CMPSCI 691ST Systems **Fall 2011**

Lecture 18

Lecturer: Emery Berger *Scribe: Sean Barker*

18.1 Distributed and Concurrent Systems

In this lecture we discussed two broad issues in concurrent systems: clocks and race detection. The first paper we read introduced the Lamport clock, which is a scalar providing a logical clock value for each process. Every time a process sends a message, it increments its clock value, and every time a process receives a message, it sets its clock value to the max of the current value and the message value. Clock values are used to provide a 'happens-before' relationship, which provides a consistent partial ordering of events across processes.

Lamport clocks were later extended to vector clocks, in which each of n processes maintains an array (or vector) of n clock values (one for each process) rather than just a single value as in Lamport clocks. Vector clocks are much more precise than Lamport clocks, since they can check whether intermediate events from anyone have occurred (by looking at the max across the entire vector), and have essentially replaced Lamport clocks today. Moreover, since distributed systems are already sending many messages, the cost of maintaining vector clocks is generally not a big deal. Clock synchronization is still a hard problem, however, because physical clock values will always have some amount of drift.

The FastTrack paper we read leverages vector clocks to do race detection.

18.2 Race Detection

Broadly speaking, a race condition occurs when at least two threads are accessing a variable, and one of them writes the value while another is reading. In such a case, the result of the operation will vary depending on the arbitrary ordering of threads. Race detection refers to the problem of detecting when race conditions are possible in a program.

Race detection is complicated by the fact that races may be either problematic or benign – in some cases, the values that are read in the presence of a race condition may not actually matter that much, in which case the race condition is not really a problem. A classic example of this is a counter, for which missing a couple of updates is probably a non-issue and better that introducing performance-draining synchronization to eliminate the race condition.

Modern hardware typically offers atomic operations on word or double-world length data (e.g., compare and set operations), but these are not very fast in practice and so are not a panacea for these types of problems. Cache coherency is another issue with concurrent data access – when data is modified, all other copies in cache have to be invalidated and then refetched from RAM to stay in sync. This makes the system bus a central point of contention, even without regard to locking. To deal with this problem a distributed update protocol is needed that delays updates – e.g., pooling updates to a counter before applying the new value.

Digression: Bad Concurrency Practices

We discussed a few bad practices often seen that are relevant to concurrency:

1. **Custom memory allocators**. Writing a custom memory allocator may seem attractive versus using a standard allocator like `malloc`, but most of the time this doesn't actually provide any improved performance. Moreover, it prevents the usage of standard tools like Valgrind, which only understand malloc and will otherwise just see a giant chunk of memory as a program object.
2. **Benign races**. Some data races may seem to be benign, but actually have subtle effects that are not. Leaving known races unattended to can be dangerous.
3. **Ad-hoc synchronization**. Using 'clever' solutions to deal with synchronization (e.g., spinning on own value until it changes versus using a standard locking or monitor-based approach) usually turns out to be incorrect, inefficient, or both.

18.3 Race Detectors

Assuming none of the problems mentioned above, a **static race detector** (SRD) is designed to operate in a compiler-like manner – you feed a program through the SRD, which reports something like 'a race condition is possible on line 43'. SRDs are **sound**, meaning they will never miss any possible data races, but may report many false possibles. The problem is that it is impossible to check all possible program paths, and so many races may be reported where none are actually possible. Due to this, SRDs are not widely used in practice.

In contrast, a **dynamic race detector** (DRD) only sees one program path, and thus may report no data races simply because none were observed on that path. However, some DRDs can predict data races on other execution paths. This is the more common type of race detector.

18.3.1 FastTrack

FastTrack optimizes for the common sharing case where there isn't a lot of data sharing occurring. In these cases, sharing can be tracked using *epochs*, which are just single numbers. In contrast, many race detectors, including the original one in the 80's, are vector-clock based, which involve very high overhead. This overhead may also distort traces by slowing down execution of the program. FastTrack avoids the overhead of vector clocks when possible, and reports no false positives in general (assuming no ad-hoc synchronization, etc). When required, FastTrack falls back on using complete vector clocks, but this is rarely needed in practice.

18.3.2 Lock-set Detectors

The alternative to vector-clock based detectors is lock-set based detectors, such as the Eraser system. These types of detectors can generalize to other program paths, although they may report some false positives. The idea is to track the set of synchronization objects protecting every memory object, and report races when this set becomes empty (i.e., the object is unprotected). Tracking this set of locks can be thought of as traversing a **lattice**, in which any set of locks higher up in the lattice as more locks, the bottom of the lattice as no locks, and the top has the global lock set. Locksets are intersected as data is accessed. For example, if $L_x = \{A\}$:

1. Acquire lock A – new lockset is $\{A\}$
2. Acquire lock B – new lockset is $\{A, B\}$
3. $x \leftarrow 1$
4. $L_x \leftarrow L_x \cap \{A, B\} = \{A\}$
5. Unlock B – new lockset is $\{A\}$
6. $x \leftarrow x + 1$
7. $L_x \leftarrow L_x \cap \{A\} = \{A\}$
8. (later) Race reported if L_x is empty

A state machine can protect against some common false positives, such as variables that are only ever used on a single thread.

18.4 Non-Race Concurrency Issues

In general, the absence of races does not make a program correct. The classic example is a bank transfer, with variables *bal* (the total bank balance), *me* (my balance), and *you* (your balance). I want to take some money out of the bank and give it to you:

1. $bal = bal - 10$
2. $me = me - 10$ // now I have cash in hand
3. $you = you + 10$
4. $bal = bal + 10$

To make this race-free, we can put a lock on every variable. This works, but there is still ‘manufactured money’ between the 1st and 2nd lines, since $me + you$ is greater than bal – this is an ‘atomicity violation’ or a ‘data anomaly’.

We also touched on some other concurrency problems and other issues:

- **Deadlock** – deadlock occurs when no threads are able to make progress due to locking. One method of trying to prevent deadlock is to prevent schedules that lead to deadlock from happening again.
- **Static analysis requirements** – performing any kind of static analysis requires access to all source code, including libraries and the like. This is often not feasible, and hence prevents useful static analysis from being performed.
- **Exceptions** – one idea mentioned was to make data races a runtime exception. Exceptions were also singled out as a good idea that end of being somewhat useless, since the majority of runtime exceptions can’t really be dealt with in a useful way (e.g., a index out of bounds exception probably just means you print an error trace and stop doing whatever you were doing).
- **Slow programming languages** – many ‘slow’ programming languages (such as Ruby, Perl, etc) are very popular today, largely because optimizing the efficiency of the programmer is more important than optimizing the program itself with today’s hardware.