Starving Philosophers: Experimentation with Monitor Synchronization

Steven Robbins
Division of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

Abstract

Textbook discussions of synchronization seldom go beyond a brief introduction in terms of classical problems. This paper presents a simulator for the monitor solution of the dining philosophers problem that students can use to experimentally explore how such a solution might behave in practice. The simulator, which can be run remotely from a browser or can be downloaded for running locally, is written in Java so that it can be run on almost any system.

1 Introduction

Most operating systems textbooks illustrate synchronization techniques such as semaphores, monitors, and message passing by solving classical synchronization problems (bounded buffers, readers and writers, and dining philosophers) but do not address implementation issues that might affect the behavior of solutions in practice [6, 8, 9, 10]. As a result, students have little real understanding of how synchronization actually works. This paper presents a simulator that is designed to be used in conjunction with a standard operating systems textbook to allow students to experiment with these concepts. The simulator can be used to introduce students to key techniques of experimentation [11, 12, 13] while exploring monitor behavior and the idea of starvation as it applies to the dining philosophers problem. Monitors were chosen for the simulation because of the emergence of the Java language which uses monitor concepts for its synchronization. The dining philosophers problem is an interesting example, because there is a trade off between optimization of parallelism and elimination of starvation. The next section of the paper discusses monitors and how the choice of implementation might affect behavior. The classical dining philosopher's problem is reviewed in Section 3 and the problem of starvation is discussed. Sections 4 and 5 introduce the simulator, and Section 6 shows how the simulator might be used to understand different implementation issues. Section 7 discusses resources available to support the use of this simulator.

2 Monitors and Monitor Classification

A monitor is a synchronization construct that enforces mutual exclusion. Monitors are typically supported by a programming language rather than by the operating system. They were introduced in Concurrent Pascal [1] and are the synchronization mechanism used in the Java language. A monitor contains code and data. All of the data and some of the code may be private to the monitor, accessed only by code that is part of the monitor. Each monitor has a single lock, and a task must acquire the lock to execute monitor code. The task that owns the monitor lock is called the *active* task. Only one task can be active in the monitor at a time.

A task can acquire the monitor's lock through one of several monitor queues. It gives up the lock by returning from a monitor method or by blocking on a condition variable.

A condition variable is not a variable at all. In fact it is just a queue that is part of the monitor. Sometimes these are called *event queues*, but we will use the expression *condition variable queue* here. A condition variable queue can only be accessed with two monitor methods associated with this queue. These methods are typically called *wait* and *signal*. The *signal* method is called *notify* in Java.

A task that holds the monitor lock may give it up and enter a condition variable queue by executing the corresponding wait method. A task that holds the monitor lock may revive a task waiting in a condition variable queue with the notify method of that queue. The notify method removes one task from the condition variable queue if the queue is not empty. Since the notifying and notified tasks may not both be active in the monitor simultaneously, at least one of these tasks must block. A task is blocked by putting it in a queue. The behavior of a monitor is determined by the scheduling of four types of queues.

Condition variable queues: There may be any number of condition variable queues for a given monitor. Each queue has associated wait and notify methods for putting tasks in the queue and taking them out.

The entry queue: Each monitor has one entry queue. When a task attempts to access a monitor method from outside the monitor, it is put in the monitor's entry queue.

The signaller queue: Each monitor has one signaller queue. When a task performs a notify, it is put in this queue.

The waiting queue: Each monitor has one waiting queue. When a task is removed from one of the condition variable queues, it is put in this waiting queue.

The last three of these queues will be referred to as *monitor queues*. Processes in the monitor queues are waiting to acquire the monitor lock. A monitor has one of each of these queues, but in some implementations these queues may be combined.

When the monitor lock becomes free (because the task holding the lock returns from a monitor method or enters one of the queues), the tasks in the three monitor queues compete for the lock. The behavior of the monitor is determined by the relative priorities and queue disciplines of the three monitor queues.

In the monitors we will discuss here, the signaller queue will have the highest priority of the three monitor queues. When a task executes a notify method, it is put in this queue and loses the monitor lock. However, since this queue has the highest priority, it immediately regains the lock. This is equivalent to not having lost the lock at all, so for the rest of this paper we will ignore the signaller queue.

The queues of a monitor (ignoring the signaller queue) are shown in Figure 1. The arrows show how tasks can move from one queue to another.

Monitors are classified by the scheduling priorities of the monitor queues. If we ignore the signaller queue, there are three possibilities.

Monitors in which the entry queue has a higher priority than the waiting queue have unacceptable behavior [2] and have not been given a name. We will call these *Entry Priority* monitors. Monitors in which the waiting queue has a higher priority than the entry queue are generally called *Signal and Continue* monitors [2, 4] and those in which the two queues have equal priority are called *Wait and Notify* monitors [2, 5].

The queuing discipline also affects monitor behavior. For each of the monitor classifications we consider three queueing disciplines: FIFO, LIFO and random.

3 Dining and Starving Philosophers

The dining philosophers problem is discussed in many textbooks as a classical problem illustrating the concepts of critical regions and synchronization [6, 8, 9, 10]. Five philosophers sit around a circular table. In front of each philosopher is a plate of rice and between each plate is one chopstick. In order to eat, a philosopher must pick up both chopsticks next to his plate. Each philosopher is in one of the states: thinking, hungry or eating. When a thinking philosopher becomes hungry, he changes to the hungry state and attempts to get both of the needed chopsticks. If he can do so, he changes to the eating state, eats for a while, and then puts down the chopsticks and reenters the thinking state. (This problem is usually described in terms of forks and spaghetti, but spaghetti is typically eaten with a single fork or a fork and a spoon rather than with two forks.)

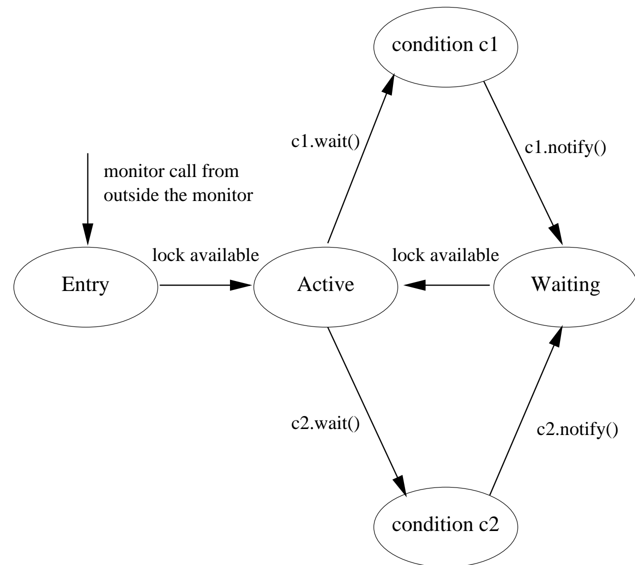

Figure 1: The queues associated with a monitor.

Several operating systems textbooks give essentially the same monitor solution to this problem [6, 8]. Most discussions of the dining philosopher's problem require that a solution does not allow a philosopher to starve. For example, Silberschatz and Galvin state the following:

> Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Whether a particular monitor solution avoids starvation may depend on the implementation of the monitor. It may depend on the relative priority of the entry and waiting queues and it may depend on the queue semantics (FIFO, LIFO, random, etc.). Silberschatz and Galvin and Stallings draw pictures of monitors representing the queues as FIFO queues, but do not explicitly discuss this issue.

Most textbooks present the same monitor solution to the din-

ing philosophers problem. With this algorithm, a philosopher can starve if one of its neighbors is always eating. An example of this is shown in Section 6.1. Most standard textbooks ignore the question of starvation, although Stallings [9] does discuss this in an exercise. Hartley [3] gives several solutions to the dining philosophers problem that avoid starvation. One simple idea in avoiding starvation is to not allow a philosopher to eat twice while a neighboring philosopher remains hungry. Figure 2 gives pseudocode for one such solution, which I call the *Polite* algorithm. It is a modification of the standard monitor solution given in [8] using the starvation avoidance ideas from [3]. The modifications to the traditional solution are shown in boldface.

```
monitor diningPhilosophers {
   int[] state = new int[5];
   boolean[] leftHungry = new boolean[5];
   boolean[] rightHungry = new boolean[5];
   static final int THINKING = 0;
   static final int HUNGRY = 1;
   static final int EATING = 2;
   condition[] self = new condition[5];

   public diningPhilosophers {
      for (int i=0;i<5;i++) {
         state[i] = THINKING;
         leftHungry[i] = false;
         rightHungry[i] = false;
      }
   }

   public entry pickUp(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait;
      rightHungry(left(i)) = false;
      leftHungry(right(i)) = false;
   }

   public entry putDown(int i) {
      state[i] = THINKING;
      test(left(i));
      if (state[left(i)] == HUNGRY)
         leftHungry[i] = true;
      test(right(i));
      if (state[right(i)] == HUNGRY)
         rightHungry[i] = true;
   }

   private test(int i) {
      if  (state[right(i)] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[left(i)] != EATING)  &&
            !leftHungry(i) && !rightHungry(i) ) {
             state[i] = EATING;
             self[i].signal;
          }
   }

   private int left(int i) {
      return (i+1)%5;
   }

   private int right(int i) {
      return (i+4)%5;
   }

}
```

Figure 2: The *polite* monitor solution of the dining philosophers problem that avoids starvation.

In a theoretical context, starvation usually means indefinite blocking, meaning that there is not a bound to the time (or more precisely, the number of times other philosophers eat) between hungry and eating for a given philosopher. In a practical context, we would like the bound to be reasonably small. If there are five philosophers, the bound should be around 5, not 5000. The *Polite* algorithm avoids starvation in this sense. However, a price is paid for this in that some philosophers remain hungry longer than they might otherwise have. Another way of saying this is that the solution does not allow as much parallelism as the traditional one.

We take an experimental approach to the problem of starvation here. We give probability distributions for the eating and thinking times for each philosopher and look at the resulting times philosophers are hungry. We can consider starvation to occur when a philosopher is hungry for a given time, or we can compare different algorithms or monitor implementations by examining average hungry times. In Section 6.2 we compare the traditional algorithm, which allows starvation, to the polite algorithm which does not.

4 The Simulation Program

The simulation program is written in Java and can be run either locally as a Java application or remotely from a browser as a Java applet. In either case a window appears as shown in Figure 3. The window shows the philosophers around a round table. Each philosopher is shown as a box in a unique color with a letter inside indicating the state of the philosopher (Thinking, Hungry, or Eating).

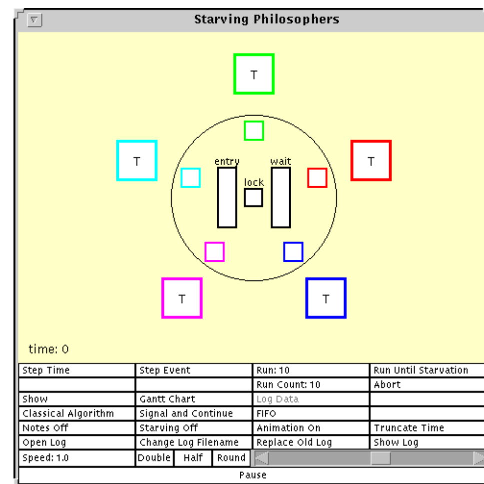

Figure 3: A view of the main simulator window.

The round table represents the monitor. Inside the table, near the edge is the condition variable queue, *self*, for each philosopher. It is represented as a small box of the same color as the philosopher. Each condition variable queue is either empty (filled with white) or has the corresponding philosopher waiting in the queue (filled with the color of that philosopher).

At the center of the table are the entry and waiting queues and the monitor lock. The monitor lock is represented by a small box. When a philosopher task owns the lock, the box

is filled with the color of that philosopher. The entry and waiting queues are represented by rectangles. They are filled with strips of color representing the philosopher tasks in the queue.

The simulator is event driven. It uses a virtual time that is an integer shown in the bottom left corner of the window. Each philosopher has a distribution for its thinking and eating times. Available distributions include constant, uniform, and exponential. Values from the exponential distribution are rounded down to an integer.

Three types of monitors are implemented, *Signal and Continue*, *Wait and Notify*, and *Entry Priority*. In the last of these the entry queue has a strictly higher priority than the waiting queue. As mentioned in Section 2 this type of monitor has undesirable properties, but tests show that several Java implementations may have this behavior. Each of these three types of monitors can have three disciplines for the entry and wait queues, FIFO (first-in/first-out), LIFO (last-in/first-out) and random. In the solution to the dining philosophers problem presented, the condition variable queues can each have at most one entry.

The simulator can be used in three ways.

In *Single-step Mode* the simulator will run one time step. After each step you can examine the state of each philosopher.

In *Multiple-step Mode* you can run for a given amount of time.

In *Run Until Starvation Mode* the simulator will run until at least one philosopher has starved.

In any of these modes the simulator can animate the process so you can see a philosopher task move from one queue to another as the philosopher changes from thinking to hungry to eating and back to thinking again.

5 Using the Simulator

The simulator can be used to illustrate different aspects of monitors and monitor solutions to the dining philosophers problem.

A class demonstration that uses the simulator in animation mode can illustrate the ideas of monitor implementation and illustrate monitor locks. It can illustrate how the tasks move from the condition variable queues to the waiting queue and how they compete for the monitor lock when they are in the entry and waiting queues. If access to the web is available in the classroom, simply accessing the simulator web page [15] brings up the simulator in a mode that is useful for classroom demonstration. By stepping through time, the simulator animates the movement of the processes among the various monitor queues in the case of starvation discussed in Section 6.1. The speed of the animation can be modified with a slider and the animation can be paused and resumed at any time to facilitate class discussion.

The simulator can be used to illustrate how to design experiments to test hypotheses or make comparisons. Once a set of characteristics of the philosophers has been chosen (number of philosophers, thinking and eating times, etc.), a comparison can be made between the traditional and polite algorithms or between queue priority schemes or queueing disciplines.

The simulator can be run until a philosopher starves and the time until starvation can be compared. Alternatively, the simulator can be run for a given number of time steps and average or maximum hungry times can be compared.

The simulator uses the Jeli logging facility [7] to store the results of the simulation in HTML format so that it can be viewed by a standard browser. The simulator can produce tables of data giving information about a given philosopher (such as average and maximum time in each state) and summary information. It can trace the states of any philosopher and can produce a Gantt chart showing the state of each philosopher as a function of time.

6 Sample Results

Several examples of running the simulator are discussed in this section.

6.1 Gantt Charts

Gantt charts for the simulation run for a case in which the traditional algorithm exhibits starvation are shown in Figure 4. The top chart displays the traditional algorithm, and the bottom one displays the polite algorithm. The two charts are the same until time 5. Philosophers 1 and 3 (P1 and P3) both become hungry at time 2. P1 starts eating at time 3 when P2 finishes eating. P2 becomes hungry again at time 4 and P1 finishes eating at time 5. In the polite algorithm, P2 noted that P3 was still hungry when P2 finished eating, so it does not start eating at time 5. This allows P3 to start eating at time 6 when P4 finishes eating. In the traditional algorithm, if one of P2 and P4 are always eating, P3 will starve.

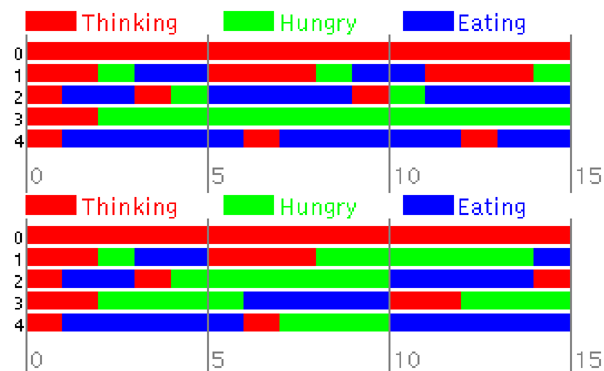

Figure 4: Gantt charts for the traditional (top) and polite (bottom) algorithms.

The simulator allows multiple runs to be displayed simultaneously for easy comparison. Two windows like that in Figure 3 appear simultaneously. The run buttons for these are

linked, so that pushing a run button on one causes both simulations to run in parallel. This is very useful for comparison in animation mode.

6.2 Consequences of Avoiding Starvation

The polite algorithm avoids starvation in the theoretical sense. No philosopher is postponed indefinitely, but there is a price to pay. While no more than four philosophers can eat before a given hungry philosopher eats, some philosophers may be delayed from eating, even when both chopsticks are free. This can increase the average time that a philosopher is hungry.

Consider the case in which the eating and thinking times times are exponentially distributed (rounded down to an integer) with an average of 10. The simulator was run using each algorithm for 100 time steps. Figure 5 shows two tables generated by the simulator, the top one for the traditional algorithm and the bottom one for the polite algorithm. For the polite algorithm the maximum hungry time is 16 compared to 23 for the traditional algorithm, but the average hungry time for the polite algorithm is larger by about 10 percent.

State	Count	Time	Fraction	Average Time	Min Time	Max Time	Standard Deviation
Thinking	22	238	.47600	10.81818	0	34	.46691
Eating	18	175	.35000	9.72222	0	28	.36090
Hungry	19	87	.17400	4.57895	0	23	.35709

State	Count	Time	Fraction	Average Time	Min Time	Max Time	Standard Deviation
Thinking	22	230	.46000	10.45455	0	34	.45573
Eating	18	172	.34400	9.55556	0	28	.35680
Hungry	19	98	.19600	5.15789	0	16	.28434

Figure 5: The upper table is for the traditional algorithm and the lower one is for the polite algorithm.

6.3 Monitor Implementation

The monitor implementation does not affect the dining philosophers problem very much since it is rare that the waiting and entry queues simultaneously are not empty, and when they are, the order in which they are emptied has no affect. This is because philosophers in the waiting queue leave the monitor immediately when they are awakened. However, the discipline of the queues can have an effect on the output. It is easiest to see this in the case of an even number of philosophers, P0 through P5. Suppose all philosophers become hungry at the same time and are put in the entry queue in numerical order. If they are removed in FIFO order, P0, P2, and P4 will eat. If this continues, half of the philosophers could be eating at any time.

However, if P0 is taken out followed by P3, only P0 and P3 will be allowed to eat and at least one of the philosophers will have to wait through two rounds of eating before he can eat.

7 Conclusions

The dining philosophers simulator can be used in several ways. It can be used to teach about monitors and monitor implementation options, it can be used to teach about the dining philosophers problem and it can be used as a testbed for teaching experimental techniques in computer science.

The web site [14] contains a number of resources for using the simulator. You can run the simulator directly from the web, storing the output files on our server so that they can be viewed or printed using a standard browser. All of the examples discussed here can be run from the web. The simulator can be downloaded from the web so that it can be run locally. A user's guide is also available on the web page.

8 Acknowledgments

This work has been supported by an NSF grant: *A Web-Based Electronic Laboratory for Operating Systems and Computer Networks*, DUE-9752165.

References

[1] Brinch Hansen, P., "Monitors and Concurrent Pascal: A personal history," *2nd ACM Conference on the History of Programming Languages*, 1993, pp. 1–25.

[2] Buhr, P. A. and Fortier, M., "Monitor classification," *ACM Computing Surveys*, **27**, (1), 1995, pp. 63–107.

[3] Hartley, S., *Concurrent Programming, The Java Programming Language*, Oxford, 1998.

[4] Howard, J. H., "Signalling in monitors," *Proceedings of the Second International Conference on Software Engineering*, 1976, pp. 47–52.

[5] Lampson, B. W., and Redell, D. D., "Experiences with processes and monitors in mesa," *Commun. ACM* **23**(2), 1980, pp.105–117.

[6] Nutt, G., *Operating Systems, A Modern Perspective*, 2nd Edition, Addison Wesley, 2000.

[7] Robbins, S., "Remote logging in Java using Jeli: A facility to enhance development of accessible educational software," *Proc. 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, pp. 114–118.

[8] Silberschatz, A. and Galvin, P. B., *Operating System Concepts*, 5th edition, Addison-Wesley, 1998.

[9] Stallings, W., *Operating Systems*, 2nd edition, Prentice Hall, 1995.

[10] Tanenbaum, A. S., *Modern Operating Systems*, Prentice Hall, 1982.

[11] Tichy, W. F., "Should computer scientists experiment more?" *IEEE Computer*, May 1998, pp. 32–40.

[12] Tichy, W. F., et al., "Experimental evaluation in computer science: A quantitative study," *J. Systems and Software,* Jan 1995, pp. 1–18.

[13] Zelkowitz, M. V. and Wallace, D. R., "Experimental models for validating technology," *IEEE Computer*, May 1998, pp. 23–31.

[14] **http://vip.cs.utsa.edu/nsf/**

[15] **http://vip.cs.utsa.edu/nsf/sf/**