A Study of Common Pitfalls in Simple Multi-Threaded Programs

Sung-Eun Choi[*]
Los Alamos National Laboratory
Advanced Computing Laboratory
P.O. Box 1663, MS B287
Los Alamos, NM 87545 USA

E Christopher Lewis[†]
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA

Abstract

It is generally acknowledged that developing correct multi-threaded codes is difficult, because threads may interact with each other in unpredictable ways. The goal of this work is to discover common multi-threaded programming pitfalls, the knowledge of which will be useful in instructing new programmers and in developing tools to aid in multi-threaded programming. To this end, we study multi-threaded applications written by students from introductory operating systems courses. Although the applications are simple, careful inspection and the use of an automatic race detection tool reveal a surprising quantity and variety of synchronization errors. We describe and discuss these errors, evaluate the role of automated tools, and propose new tools for use in the instruction of multi-threaded programming.

1 Introduction

Multi-threading is a powerful programming paradigm, useful in many problem domains. It is a convenient structuring tool for applications that are logically comprised of asynchronous components, such as windowing applications and operating system services. Multi-threading is also appropriate for expressing fine grain sharing such as that arising from data parallel computations in which threads simultaneously perform nearly the same computation on different data. In fact, direct support for very fine grain multi-threading has even been implemented in hardware [1].

Although multi-threading provides a conceptually simple abstraction, in practice, multi-threaded *programming* is challenging, because threads interact with each other in unpredictable ways. Multiple threads usually share data, requiring synchronization to manage their interaction. Synchronization must ensure a deterministic outcome independent of how threads are scheduled or how their instruction streams are interleaved.

This paper describes experiences from analyzing a collection of simple multi-threaded programs. We evaluate 180 programs written by students who were new to multi-threaded programming. We describe the common errors and discuss their origins. We use this catalog of errors to outline principles to abide by when writing multi-threaded programs. Finally, we discuss our experiences with Eraser [6], an automatic, dynamic race detection tool, and the potential for other useful debugging tools.

The results discussed in this paper will be of use to educators, program developers, and tool developers. Educators can teach common pitfalls and instill good habits for multi-threaded programming. Program developers can become aware of common errors and the potential causes. Tool developers can tailor their tools to the types of errors that occur in practice and develop new tools for the types of errors we have found by manual inspection.

This paper is organized as follows. The next section describes our experimental context. Section 3 presents the results of analyzing the multi-threaded applications and the role of Eraser in finding errors in the programs. Section 4 describes the potential use of tools based on our experience with Eraser. The final section gives conclusions.

2 Experimental Context

In this section we describe the multi-threaded program suite used in this study, and we summarize the process by which they were evaluated by inspection and via the Eraser tool.

[†] E Lewis was supported in part by a Bradley Dissertation Fellowship.

From *Proceedings of the Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, March 2000. Austin, TX USA.

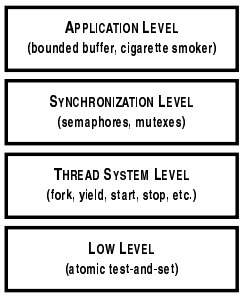

Figure 1: Structure of the CSE 451 threads project. Each layer builds on the layers below it. We considered codes the students wrote for the application and synchronization layers.

2.1 Program Suite

The evaluation program suite comes from student programming projects from three offerings of the University of Washington introductory operating systems course (CSE 451) [2]. Over three quarters the course was taught by two different instructors with four different teaching assistants (including the authors), but an identical programming project was assigned. The project assignments were mature and well organized, and the students implemented their projects in the C programming language.

The students were charged with writing a small user-level threads system and applications to exploit it. The project familiarized students with operating systems issues; and for most, it was their first encounter with threads, synchronization mechanisms and, often times, serious programming. Building upon an atomic test-and-set routine, they implemented support for thread creation, scheduling, *etc*., and they provided synchronization support via semaphores and mutexes with condition variables. In addition, they implemented two multi-threaded applications: a bounded buffer application [7, page 109] and a solution to the cigarette smoker problem [7, page 212]. Figure 1 summarizes the project's principle abstractions. Although the students implemented the top three levels, we only examined the *application* and *synchronization* portions of the project in this study.

2.2 Inspection

Virtually all of the programs we examined produced correct output, but this alone did not ensure that they were correct. The students' programs were sufficiently simple and short that careful inspect by an experienced programmer revealed the synchronization errors they contained. Both authors have taken and served as teaching assistant for the introductory operating systems course, so they are intimately familiar with the project and multi-threaded programming in general. Each program was independently examined twice to identify errors.

2.3 Eraser

We used Eraser to automatically evaluate the programs. Eraser is a tool for finding data races in Pthreads multi-threaded programs that synchronize via locking [6]. It dynamically checks that concurrent accesses to shared data observe a consistent locking discipline. Eraser implements the *lock set* algorithm to identify races [6]. This algorithm dynamically maintains a candidate set of locks for each potentially shared word. All locks that *may* protect a word appear in the word's candidate lock set. Initially, each lock set contains all locks. On every access to shared data, the lock set associated with the data is replaced by the intersection of itself and the locks currently held. If the candidate lock set ever becomes empty, a race is flagged, for the associated data is unprotected. Eraser supports a set of programmer annotations to inhibit false positives when using non-Pthreads synchronization mechanisms. Eraser instruments a program binary's loads, stores and Pthreads library calls using ATOM [8], a binary rewriting tool for the DEC Alpha.

Eraser could not be directly applied to students' programs, because they were not Pthreads applications. We transformed them into Pthreads applications by rewriting the abstractions on which they were built in terms of Pthreads primitives. For example, in order to evaluate the *application* level programs with Eraser, we linked them with semaphore and mutex codes implemented in terms of Pthreads primitives. We only evaluated the *application* and *synchronization* level codes, because the thread system code achieved atomicity via interrupt disabling, which Eraser is not equipped to handle.

3 Pitfalls

We discovered a great many synchronization errors in the programs we examined. In this section, we enumerate and discuss the errors and our experience with Eraser.

3.1 Errors

The errors we detected fall into one of three categories. They are data races, deadlock or miscellaneous, each discussed below. We examined four applications (semaphore, mutex, bounded buffer, and cigarette smoker) from each of 54 project groups (because of submission or interface problems we only considered 180 applications, not the total 216). Figure 2 summarizes our findings. We discovered errors in 56 applications. Twenty-three contained data races, 23 deadlocked and 28 exhibited miscellaneous synchronization problems.

Data Races. A *data race condition* exists when multiple entities (in this case threads) concurrently read and write the same data, and the outcome of the execution depends on the particular order in which the accesses take place [7, page 165]. The entities must synchronize to avoid race conditions.

	programs w/ errors	*data race*	*dead-lock*	*misc.*	*Eraser false +*
semaphore	19 of 46	2	7	19	3
mutex	18 of 40	5	12	8	0
bounded buf.	5 of 46	3	1	0	0
cig. smoker	14 of 48	13	3	1	1
total	56 of 180	23	23	28	4

Figure 2: Summary of errors. A total of 180 programs were examined. Some programs had more than one type of error. Multiple instances of a single type of error in a single application were counted once.

We discovered 23 data races, most of which were benign.

Nine of the data races we discovered were benign, because reads and writes of integers were atomic on the students' hardware platform. Some students did not use locks when reading shared integers. Furthermore, some did not protect writes of shared integers when there was only a single writer. We do not recommend that beginning multi-threaded programmers make such atomicity assumptions.

The remaining data races effected correctness. *(i)* The most obvious source of a data race was when a *shared variable was not protected* by a lock. It appeared that some students were confused about what data was shared. Nine programs suffered from this problem. *(ii)* Similarly, data races arose when a *lock was not acquired* to protect an access even though one existed. Most frequently this happened when a lock was acquired outside a loop, but released within. Three programs contained this error. *(iii)* Data races also arose from *accidental sharing*: one group made what should have been an automatic variable (*i.e.*, on the stack) global, thus it was unintentionally shared by all threads without being protected by a lock. *(iv)* One program *prematurely released a lock*, suggesting that the student was not entirely certain of what accesses needed to be protected by the lock. *(v)* Another program contained *redundant, unprotected initialization* of shared variables. *(vi)* One program used *multiple locks* to protect a single shared variable, but only one at a time. Specifically, readers and writers of a shared variable used different locks, thus there was no synchronization between the two classes of threads.

Deadlock. *Deadlock* occurs when a multi-threaded program is unable to make progress because a thread is waiting for a condition that will never happen. Twenty-three of the programs suffered from intermittent deadlock.

Deadlock arose in 11 programs because they contained a *non-atomic unlock-and-stop* routine. This routine releases a lock, places the running thread on a queue, and blocks. If the thread is preempted between the second and third steps, another thread may reschedule the first thread before it blocks. In this case, the first thread runs and immediately blocks, never to be awakened. Deadlock also occurred in two programs where *signals were lost*. This happened when one thread signaled a thread that had not yet issued a wait. When the wait is eventually issued, it is never signalled again. Both of these deadlock cases suggest a lack of understanding of some of the more subtle issues of multi-threading and the implications of preemption.

Deadlock also arose in two programs when they used an incorrect conditional to spin on a test-and-set lock. Deadlock resulted in eight programs for one of the following reasons: *(i)* unrelated locks were used to protect a single shared variable, *(ii)* test-and-set primitives were confused with mutexes at the application level, *(iii)* locks were not ever released, and *(iv)* threads tried to reacquire locks that they already held.

A frequently cited example of deadlock is when there is a circular dependence between threads waiting for locks [3, 5]. None of the programs exhibited this form of deadlock, except the trivial case when a thread attempted to acquire a lock it already held. We expect that circular dependences are more common in larger applications, but they are not a primary concern for beginning multi-threaded programmers.

Miscellaneous. There were a number of miscellaneous synchronization problems. Unnecessary use of interrupt disabling *and* lock acquisition and release was the most common error in this category. Fourteen programs suffered from this problem, suggesting that students did not have confidence in their use of locks, so they tried to patch up the code (and make it correct) by disabling interrupts. It also suggests an unsophisticated understanding of locking. This approach never corrected any problems and was even found in applications that were otherwise correct.

Another problem was the use of a single global lock to protect all shared data. Four programs contained this problem. We suspect that this comes from more than just laziness; some students did not understand that only related data should be protected under one lock. This is often a benign problem, but it can lead to lock contention. The remaining 10 errors were due to inappropriate locking: *(i)* assuming that the unlock-and-stop routine returns with the test-and-set lock re-acquired and *(ii)* needlessly unlocking the same lock repeatedly.

3.2 Experience with Eraser

When Eraser was applied to the program suite, it detected 27 races, 23 of which we have already discussed, above. The remaining four were false positives. Three programs implicitly "pass" a lock from one thread to another. In other words, the thread that acquires the lock does not release it with the understanding that another thread will proceed assuming that it has the lock. The second thread may release the lock or pass it to another thread. Birrell recommends that lock passing never be used due to the difficulty of verifying the correctness of such code [3]. On the other

hand, lock passing reduces the number of calls into the locking routines, which indirectly reduces the amount of work the thread scheduler performs. Moreover, if lock passing is used exclusively, an application can affectively take control of thread scheduling. Obviously this is not a synchronization mechanism for the naive, but if Eraser understands such a synchronization mechanism, it may prove useful for more advanced programmers who are concerned about performance. A straightforward use of the `EraserWriteLock` and `EraserWriteUnlock` annotations will tell the Eraser runtime system the point at which a thread no longer assumes ownership of the lock (effectively releasing it) as well as the dual for the thread that receives the lock (effectively acquiring the lock).

A false race was also reported when threads accessed a circular buffer through protected "front" and "back" pointers into the buffer. This is not a race because the buffer is only accessed through the "front" or "back" pointers. Savage *et al*. describe a similar false positive, which they eliminate via the `EraserReuse` annotation [6]. We can do the same.

An unintended side effect of running an Eraser instrumented binary is that the instrumentation changes the timing characteristics of the program. This alone exposed deadlock in a great many programs that consistently ran to completion without instrumentation.

3.3 Discussion

Eraser was enormously useful in this study. Even in these simple codes errors were common and tedious to find manually. The students generally believed their codes were correct, and the teaching assistants often did not find the errors, because there was no feedback suggesting that an error existed. Eraser's primary value was in discovering races even when a particular thread scheduling did not reveal a data race. Despite the simplicity of these programs, we believe a significant fraction of the races came from carelessness, which Eraser was well-equipped to uncover. Eraser's value in finding these errors grows with program complexity.

Programmers may help detect other errors through the use of assertions. For example, if the programmers assert that a lock is held before it is released, they will discover cases where a lock is being released repeatedly. Many of the other errors came from a lack of understand of the issues of multi-threaded programming. Eraser is still valuable, for it forces programmers to understand their programs well enough to interpret Eraser's finding and correct their errors. Without Eraser students are more likely to cross their fingers and hope their program works.

We recommend that new programmers be aware of the roles of correctness and efficiency in writing multi-threaded programs. Birrell advises first developing a correct solution which may later be optimized, rather than developing an optimized solution which may later be made correct [3]. Once correctness is achieved, refinement can begin if performance studies suggest that it is necessary. Performance studies should be used throughout the refinement process to validate the refinement's value. A programmer should not obfuscate the code without benefit. We recommend instructors preach this philosophy, for even the more advanced students frequently got burned by their own cleverness. We also recommend that instructors *teach* mistakes, including the ones presented in this paper. Teaching students the common mistakes can drastically shrink the space of potential errors.

4 Potential for Tools

Eraser was useful for analyzing the applications in the previous section, but it is limited to detecting data races in codes that synchronize with locks. In this section, we propose other useful tools and discuss the current status of existing tools for debugging multi-threaded applications.

4.1 Other Useful Tools

This section propose three other useful tools for multi-threaded program debugging.

Supporting other synchronization. Eraser's generality is limited by the fact that it only considers synchronization by locking. Multi-threaded parallel scientific codes often use more global forms of synchronization such as *barrier synchronization*, where all threads rendezvous at a specified program point, and restricted *fork-join*, where a single thread forks many threads that run to completion after which control is returned to a single thread. An important observation in both cases is that accesses to shared data in one context (between successive barriers or after forking threads but before joining) do not conflict with accesses in another context. Eraser can be extended to handle such synchronization by resetting the global state (lock sets and data states) at each barrier or fork. One implication of this approach is that shared data may be protected by different locks in different contexts. This may have stylistic problems, but it will still prevent races.

Deadlock detection. A common cause of deadlock in complex applications is a circular dependence of threads waiting for resources. Deadlock can often be *detected* by the programmer because the program ceases to make progress. Nevertheless, finding the cause of deadlock is difficult. A tool to aid in deadlock detection caused by circular dependences can be implemented using a *waits-for* graph of threads waiting for locks; a cycle in the graph indicates that deadlock has occurred. Deadlock conditions caused by thread scheduling errors can be detected when no threads are available on the thread system's ready queue; when this situation is detected, the condition variables can be scanned to identify waiting threads. Notice that unlike with races, these approaches are not guaranteed to detect all potential dead-

lock situations. Changing a program's timing characteristics or forcing frequent context switches can increase the probability of exposing deadlock situations.

Performance debugging. Performance debugging in multi-threaded programs is perhaps as difficult as debugging for correctness. Sharing is the primary source of unexpected performance degradation in multi-threaded programs. Statistics such as the average number of threads waiting for a lock and the number of times a lock is acquired can be used to identify critical sections of code that should be optimized. On machines with multiple processors that implement shared memory, data locality also contributes significantly to poor performance. Information about the physical location of the threads that use a lock and the shared data accessed can also be used to pinpoint performance bottlenecks.

4.2 Current Status of Tools

Visual Threads [4] is a diagnostic tool for multi-threaded applications that implements Eraser's race detection algorithm, as well as deadlock detection and performance statistics. The tool includes many of the feature discussed above, yet it is not a viable solution for the classroom. Visual Threads is only available for systems running Digital UNIX on Alpha processors, considered a high-end system for most consumers especially educational institutions.

The difficulty with making such tools widely available is that they typically require binary instrumentation which is architecture specific by nature. A slightly less general approach is to provide a library of data types and synchronization mechanisms that have already been instrumented. For example, the SMARTS library [9], a C++ class library for parallel programming, implements Eraser's race detection algorithm for its data objects. Shared data objects inherit from the base class which provides explicit read and write methods that modify the lock set if race detection is enabled. The library also provides synchronization mechanisms that are aware of the race detector. This approach is very attractive for the classroom setting because it is portable, inexpensive, and reusable. Moreover, most project courses such as those described in the previous sections already have a software base, and thus the cost of implementing the tools is amortized across subsequent offerings of the course.

5 Conclusion

Multi-threading is a popular programming paradigm used in a variety of domains. The catalog of common errors presented in this paper shows that new programmers make synchronization errors even in programs with very simple specifications. The students often lacked the experience and the teaching assistants lacked the time to identify subtle synchronization errors in programs that only failed under certain rare thread schedules. In fact, most programmers of erroneous code believed it was correct, for they did not get feedback suggesting otherwise. It is our hope that the pitfalls presented in this paper will serve as a guide so that students may avoid these mistakes. Furthermore, we encourage instructors to advocate optimization only when correctness has been achieved and only when performance studies demonstrate significant improvement.

Finally, we have found that automated tools, such as the Eraser dynamic race detector, can be a valuable resources in debugging multi-threaded programs. Eraser gives feedback for erroneous programs even if a particular thread schedule does not reveal it. This helps programmers identify their synchronization bugs. Furthermore, it serves as a starting point for reasoning about and correcting erroneous code. We believe that there are opportunities for the development of other automated tools—such as those discussed in this paper—to assist in multi-threaded programming, particularly in the classroom.

Acknowledgments. We thank Stefan Savage for his Eraser savvy and Brian Bershad for the inspiration.

References

[1] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. The Tera computer system. In *Proceedings of the 1990 ACM SIGARCH International Conference on Supercomputing (ICS '90)* (June 1990), pp. 1–6.

[2] Bershad, B., and Levy, H. M. CSE 451: Introduction to operating systems. University of Washington Department of Computer Science and Engineering. Spring 1996, Autumn 1996, and Winter 1997. *http://www.cs.washington.edu/education/courses/451/.*

[3] Birrell, A. D. An introduction to programming with threads. Tech. Rep. 35, Digital Equipment Corporation, Systems Research Center, January 1989.

[4] Compaq Computer Corporation. Visual threads home page. *http://www.unix.digital.com/visualthreads/.*

[5] Ousterhout, J. K. Why threads are a bad idea (for most purposes). Invited talk at 1996 USENIX Conference, Jan. 1996. *http://www.scriptics.com/people/john.ousterhout/threads.ps.*

[6] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multi-threaded programs. *Transactions on Computer Systems 15*, 4 (November 1998), 391–411.

[7] Silberschatz, A., and Galvin, P. *Operating Systems Concepts, Fourth Edition.* Addison-Wesley, 1994.

[8] Srivastava, A., and Eustace, A. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)* (1994), pp. 196–205.

[9] Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shende, S., Oldehoeft, R., and Smith, S. SMARTS: Exploiting temporal locality and parallelism through vertical execution. In *Proceedings of the 1999 ACM SIGARCH International Conference on Supercomputing (ICS '99)* (June 1999), pp. 302–310.