PDB is not a fancy debugging tool, but it really can be effective if used well. Here for instance is something I recommend you use within PDB when debugging a SimPy application:

```
alias c c;;now()
```

This replaces PDB's continue command by the sequence: continue; print out the current simulated time. Try it! I think you'll find it very useful. If so, you might put it in your **.pdbrc** startup file, say in each directory in which you are doing SimPy work.[16]

Of course, you can also change the alias temporarily to automatically call your function which I suggested earlier:

```
alias c c;;ShowStatus()
```

and make sure that when you write the function you include a call to **now()**.

Starting Over:

During your debugging process, you will often need to start the program over again, even though you have not finished. To do this, first stop the simulation:

```
(Pdb) stopSimulation()
```

Then hit **c** a couple of times to continue, which will restart the program.

If your program runs into an execution error, hit **c** in this case as well.

Repeatability:

The debugging process will be much easier if it is repeatable, i.e. if successive runs of the program give the same output. In order to have this occur, you need to use **random.Random()** to initialize the seed for Python's random number generator, as we have done in our examples here.

Peeking at the SimPy's Internal Event List:

Here is another trick which you may find useful. You can print out SimPy's internal event list with the following code in each of your PEMs:[17]

```
from SimPy.Simulation import _e
```

(Note that if a Python name begins with _, you must explicitly ask for access; the wildcard form of **from...import...** doesn't pick up such variables.)

The internal events list is **_e.events**, and is implemented as a Python dictionary type, showing the events (address of threads) for each simulated time in the future. For example,

[16] Or, put it in one special directory, say your home directory, and run a link from each other directory where you use it.

[17] As of this writing, I don't see why the statement doesn't work if written globally.

```
(Pdb) _e.events
{4.9862113069200458: [<SimPy.Simulation._Action instance at
0x4043334c>], 3.3343289782218619: [<SimPy.Simulation._Action instance at
0x4043332c>]}
```

And as mentioned earlier, you can print out the wait queue for a **Resource** object, etc.

3.6.3 Emacs and DDD

Both of these give a nicer interface to PDB. Again, see my Python tutorial for details on how to use them.

Since both of them use PDB, remarks made above for PDB apply. In particular, I strongly recommend that you use the alias

```
(Pdb) alias c c;;now()
```

or even

```
(Pdb) alias c c;;now());;_e.events
```

DDD has a nice feature whereby specified variables can be displayed constantly at the top of the screen. Make liberal use of it.

3.6.4 SimPy's Tracing Library

SimPy includes a special version of the file **Simulation.py**, called **SimulationTrace.py**, which you may find useful in your debugging sessions. Largely, what these do is to formalize and automate some of the tips I've given above.

3.7 Online Documentation for SimPy

Remember that Python includes documentation which is accessible in interactive mode, via **dir()**, **help()** and PyDoc. See my Python tutorial for details.

Of course, you can also look in the SimPy source code.