

Experimentation with Bounded Buffer Synchronization

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

Abstract

Undergraduates are usually introduced to synchronization in operating systems through a discussion of classical problems such as reader-writer or producers-consumers. The traditional approach to teaching these topics is not effective in conveying to students how programs with incorrect synchronization actually behave. This paper introduces a simple probabilistic model for synchronization failure and shows how students can empirically study these issues. These activities are supported by a simulator that students can use to explore synchronization in the context of the bounded buffer problem. The simulator is written in Java and can be used either standalone or from a standard browser. Students can save the data and graphs generated by the simulator in a log file in HTML format.

1 Introduction

Synchronization plays a central role in modern operating systems and distributed computing. It is also one of the most difficult topics to teach, partly because it is abstract and partly because synchronization problems depend on rare events. Critical sections typically comprise a very small percentage of the code, even when the algorithms are parallel in nature. On a single CPU system, a problem occurs only when one process loses the CPU while executing a critical section (a rare event) and another process enters its critical section (a possibly rare event). Programs in which the synchronization is not done correctly can give correct results most of the time. Failures, when they do occur, are not repeatable and may occur at places seemingly unrelated to the synchronization.

Several articles have proposed new ways to introduce synchronization into an undergraduate operating systems

course [1, 2, 3, 6]. The approach discussed here relies on experimentation by the student. Often, students, as well as professionals, consider a program *working* when it gives the correct answers for a fixed number of test inputs. Students should understand that even a large number of tests is not sufficient for a program that relies on synchronization. While a proof of correctness is beyond the scope of most undergraduate courses, students must learn how to analyze synchronous code. This paper presents a simulation tool for experimenting with the bounded buffer problem, a classical synchronization problem described in many textbooks [7, 8, 9]. The tool allows students to determine the effect of ignoring synchronization issues in an empirical setting.

In the next section we discuss the role of experimentation in the computer science curriculum. The bounded buffer problem is then presented along with an analysis of a failure in the synchronization. Section 4 presents some sample questions and experiments that could be explored empirically through simulation. This is followed by a discussion of the simulation program and some sample results.

2 Experimentation in Computer Science

Experimentation is the centerpiece of the traditional scientific method. Experimental exploration can provide new insights, eliminate unproductive approaches and validate theories and methods. The 1991 curriculum report of the ACM [12] indicates that each area of the curriculum should employ the processes of theory, abstraction and design. It lists the following elements of abstraction:

- Data collection and hypothesis formation
- Modeling and prediction
- Design of an experiment
- Analysis of results

These elements are often lacking from both the undergraduate computer science curriculum and the traditional computer science research literature [10, 11, 13]. Experimentation needs greater emphasis in the computer science curriculum at all levels. The bounded buffer simulation described here provides a natural mechanism for incorporating experimentation into an undergraduate operating systems course.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2000 3/00 Austin, TX, USA
© 2000 ACM 1-58113-213-1/00/0003...\$5.00

3 The Producer-Consumer Problem

One of the simplest synchronization examples is the *bounded buffer problem*, also known as the *producer-consumer problem*. Often, this example is used to introduce the importance of synchronization because of its wide applicability. Producer processes produce items that are stored in a shared circular queue, while consumer processes remove these items and process them. Synchronization is needed to guarantee that producers do not insert items into the queue when the queue is full, that consumers do not try to remove items when the queue is empty, and that each produced item is consumed by exactly one consumer.

Unsynchronized versions of the producer and consumer loops are shown below. The bounded buffer is implemented as a circular queue of size n . The number of items stored in the queue is kept in `counter` while `in` and `out` point to the next empty slot and the last full slot, respectively.

Producer:

```
nextp = produce();
while (counter == n) ;
buffer[in] = nextp;
in = (in + 1)%n;
counter++;
```

Consumer:

```
while (counter == 0) ;
nextc = buffer[out];
out = (out + 1)%n;
counter--;
consume(nextc);
```

When there is only one producer and one consumer on a single CPU system, the code is correct as long as the incrementing and decrementing of the counter is atomic. However, on most RISC machines `counter++` is implemented with three assembly language instructions:

```
R1 = counter
R1 = R1 + 1
counter = R1
```

If the producer loses the CPU after the first or second of these instructions and the consumer decrements the counter, the counter will end up with an incorrect value. If the consumer decrements the counter exactly once before the producer runs again, the counter value will be one more than it should be. This case can result in an item being consumed more than once or an uninitialized item being consumed. Alternatively, if the consumer loses the CPU in the middle of decrementing the counter, the counter can end up with a value that is too small and a produced item may be overwritten before it can be consumed.

An *internal inconsistency* occurs when the counter value does not agree with the number of items inserted but not removed from the buffer. At this point the program does not correctly represent the actions taken by the producer and the consumer.

The program may run for an arbitrarily long time after an internal inconsistency before it actually loses an item or consumes one twice. In fact, it may be possible for the program to correct itself before any actual harm is done. An *external inconsistency* occurs when the program consumes an incorrect item or consumes items out of order (indicating a lost item).

The distinction between internal and external inconsistencies is an important one that is often overlooked in the discussion of synchronization. In quantum physics, a distinction is made between the state of a system, represented by a wave function, and the observables of the system, the things that can be measured. The internal state of a bounded buffer implementation is not directly observable. To make it observable, the code needs to be instrumented or run under a debugger. Since failure depends on timing, the instrumentation, that is the act of observing the internal state, changes the probability of failure. This is analogous to the Heisenberg uncertainty principle. For this reason, simulation or analytic analysis is necessary.

3.1 A Simple Analysis

How likely is failure in the unsynchronized bounded buffer problem? Failure questions are difficult to answer in general, but it is possible to make a few simplifying assumptions to help gain insight. Consider the case of estimating the probability of internal failure in the bounded buffer problem with one producer and one consumer. Suppose that the producer loop takes k cycles and that each of the three assembly language instructions to increment the counter takes one cycle to execute. The probability that when it loses the CPU, the producer is in this critical section is just $2/k$. This alone does not guarantee that the program will eventually have an internal inconsistency. The consumer still needs to execute its critical section. Suppose that the consumer is always ready and that the priorities are such that it will get the CPU before the producer runs again. In this case the consumer is guaranteed to execute its critical section if the buffer is not empty and the quantum is longer than the consumer loop. If on the other hand, the quantum, q , is shorter than the loop, the probability of the consumer reaching its critical section is just q/k .

Simple probability arguments can be used to show that if an event has probability p of occurring in an interval, the average number of intervals before the event occurs is $1/p$. Thus, in the first case (large quantum) the average number of times the producer is in the CPU before failure is $k/2$, and the average number of cycles (by the producer) until failure is $qk/2$.

In the second case (small quantum), the probability of failure is $2/k \times q/k$, or $2q/k^2$. The average number of CPU bursts until failure is $k^2/2q$ and the average number of cycles until failure is $k^2/2$.

Note that the two cases agree when $q = k$.

3.2 Limitations of the Simple Analysis

This simple analysis, which is suitable for discussion in an undergraduate course, has a number of limitations. The following assumptions have been made.

- 1) The producer and consumer never block on the buffer full or empty, or at least this is a rare occurrence.
- 2) No I/O or other systems calls cause the process to lose the CPU prematurely.
- 3) The program is running on a single CPU machine that is shared by a single producer and single consumer. Otherwise there are additional critical sections.
- 4) The number of cycles in the producer and consumer loops are about same. This is not a necessary assumption, but it makes the analysis simpler.
- 5) The scheduling algorithm is such that when one process loses the CPU, the other one will run before the first one gets in again.
- 6) No pattern is established so that, for example, the producer does not always lose the CPU at exactly the same line of code. This could happen if the quantum (measured in cycles) and the number of cycles in the loop are related in some fixed way. We assume that the number of cycles in the produce and consume procedures are uniformly distributed in some fixed interval.

Many questions cannot be answered with this simple analysis and are better addressed empirically.

4 Sample Questions and Experiments

Here are some questions that can be posed to operating systems students to acquaint them with the problem in preparation for using the simulation.

- 1) Construct a sequence of events that would cause the counter to have a value one greater than it should. This is an internal inconsistency. After this internal inconsistency occurs, what type of external inconsistency could occur?
- 2) Construct a sequence of events that would cause the counter to have a value one less than it should. What type of external inconsistency would occur after this internal inconsistency?
- 3) Assume that the loops of the producer and consumer take exactly k cycles each and that a quantum of one cycle is used. (In this scenario the processes alternate execution.) How many cycles would it be before an internal inconsistency occurs?

Note: In this case an inconsistency might not occur at all. If the `produce()` and `consume()` take a large amount of time compared to the other instructions, the consumer would block waiting for the producer to produce one item and after that the producer would always be producing while the consumer is in its critical section. For this reason, it is useful to assume that the time

for producing and consuming is not constant, but has some known average value.

- 4) Give an example in which the instructions each take a constant amount of time and no inconsistency will occur.
- 5) Explain how it is possible that an internal inconsistency could occur, but it would not lead to an external inconsistency.

Here are some ideas for experiments that could be run if a suitable simulator were available.

Experiment 1: Run the simulator for a certain number of cycles with a given configuration. Determine the probability that an internal failure will occur. Determine the probability that an external failure will occur. Make several runs and see what happens. You only need to keep track of the fraction of the runs in which a failure occurs.

Experiment 2: For a given configuration run until internal inconsistency. Plot the distribution of the number of cycles until failure. Compare your results with the simple analysis from Section 3.1. If the results differ, explain why this may be. For the same configuration, determine the average number of cycles before external inconsistency. Compare these.

Experiment 3: The simple analysis predicts that when the quantum is large, the average number of cycles until internal inconsistency is proportional to the square of the number of cycles in the loop. Design and perform experiments to test this hypothesis. Perform the same experiments with external inconsistency.

Experiment 4: Determine the effect of the quantum on the average number of cycles until an internal failure occurs. The simple analysis predicts that there is a range of values for which the dependence is linear. Is this supported by the simulation?

The bounded buffer simulator described below allows students to perform each of the experiments described above and collects the data necessary for analysis of these experiments in a log file.

5 The Simulation Program

The main window of the simulation program is shown in Figure 1. The buffer and the shared variables are shown at the top, followed by the producer and consumer code with the number of cycles required for each line. Also displayed are the number of times each line was interrupted by a quantum expiration and the number of times it was executed. Below the code is a panel of buttons for controlling the simulator and an area for displaying information about the progress of the simulation.

The code for the producer and consumer is hard-coded into the program. In this implementation a process executes a `yield` instruction when it blocks. This just puts the process

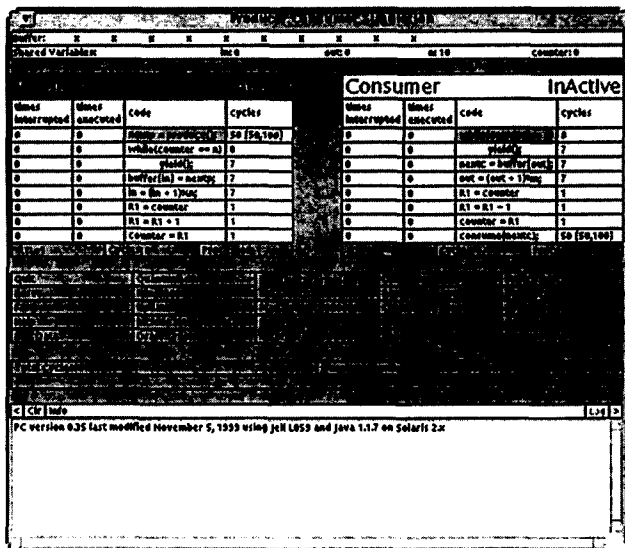


Figure 1: A view of the main simulator window.

back in the ready queue. The quantum and the number of cycles required for each line of code are set in a configuration file, allowing students to experiment with the effect of changing these parameters. The simulator can be run in several modes depending on the level of detail required.

At the lowest level, the user can step through the code one cycle at a time, manually switch active processes, or run a fixed (user settable) number of cycles allowing the quantum to determine which process is running.

At the next level of detail, the user can perform a single run of the program. The run can be set to terminate after a certain number of cycles, when the protocol fails, or when either the protocol fails or a certain number of cycles have been executed. The failure test can be either in terms of internal inconsistency or external inconsistency.

Lastly, multiple runs can be set up and statistics gathered for the runs.

The simulator creates tables and graphs of statistics in a log file in HTML format so that they can be viewed from a standard browser. The graphs are stored as GIF images. This facility allows students to easily produce a log file of an experiment that can be printed out or viewed on line. A sample log file is shown in Figure 2. The result of one experiment is shown. In this experiment, 100 runs are done, each stopping when an external failure occurs. The simulator keeps track of when the first internal failure of each run occurs. The average number of cycles until internal failure is 7797.23, while the average for external failure is 147,934.19 cycles. A histogram comparing the number of cycles until internal or external failure is shown at the end of the log file.

The simulator is written in Java and uses the Jeli [16] package and both were developed as part of an NSF grant [14].

Consumer/Producer Simulation Log for Steven Robbins generated Wed Nov 17 13:43:27 CST 1999

PC version 0.35 last modified November 5, 1999 using jeli L059 and java 1.1.7 on Solaris 2x
Java version 1.1.7 OS Solaris version 2x 1.063963847

Producer		Consumer	
code	cycles	code	cycles
nextp = produce();	[50,100]	while(counter == 0)	8
while(counter == n)	8	yield();	7
yield();	7	nextc = buffer[out];	7
buffer[in] = nextp;	7	out = (out + 1) % n;	7
in = (in + 1) % n;	7	RI = counter;	1
RI = counter;	1	RI = RI - 1;	1
RI = RI + 1;	1	counter = RI;	1
counter = RI;	1	consume(nextc);	[50,100]

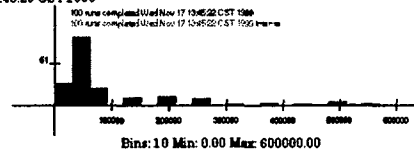
Wed Nov 17 13:43:30 CST 1999 Starting 100 runs until external failure
Wed Nov 17 13:45:22 CST 1999 Runs Completed

Producer				Consumer			
interrupted	executed	code	cycles	interrupted	executed	code	cycles
135784	74098	nextp = produce();	[50,100]	14797	75454	while(counter == 0)	8
14323	74523	while(counter == n)	8	431	1743	yield();	7
131	449	yield();	7	12311	73711	nextc = buffer[out];	7
12406	74051	buffer[in] = nextp;	7	12093	73611	out = (out + 1) % n;	7
12592	74032	in = (in + 1) % n;	7	1698	73611	RI = counter;	1
1821	74032	RI = counter;	1	1701	73611	RI = RI - 1;	1
1713	74030	RI = RI + 1;	1	1698	73611	counter = RI;	1
1786	74028	counter = RI;	1	134492	73611	consume(nextc);	[50,100]
Cycles: 7410872 Processed: 74028				Cycles: 7362547 Processed: 73611			
Runs: 100 Buffer Size: 10				Quantum: 40 Total Cycles: 14793419			
Average Number of Cycles: 14793419							

Cycles	Total	Number	Average	Minimum	Maximum	SD
Total	14,793,419	100	147,934.19	1,066	853,210	1,36,536.46
Producer	7,410,872	100	74,108.72	872	276,664	68,270.68
Consumer	7,382,547	100	73,825.47	614	276,546	68,266.06
Internal Failure	779,723	100	7,797.23	645	25,491	6,726.12
Internal Producer Failure	390,885	50	7,817.70	686	23,814	6,007.57
Internal Consumer Failure	388,838	50	7,776.76	645	25,491	6,430.03
External Failure	14,793,419	100	147,934.19	1,066	853,210	1,36,536.46
External Skip Failure	6,637,814	40	165,945.35	96,733	424,305	93,791.50
External Repeat Failure	8,155,605	60	135,926.75	1,066	853,210	157,624.02

Failure	Number	Percentage
Internal	100	100.00
Internal in Producer	50	50.00
Internal in Consumer	50	50.00
External	100	100.00
External Skip Item	40	40.00
External Repeat Item	60	60.00

Wed Nov 17 13:45:29 CST 1999



Wed Nov 17 13:46:06 CST 1999 Close Log

Figure 2: A portion of a log file as seen from a browser.

This simulator is one of a number of interactive simulations developed as part of this grant for use in the computer science curriculum [4]. The simulator can be downloaded and run on any machine supporting Java. This method of execution allows users to set their own configuration files and store log files locally. Alternatively, the simulator can be run from a standard browser by just giving it the appropriate remote URL. The Jeli package allows applets run from a browser to

