A STATE MACHINE LANGUAGE FOR THE UNDERGRADUATE OPERATING SYSTEMS COURSE

Dale E. Parson
Department of Computer Science, Kutztown University of PA
parson@kutztown.edu

ABSTRACT
Assigning projects in an undergraduate operating systems course that includes a mix of software development students and information technology students, many without a background in assembly language and machine architecture, can be challenging. Assigning low level simulation code or operating system modules can lead to frustration and failure for some students. Also, that approach takes too long to explore more than a few basic algorithms. Assigning paper projects may be uninteresting and can lead to rote memorization. This paper explores the approach of requiring students to write state machines, using a notation based on State Diagrams of the Unified Modeling Language, that implement the core logic of various CPU schedulers, page replacement algorithms, disk scheduling algorithms, and other operating system mechanisms. This approach concentrates student efforts more on analysis and less on low level coding. The paper explores the Python-based compiler technology and run-time system used to implement the state machine simulation framework, along with several example assignments, auxiliary tools for visualization, deficiencies in the current framework, and their remedies.

KEY WORDS
operating system; Python; simulation; state machine.

1. Introduction

The work reported in this paper arose out of a need to change the nature of projects in the undergraduate Operating Systems (OS) course at Kutztown University in 2013. Teaching operating systems can be a challenge at any school because of the ever increasing number of subtopics such as multiprocessor systems and machine virtualization. Determining course projects has been particularly problematic at Kutztown because there are two different categories of students and course sequences in the Computer Science Department, namely, those of the Software Development Track (SD), and those of the Information Technology Track (IT). The SD program focuses on the knowledge and tasks of core computer science and software engineering, while the IT program focuses on system administration, network administration, database administration, web development, and security. While IT students learn the same programming basics as SD students in the CS 1-2 course sequence using C++, many IT students do not consider programming to be among their primary educational or career objectives.

Moreover, the prerequisites for the junior / senior Operating Systems course do not emphasize programming or machine architecture. Course prerequisites consist of either the Data Structures or the Information Technology Systems course, completion of 18 credits of CS courses numbered 125 or higher, and a GPA in the CS courses of 2.25. The IT Systems course, the gateway course of the IT Track, does not entail programming. Its prerequisites in turn are the CS 1-2 sequence and an introductory course in Discrete Mathematics (CS 125). Most IT students have taken a Web Programming course that includes development in Javascript and PHP by their junior year. Finally, the sophomore course in Computer Organization and Assembly Language is not a prerequisite.

Thus, many IT students arrive at Operating Systems with only the CS 1-2 background in procedural programming and elementary data structures, and many students have not programmed in assembly or gained knowledge of processor architecture. These facts make creation of interesting and productive course projects a challenge. Requiring students to write operating system simulations in C/C++, or to modify OS modules, is unreasonable and counterproductive. Many conscientious students, all of whom have taken the prerequisite courses, cannot succeed in such projects. Conversely, treating Operating Systems as a “theory” course with paper projects is uninteresting and leads to excess reliance on rote memorization.

The author taught the Operating Systems course for the first time at Kutztown in fall 2013, largely because of his ideas for an approach to projects. The approach is one of having students design high level simulations of operating system algorithms and data structures using a custom state machine (SM) notation based on the constructs of State Machine Diagrams of the Unified Modeling Language (UML) [1]. These state machines compile to executable code that students run as simulations.

The approach explained in this paper does not pertain strictly to the prerequisite structure at Kutztown, however. OS courses with more typical prerequisites can benefit from the approach. Related work takes the form of the two conventional approaches to assignments [2-4]. One is basically "theory" in which students compute scheduling and delay times and other parameters using mechanisms such as queuing theory. The other is modification of low-level operating system modules such as schedulers in a language such as C. Availability of open source operating systems like Linux supports this approach. The problem with the former approach is that students never have the concrete experience of building and debugging a working system. The problem with the latter is that there are too many incidental problems in working with low-level implementation code in a large software system. Focus is too much on the trees and too little on the OS forest.

Some textbooks add simulation as a third form of assignment [2]. The problem is that these simulations are hard-coded in a low-level language. They do not give students the opportunity to engage in high-level analysis and design. The present approach uses simulation, but it differs from existing approaches in several ways. Students design state machines that are high-level abstractions of real implementation methods. They code in an analysis and design language that abstracts away from incidental implementation details. Designing a state machine requires understanding key algorithms and data structures from the top down, yielding a broad perspective and requiring students to tackle an entire problem instead of little pieces of one. Unlike non-programming projects, however, students build something that runs at the end of an assignment. The current approach occupies a middle ground between the two conventional approaches. It is abstract enough to avoid spending excessive time on low-level details of coding, it is expressive enough to capture the central structures and dynamics of actual OS mechanisms, and it is concrete enough to give students the experience of designing and building programs. The remainder of the paper explains the approach, the development of tools, its strengths and its weaknesses.

2. State machines for OS algorithms

2.1 State machine diagram requirements

Figure 1 is an example state machine diagram from our fall 2013 textbook [2] that shows very high level, abstract states for an operating system's scheduling of a process or thread of execution. A label names each *state*, pictured as an ellipse. A label on a directional *transition* signifies an *event* that must arrive at the preceding state that will then drive the state machine into the subsequent state. For example, the state machine of Figure 1 commences execution in the *new* state. Upon arrival of an *admitted* event, signifying OS admission of a process or thread into the processor scheduling sequence of activities, the state machine advances into the *ready* state, from which the state machine can later advance to the *running* state after arrival of a *scheduler dispatch* event. It is while in the running state that a single-threaded process, or a thread of a multi-threaded process, executes processor instructions.

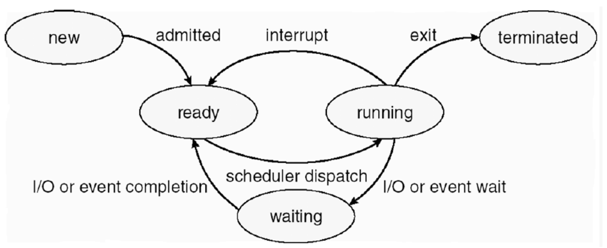

Figure 1: Textbook process-scheduling state machine

Figure 2 shows the required project state machine for the first assignment of fall 2013. This was the introductory project, so the machine is a simpler version of Figure 1. Project goals were to get students used to programming in the form of state machines, to test the underlying infrastructure, and to find out how viable this approach might really be. The sleep(N) procedure simply causes the simulated process to expend N ticks of simulated time, where a *tick* is an abstract unit of time. The yieldcpu() procedure is an alias for sleep(0), used to yield simulation to another state machine object running on another simulation thread. Later assignments replaced sleep(N) with more typical procedure calls for entering and leaving specific OS queues and consuming processor time.

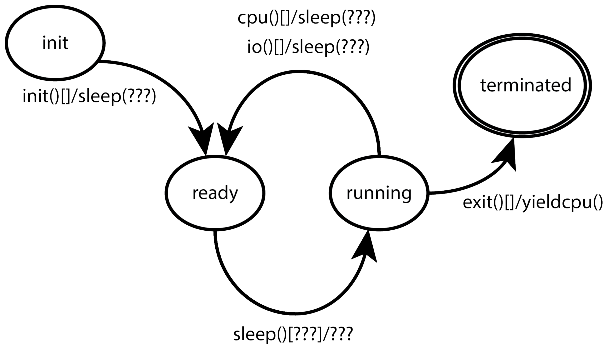

Figure 2: Assignment 1 process-scheduling SM

A key to understanding a UML State Diagram such as Figure 2 is to understand the various fields that can appear tagged to a diagram transition. For example, the following tag appears on the transition from state *init* to state *ready* in Figure 2.

init()[]/sleep(???)

The transition tag fields are as follows.

- *init* is the name of an arriving *event*. The state machine framework created for this course assigns the name *init* to the initial state, as well as to the initial event, for any state machine. State machine *actions* generate additional state machine events as discussed in the action bullet below.

- An arriving event may carry *arguments* between the parentheses following the event name. None of the events in Figure 2 carry arguments. Later examples do. Arguments are available for processing by subsequent tag fields on a transition.
- A state ignores the arrival of an event type not matched by an event name in any outgoing transition.
- The square brackets following the event argument list enclose an optional boolean *guard condition*. Similar to boolean tests in procedural programming languages, a guard expression tests conditions and relationships among event arguments and *state variables* defined by the state machine writer. If a guard condition's expression evaluates to true after binding the event arguments on its transition and resolving any state variables, the state machine takes the transition. When the guard expression is empty, as it is for most transitions in Figure 2, it evaluates to true upon event arrival. The event, alone, is enough to trigger the transition in the absence of a guard condition. The guard condition of "???" on the bottommost transition of Figure 2 was a problem for students to solve, to be discussed below.
- Finally, an optional tag field following the / delimiter comprises a UML *action*, which is a sequence of executable statements. The state machine framework used in the course supports, within one action, execution of multiple statements that can include *procedure calls*, *function calls*, *arithmetic* and *logical expressions*, and *assignment* to *state variables*. The framework supplies a high level library of procedures and functions that provide the basic simulation actions for students. Later assignments required the students to replace calls to high level procedures with student-supplied state machines that implemented the semantics of those procedures.
- One initial requirement of the framework is for the final action step in a transition's action to consist of a call to a procedure that generates a deferred event leading out of the subsequent state. In Figure 2, the /sleep(???) *actions* on the transitions leading into the *ready* state schedule the completion-of-sleep() *event* that leads out of the ready state.
- Later assignments supported communication between state machines simulating threads of execution by allowing a transition to wait for an event generated within a different transition in another state machine instance. There can be multiple, active instantiations of a single state machine. This communication extension goes beyond Figure 2, where the final action on every transition schedules the event type that matches a subsequent outgoing transition.
- A transition's action is optional. When the action is missing, the only state variable modified by taking the transition is the identity of the state itself.

The preceding bullets give the standard interpretation of UML tag fields appearing on a transition. To summarize, a transition tag identifies the *event type* to which the transition responds, optional *arguments* that accompany the event type, an optional boolean *guard condition* that suppresses the transition when false, and an optional sequence of *action* steps that can invoke library procedures and functions, evaluate expressions, assign values to variables, schedule later events, and in later versions of the framework, send events to other state machine instances that are simulating threads of execution.

2.2 State machine diagram tools

The author investigated several open source, graphical state machine capture tools that generate simulation code of some form. None of the open source frameworks met the project requirements. All of them over-couple a graphical front end to the details of the simulation back end. The Simulink framework [5] could have done the job, but its price was beyond any available budget.

The author settled on the approach of creating a text-based notation consisting of the UML tag fields and syntax illustrated in Figure 2, using the symbol "->" in place of a graphical transition. This approach is favourable for a number of reasons. First, every construct in Figure 2 except for the ovals and arrowed arcs takes the form of text. Second, it is easy to incorporate declaration of named states such as *init* and *ready* into a textual state machine language.

Third, it is straightforward to implement both the compiler and the run-time system for the state machine language using the Python programming language [6,7]. The author had previously used Python in undergraduate and master's level compiler design courses. Python comes with an excellent LALR parser generator called PLY [8] modeled after the C-based YACC generator of Unix [9]. The Python language and its powerful regular expression library integrate seamlessly into PLY. Beyond the scanning and parsing stages of a compiler, Python has built-in, notational support for mutable and immutable sequences of objects, sets of objects, and maps from one object set to another, making construction of annotated parse trees, abstract syntax trees, and symbol tables easy for a knowledgeable compiler writer. Also, Python readily supports dynamic loading of plug-in code generators for alternative compiler target virtual machines.

Implementing the run-time system for the state machine language is helped immensely by the fact that Python's run-time virtual machine includes the *eval* function and *exec* procedure for interpreting source code. The guard expressions and action steps of Figure 2 consist of Python expressions, statements, and assignments. Python uses algebraic expressions, including function calls and assignment statements, and many of the guards and actions of the Python-based state machine language look very similar to procedural statements in Java. The back end of the SM compiler generates stylized Python source code that runs within the dynamic context of a multithreaded state machine simulator. The Python VM supplies much of the run-time machinery of the simulator. The course's simulation framework allocates one Python thread per state machine instance, but it schedules only one such thread to execute at a time. Using one thread per state machine allows state variables to reside in each thread's stack, simplifying the job of state machine context switching within the simulation scheduler. This approach is basically cooperative multitasking within a Python-based simulation framework. Python is not a fast simulation framework, but its interpreted nature minimizes the work load on a full-time teaching professor in creating and supporting such a framework. A 64-context Sparc multiprocessor was available for exclusive use by the two sections of students in the fall 2013 course offering, so somewhat long-running (5 minutes per simulation), CPU-intensive simulations were not a problem. A PC per student would have worked as well.

```
# A comment line begins with '#' as the first token.
machine processor {
    start init, state afterstart, accept alldone ;
    init -> afterstart init()[]/@fork()@,
    afterstart -> alldone fork(pid, tid)[]/@
        idle(0)@;
}
machine thread {
    enteredRunning = 0 ;
    start init, state ready, state running, accept terminated ;
    init -> ready init()[]/@sleep(50)@,
    ready -> running sleep()[@enteredRunning == 0@]/
        @enteredRunning += 1 ; cpu(10)@,
    ready -> running sleep()[@enteredRunning == 1@]/
        @enteredRunning += 1 ; io(-1)@,
    ready -> running sleep()[@enteredRunning > 1@]/
        @trigger(1, "exit")@,
    running -> ready cpu()[]/@sleep(50)@,
    running -> ready io()[]/@sleep(50)@,
    running -> terminated exit()[]/@sleep(0)@;
}
processor
```

Listing 1: Implementation of Figures 1 & 2

Listing 1 shows the full solution to the students' first assignment. The paired '@' delimiters simplify lexical analysis. Later assignments were much bigger and more complex. The goals of this assignment were to get some practice for the students, the instructor, and the new tools. There are actually two state machine types in Listing 1. The *processor* state machine compiles to a Python class that houses simulated functional units such as I/O units, and that provides executable code that starts process simulations. It declares three states, *forks* one simulated process in its init -> afterstart action, and then terminates. The transitions of processor state machines in later projects fork many more simulated processes and threads.

State machine *thread* is the main machine of interest in the course assignments. There is also an I/O state machine built into the language and run-time environment, that the author intended to expose for student programming late in the semester, but time ran out. All student work consisted of writing and extending thread state machines. Note that there are two uses of the word *thread* in this discussion. One is a Python class, for example as defined in Listing 1, that simulates an OS thread of execution. There can be many objects of this class, each one constructed by a call to the *fork* procedure, as it appears in Listing 1, from within a processor or thread state machine. The other use of the word *thread* is for a Python thread. Each instantiated state machine object is an *active object* in UML parlance, meaning that the object runs in its own thread of execution. Each state machine definition like those in Listing 1 defines a class, and each object of such a class runs on a thread. The simulator schedules only one Python thread at a time, implementing cooperative multitasking, in the interest of simplifying maintenance of per-state-machine variables as previously discussed.

The code of Listing 1 has a not-quite one-to-one correspondence to the diagram of Figure 2. Code for state machine *thread* initializes a state variable *enteredRunning* with a value of 0. State variables need not be declared, and they may be initialized dynamically within transition actions, although initializing them as in Listing 1 helps to highlight their presence. Python supports run-time typing of variables. A Python variable is a reference to an object of primitive or composed type, and a variable may bind to objects of different types at different times. Variable *enteredRunning* counts how many times execution has entered the running state during execution. There are three different ready -> running transitions, each triggered by the end of a sleep() event – a transition's event arrives when the activity corresponding to the event has completed – and each with a different guard condition that tests the value of *enteredRunning*. The first temporal transition with "enteredRunning == 0" increments the variable and schedules a cpu() event after 10 ticks in the running state. The second transition similarly increments the variable and schedules an io() event. The third transition triggers an *exit* event by using the *trigger* procedure, which is an extension mechanism that supports the creation of application-specific event types. This event takes the state machine into its *terminated* state, which the code of Listing 1 shows to be an *accept state*. Three types of states appear in Listing 1, namely, a unique *start state* for each state machine, zero or more regular states, and one or more accept states. Entry into an accept state terminates the simulation thread for its machine object.

Despite the fact that capture of a UML State Diagram is readily supported by textual programming notation, availability of a graphical diagram is desirable for documentation and debugging. The solution is to have the state machine compiler back end generate the directed acyclic graph (DAG) notation accepted by the Graphviz set of open source visualization tools [10], and then to use Graphviz utilities to generate JPEG graphical images.

Figure 3 shows the graph generated from the *thread* SM code of Listing 1. A triangle denotes the start state and an octagon denotes an accept state. A label on each state shows the state's machine name, state name, and an internal number used for debugging the compiler and run-time simulator. A label on each transition shows the event type and declaration order of the transition with respect to its predecessor state. Order of transition declaration is important for two transitions with the same event type, both with true guard conditions, leaving the same state. Order of transition declaration is the tie breaker that determines the transition to cross in such cases.

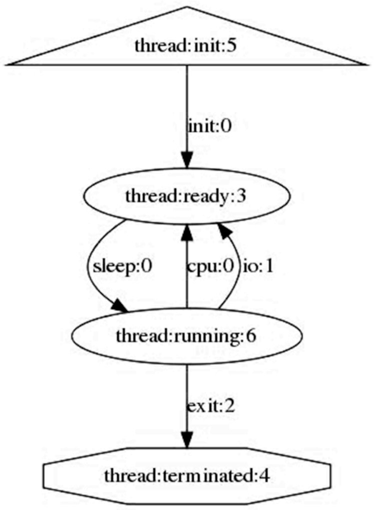

Figure 3: Generated graph for SM *thread* of Listing 1

Listing 2 shows the generated Python simulation code for SM *processor* of Listing 1. This code resides within Python class *processor*, which is a subclass of framework class __Processor__ that defines the framework fields and methods for simulating a processor. Generated class *thread* and its base class __Thread__ have a similar relationship. The "self." notation in Listing 2 is similar to the "this." notation in Java or "this->" in C++, denoting a reference to the current object. Use of "self." in Python is mandatory.

The simulation thread for a state machine runs in an infinite loop starting at line 1 in Listing 2 until hitting a return statement or an exception. Termination and exception handling code (not shown) synchronizes termination of a simulation thread with the simulation scheduler. Line 2 checks whether the scheduler has tagged termination-of-sleep data onto the SM object as the result of event completion. If so, line 3 updates the object's simulated time, event, and event arguments, and line 4 logs the arrival of the event. Otherwise, lines 6 through 8 set these simulation variables to their default values.

Lines 10 through 27 show the code that the compiler generates directly from SM *processor* in Listing 1. The structure is a hierarchical set of nested if-else statements that check the current state type, then the event type , then optional guard conditions (none in Listing 2). Guard conditions, when present, contribute an additional level of *if* testing, using Python's *eval* function to evaluate a guard's boolean expression. When a transition's conditions evaluate to true, the generated code logs the state change and then uses Python's *exec* procedure to interpret each step of the transition action. Exec's arguments *globals* and *locals* limit the scope of any assignments within an exec'd action to the variables defined and stored in those two variable sets. A state machine's state variables reside in its local set on the simulation thread's stack; the immutable global set primarily contains bindings to simulator library functions.

```
1    while True: # processor runs until return.
2        if self.__sleepResult__:
3            stime, event, args = self.__sleepResult__
4            self.logger.log(self)
5        else:
6            stime = self.scheduler.time
7            event = None
8            args = None
9        # generates custom run() code here.
10       if self.state == 'init':
11           if event == 'init':
12               self.logger.log(self, tag="DEPART")
13               self.state = 'afterstart'
14               self.logger.log(self, tag="ARRIVE")
15               exec('fork()',globals,locals)
16               continue
17       elif self.state == 'afterstart':
18           if event == 'fork':
19               locals["pid"] = args[0]
20               locals["tid"] = args[1]
21               self.logger.log(self, tag="DEPART")
22               self.state = 'alldone'
23               self.logger.log(self, tag="ARRIVE")
24               exec('idle(0)',globals,locals)
25               continue
26       elif self.state == 'alldone':
27           return
```

Listing 2: Generated code for *processor* of Listing 1

Lines 19 and 20 show the binding of local variables *pid* (process ID) and *tid* (thread ID) that accompany a *fork* event as its arguments. A library manual documents the parameter types of library procedures and functions, and the argument types that accompany events.

Some library procedure calls such as sleep(50) or cpu(10) block until the invoking thread reaches the front of a simulator priority queue that is sorted on simulated time. During the time that a blocked simulation thread is waiting for its next turn to run, the simulation scheduler performs a context switch to a different simulation thread that is ready to run. The simulator mirrors OS CPU scheduling mechanisms. There is no need to explicitly save and restore the state of a state machine object, because Python's per-thread stack contains that state.

```
000000000000,LOG,processor 0,init,ARRIVE
000000000000,LOG,processor 0,init,init
000000000000,LOG,processor 0,init,DEPART
000000000000,LOG,processor 0,afterstart,ARRIVE
000000000000,LOG,processor 0,afterstart,fork
000000000001,LOG,thread 0 process 0,init,ARRIVE
000000000001,LOG,thread 0 process 0,init,init
000000000001,LOG,processor 0,afterstart,DEPART
000000000001,LOG,thread 0 process 0,init,DEPART
000000000001,LOG,processor 0,alldone,ARRIVE
000000000001,LOG,thread 0 process 0,ready,ARRIVE
000000000001,LOG,processor 0,alldone,idle
000000000001,LOG,thread 0 process 0,ready,sleep
000000000001,LOG,processor 0,<defunct>,?
000000000051,LOG,thread 0 process 0,ready,DEPART
000000000051,LOG,thread 0 process 0,running,ARRIVE
```

Listing 3: Initial lines of a simulation log file

The simulation framework logs all state changes and event handling into a log file, as driven by log statements such as those appearing in Listing 2. Listing 3 shows the initial lines of a log file for the simulation of the assignment 1 state machine discussed so far. Fields in a log line include the simulated time in ticks, the keyword LOG or MSG, the ID of the state machine logging the line, the state name, and either the event type or the state arrival or departure status in the last column. The log file can also contain MSG entries that record arguments to a message procedure invoked as part of transition actions, usually to output student debugging information. Post-simulation, custom Python data extraction tools analyze log files to determine statistical measures (mean, max, min, standard deviation, etc.) on time spent in each state, on transition crossings, and on event arrivals. Statistical analysis provides the basis for determining the effectiveness of a simulated algorithm and for grading student projects.

3. Realistic simulation of OS algorithms

3.1 Statistical sampling and operating system queues

The state machines of Listing 1 are overly simplistic. This section considers a more realistic set of processor scheduling algorithms comprising student assignment 2. The next section considers some of the limitations of the current framework in the context of simulating alternative page replacement strategies for demand paging in student assignment 3.

The simulation framework library contains function *sample* that is an essential building block in our stochastic simulations. This function allows students to sample value ranges according to various distributions that correspond to realistic time and space distributions in processes, threads, I/O wait times, and other OS mechanisms. Listing 4 gives the documentation for using *sample*.

sample(lower, upper, distType, *parameters)
Return an integer in the inclusive range [lower, upper], where lower and upper are integers, and distType and parameters covary as follows.
distType = 'uniform' gives a uniform distribution in the range [lower, upper] with parameters ignored.
distType = 'gaussian' gives a Gaussian distribution in the range [lower, upper] with parameters (mu, sigma) where mu is an int or float within the range [lower, upper] that is the distribution mean, and sigma is the standard deviation. Results outside the range are discarded until a valid value is found and returned.
distType = 'exponential' gives a exponential distribution in the range [lower, upper] with parameter (mu,) where mu is an int or float within the range [lower, upper] that is the mean; the closer it is to lower, the steeper the drop off. Results outside the range are discarded until a valid value is found and returned.
distType = 'revexponential' gives a reverse exponential distribution in the range [lower, upper] that grows towards upper, with parameter (mu,) where mu is an int or float within the range [lower, upper] that is the mean; the closer it is to upper, the steeper the rise. Results outside the range are discarded until a valid value is found and returned.

Listing 4: Documentation for the *sample* function

Figure 4 shows the distribution of values returned from a series of 100,000 calls to sample(1, 100, 'exponential', 10.0). The values 1 and 100 give the inclusive lower and upper limits of the sampled range, 'exponential' gives the distribution type documented in Listing 4, and 10.0 gives the knee in the curve of Figure 4, with half of the 100,000 sampled values lying at or below this value 10.0. The revexponential distribution gives a mirror distribution with the high sample count appearing at the right end of the graph. Exponential sampling is appropriate for simulating CPU time bursts in an I/O bound thread. Treating sampled values as CPU ticks, the majority of CPU bursts are relatively small values. Conversely, revexponential applies to CPU bound threads, because sampling provides a majority of CPU bursts that are high values.

Student project 3 that simulated page replacement algorithms in demand paging also made effective use of several sampling distributions. The exponential sample was effective in simulating threads with good locality of reference, because most samples are some relatively small offset from a reference location in memory. Conversely, uniform distribution is appropriate for simulating threads with poor locality of reference, because each uniform memory reference has no correlation with recently preceding references.

One other essential piece of library infrastructure for simulating OS algorithms is a queue class. Python provides various composite data types built into the language. Its library supplies many more, including module *heapq* that provides a min-heap implementation of a priority queue. Custom simulation class *Queue* provides a framework mechanism whereby students can use either FIFO or priority queues, with an extra numeric enqueue argument supplying each entry's priority value for a priority queue. Students construct scheduling queues, I/O queues, message channels, and other OS queues by invoking the Queue constructor and the methods of the Queue class on a Queue object. Queue contains the usual methods for FIFO and priority *enqueue*, *dequeue*, *peek*, and *length* operations, and additional methods for removing an entry from the middle and for reordering a priority queue when priorities change, for example as required by an algorithm that ages entries.

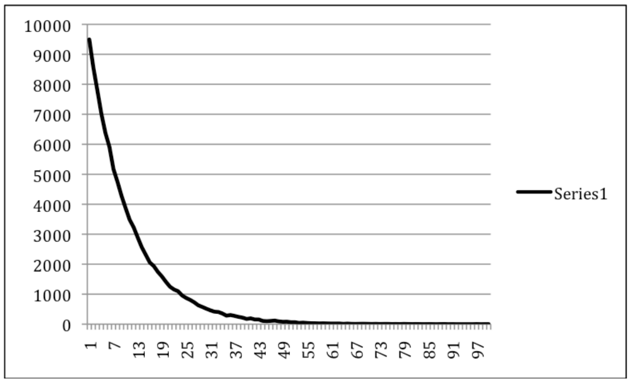

Figure 4: sample(1, 100, 'exponential', 10.0) X 100,000

3.2 Simulating CPU scheduling algorithms

Figure 5 illustrates the code for the author's handout of the *first-come, first-served* algorithm that schedules threads onto simulated hardware contexts in the order of arrival. Half of the threads are I/O bound, and the others are CPU bound, as determined by the initialization of state variable *iobound* on the *init* transition.

```
iobound = True if ((pid % 2) == 1) else False
```

At the start of each CPU burst, the transitions into the *scheduling* state determine the number of ticks in the next CPU burst using the *sample* function.

```
ticks = sample(1, 250, 'exponential', 25) if iobound
        else sample(100, 1100, 'revexponential', 1000);
```

Guard conditions on transitions leading out of *scheduling* check to determine if there is a free hardware context. If there is, the transition locks a context and leads to the *running* state. Otherwise, that state machine's simulation thread enters a FIFO queue associated with the *ready* state (the *readyq*), waiting for another thread to dequeue, and thus awaken, the next ready thread when the previous thread completes its use of a hardware context.

After a thread completes its CPU burst, it enters the *rescheduling* state, from which it pseudo-randomly selects an I/O device via the *sample* function. The framework simulates both slow terminal and fast I/O devices, with I/O delay determined by both device delay and queuing delay caused by multiple threads queued on an I/O device. The thread invokes the io(unit) procedure and then enters the *waiting* state until notification of completion via an io() event, after which it resumes CPU *scheduling*. Threads alternate between realistic CPU bursts and I/O bursts until they exceed a limit on simulated time set by project requirements (and entered on the simulator invocation command line), after which they enter the *terminated* state at the next opportunity within the state machine.

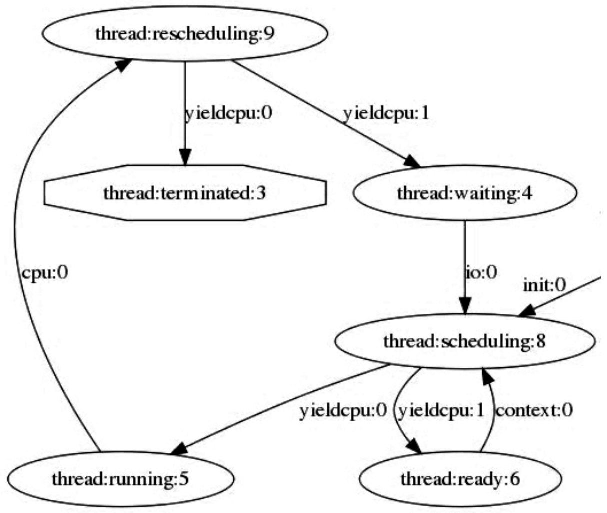

Figure 5: First-come, first-served CPU scheduling

It was necessary to add the ability to send events from one state machine object to another in order to allow a thread that is releasing a hardware context to dequeue and signal a waiting thread for assignment 2. A thread entering the *ready* state enqueues itself and waits using the following code. In this code, *thread* is the "this reference" to the state machine object that is simulating an OS thread.

```
readyq.enqueue(thread); waitForEvent('context')
```

Some time later, another thread state machine that has completed its CPU burst dequeues a waiting thread SM as follows, using Python's conditional expression similar to the "?:" construct of C or Java.

```
signalEvent(readyq.dequeue(), 'context')
    if len(readyq) > 0 else noop();
```

Students were required to rewrite the first-come, first-served scheduler as a shortest-job first scheduler, and again as a round-robin, preemptive scheduler. The state machine graph for shortest-job first is identical to Figure 5. The only change is replacement of the FIFO scheduling queue with a priority queue ordered on CPU ticks. A thread entering the ready state enqueues itself as follows.

```
readyq.enqueue(thread, ticks);
waitForEvent('context')
```

Since fewer ticks in a CPU burst correspond to shorter jobs, the *ticks* variable serves as the priority value for the priority *readyq*.

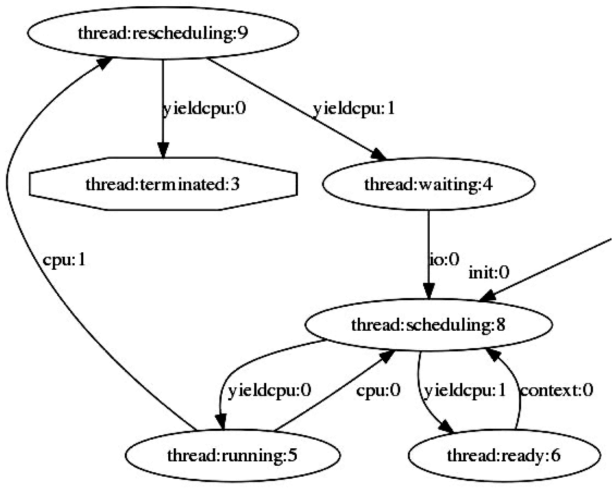

Figure 6: Round-robin CPU scheduling

Round-robin is more complex because, even though it uses the same FIFO queuing approach as first-come, first-served, it is preemptive. Students simulated preemption by maintaining extra state variables that kept track of immediate ticks to use and ticks to defer, based on the CPU burst *ticks* value (as before) and on the preemptive scheduling *quantum* value, which was specified in the assignment. The state machine graph of Figure 6 for round-robin shows an additional transition from *running* back to the *scheduling* state for cases where a thread has exhausted its quantum, but not its CPU burst. There was an assortment of student solutions to the problem, some with other states or transitions. Tolerance in the statistical analysis of the log files allowed student solutions within expected simulated time limits to pass automated tests.

Perhaps the major change in student experience from previous courses was the increase in the analysis-to-coding time ratio. Particularly given the fact that the author handed out a solution to one algorithm for each of assignments 2 and 3, and then students had to rewrite handout code using different algorithms, a lot of the solution code was already in place at the start of the project. However, learning how to use state machines, stochastic simulation, FIFO and priority queues, and inter-thread communication were all new topics for the students. Successful analysis of a problem solution was more like solving a puzzle or a proof than it was like conventional coding. Most students seemed to like the approach, and moreover, the approach allowed them to focus on high-level OS algorithm and data structure considerations. The simulation framework took care of many of the low-level coding details, once the students became conversant in using the state machine notation and the simulation library functions and procedures.

3.3 Simulating page replacement algorithms

For assignment 3 the author handed out a solution for a FIFO page replacement algorithm, which students had to rewrite into a least-recently used (LRU) solution, with an additional requirement to extend LRU with secondary priority ordering to prefer discarding unmodified pages over writing out pages modified since they were last paged into memory. Space precludes going into detail, but it is instructive to look at a few data initializations. The assignment used multiple simulated processes, each with multiple simulated thread state machine objects. OS threads within an OS process share a process control block object (PCB) that students can extend. Below are PCB data field initializations for the FIFO paging solution. This code resides in an initialization transition in the first thread of each process.

```
pcb.pagecount = sample(10, 100, 'uniform');
pcb.framecount = int(pcb.pagecount / 4);
pcb.pagetable = [[-1, 0, None]
          for p in range(0, pcb.pagecount)];
# Above page table triplets are frame number, dirty bit,
# and a reference to an entry in the victim queue.
# Initially none of the pages are mapped to frames.
pcb.freeframeQueue = Queue(ispriority=False);
for f in range(0, pcb.framecount):
    pcb.freeframeQueue.enq(f);
pcb.victimQueue = Queue(ispriority=False);
pcb.waitingForFrameQueue = Queue(ispriority=False);
```

Those PCB fields give the process page count, the physical frame count, the page table, the queue of free physical frames, the queue of pages waiting to be victimized (paged out, in this case in FIFO order), and the queue of OS thread objects waiting for a free frame for reading in a page from simulated disk.

The main logical extension from assignment 2 was the trifurcation of the *running* state into *running-from-cpu-registers*, *running-interacting-with-memory*, and *waiting-for-paging*. The latter state introduces a new form on I/O, paging, not seen in assignment 2. In addition, assignment 3 added a somewhat ad hoc *paging thread*, with its own subtree of the state machine graph, to manage the details of simulating paging out dirty pages, paging in requested pages, and synchronizing with waiting application threads. The project was a success, although the resulting state machines for the three page replacement algorithms were somewhat more complicated than necessary. The next section takes up the causes of these complications and their planned remedies.

4. Deficiencies and planned enhancements

Assignment 3 had numerous similar state transitions onto which the author and students had to copy and paste identical or near-identical action code. The usual solution in procedural programming is to write helper functions and procedures, and then to invoke these helpers rather than copying and pasting source code. Almost none of the students were Python programmers at the beginning, but by assignment 2 they had become capable of writing sequences of actions as already discussed.

The solution for the next offering of the course in fall 2014 is to add two new constructs to the state machine language, namely macros and functions. The *macro* construct will be a replacement for copying and pasting in-line code. A student will define a named macro within a state machine, after the section that initializes state variables. A macro will take optional parameters, and it will serve as the location of code that would otherwise appear repeatedly on state transitions. Those transitions will now invoke a macro instead of repeating code. The *function* construct will be similar, except that it will be a function in the sense of returning a value that is a mapping of its arguments. The main distinction for the state machine compiler is one of scope. A macro will use the variable bindings of its caller, so that it may mutate state variables as necessary. A function will use its parameter bindings in a call-by-value approach. These constructs will be straightforward to add to the language and run-time environment.

Another problem made evident by assignment 3 is the need for yet another form of subroutine. In assignment 3 the relationships of individual states and transitions of demand paging were complicated enough that the author broke them off into a separate subgraph of the thread state machine, executed only by thread 0 in each process. That dedicated thread became the *pager thread* for the process, interacting with other so-called "application threads" via fields in the PCB enumerated in section 3.3. While this approach is realistic, it is not typical. The Java Virtual Machine, for example, allocates a number of specialty service threads for tasks such as garbage collection [11], but the approach is not common among non-Java processes. For this course it was somewhat of a kludge.

Fortunately, the kludge is close to the preferred solution. The correct solution is to extend the current state machine approach into one of a *push-down automaton*. The architecture will augment the scalar *state* variable with a stack of machine-state pairs to be resumed upon termination of the current state machine. When a SM thread needs to perform a complex paging algorithm, for example, it will push its state to this stack and enter a lower-level *paging state machine*. That state machine will manage the details of paging, and it will be entered by any thread requiring the algorithm. It will no longer be necessary to reserve one thread per process as the ad hoc pager thread. Paging is just one example of a complex OS algorithm that we do not wish to over-couple into other OS algorithms such as CPU scheduling. The ability to push down into a state machine such as a *paging state machine*, and then to pop back upon completion of its work, promises to make state machines easier to extend into more complicated algorithms. Like macros and functions, it provides a form of subroutine modularity. It is also more realistic in terms of modeling the mechanisms of real operating systems.

5. Conclusion

The fall 2013 operating system course saw record enrollment for recent years, splitting into two sections of about twenty students each. Almost every student completed all projects, and earlier assignments extended into later assignments in a fairly straightforward manner. The state machine approach integrated well into the lectures, into textbook materials, and into exams. Students seemed generally positive about the approach, once they got beyond the novelty of it.

The projects also demonstrated the use of several inter-thread synchronization mechanisms not discussed in the sections above. The CPU scheduler of assignment 2 used a counter of free hardware contexts for scheduling that was essentially a *counting semaphore*.

Next, the

```
waitForEvent('context')
signalEvent(readyq.dequeue(), 'context')
```

procedure pairs provided a form of *condition variable* for synchronization of multiple OS threads.

The author explored portions of the multithreaded simulator implementation with students during lecture in order to illustrate scheduling and multithreading concepts.

Anticipated extensions to the simulation framework include the provision of macros, functions, and nested state machines via push-down automata as already discussed. The author will teach two sections of the Operating Systems course in fall 2014, and will use this framework with these extensions. Plans also include starting the example assignment 1 earlier in the semester, so that we have time to get to a fourth assignment that simulates some aspect of I/O such as disk scheduling.

The approach is not limited to the Operating Systems course with Kutztown's prerequisite structure. It occupies a middle ground between theory and practice, allowing a student to concentrate on essential aspects of algorithms and data structures in a top-down manner, ignoring many low-level implementation details that are incidental to understanding, while at the same time building a software system that runs and exhibits interesting behavior. High-level analysis and design are the primary uses of the Unified Modeling Language. Unlike many UML models, however, the models of this framework actually run and produce results.

The framework is also promising for exploration of network protocol algorithms in courses with a data communication focus. State machines often appear as part of the specification of communication protocols [12].

The author is willing to share all source code and documentation with PACISE attendees who are interested in using the framework. It is designed to work on any system running Python 2.6 or later (untested on Python 3.X at present), GNU make, and the bash shell, including Unix, Mac / OSX, and presumably Windows running Cygwin. Users must install the PLY parser generator [8] and the Graphviz visualization tools [10].

References:

[1] M. Fowler, *UML distilled*, Third Edition (Boston, MA: Addison-Wesley, 2003).

[2] A. Silberschatz, P. Galvin and G. Gagne, *Operating system concepts*, 9th Edition (Hoboken, NJ: John Wiley & Sons, 2012).

[3] W. Stallings, Operating systems: internals and design principles, 7th Edition (Upper Saddle River, NJ: Prentice Hall, 2011).

[4] A. Tanenbaum, Modern operating systems, 4th Edition (Upper Saddle River, NJ: Prentice Hall, 2014).

[5] MathWorks, *Simulink* (http://www.mathworks.com/products/simulink/, 2014).

[6] *Python programming language* – official website (http://python.org/, 2014).

[7] D. Beazley, *Python Essential Reference*, Fourth Edition (Boston, MA: Addison-Wesley, 2009).

[8] D. Beazley, *PLY* (Python Lex-Yacc) (http://www.dabeaz.com/ply/, 2008-2014).

[9] D. Brown, J. Levine and T. Mason, *Lex and YACC* (Sebastopol, CA: O'Reilly Media, 2012).

[10] *Graphviz - Graph Visualization Software* (http://www.graphviz.org/, 2014).

[11] Lindholm, Yellin, Bracha and Buckley, *The Java Virtual Machine Specification*, Java SE 7 Edition (Boston, MA: Addison-Wesley, 2013).

[12] Internet Engineering Task Force, *State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator*, (http://tools.ietf.org/html/rfc4137, 2005).