CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 8: March 6

Lecturer: Emery Berger *Scribe: Vitaliy Lvin*

8.1 Data Races

Data races form an important class of bugs that software engineers have to deal with. For example, Linux kernel has 46 documented races.

Races are typically caused by a human error: failure to follow a locking discipline or overall lack of such. Their exposure depends on non-determinism of the thread scheduler, which can interrupt threads at any time and cause their instructions to be interleaved in unpredictable ways. They can cause unpredictable crashes and data corruption. The worst property of data races is that they are notoriously hard to detect, reproduce, locate, and eliminate, which has earned them the name of Heisenbugs (from the Heisenberg Uncertainty Principle).

Let's define races more precisely.

Definition 8.1 *A program has a data race if two or more threads in that program access some variable simultaneously and at least one of them changes its value.*

This definition says nothing about the undesirable side effects of data races. Races can often be benign or have no effect on program execution.

The usual way to prevent data races is to use mutual exclusion to serialize access to shared data.

Several types of data races exist:

- Read-write conflicts
- Write-write conflicts

8.2 Data race detection

There are two general approaches to detecting data races: static and dynamic.

In the static approach the source code is being analyzed to detect unserialized accesses to shared data. It may seem like a great idea, but existing techniques are essentially heuristics with very large numbers of false positives due to the fact that static analysis cannot distinguish data dynamically allocated on the heap, etc.

Most widely-used data race detection tools use the dynamic approach - at the run time.

8.2.1 Happens-before

The notion of happens-before relationship was first introduced by Lamport in the context of distributed systems, but can be successfully applied to dynamic race detection. The purpose of happens-before relationship

is to establish partial ordering of events across different threads (processes).

Definition 8.2 *Out of two sequential events in the same thread the earlier one is said to have happened before the latter one. An unlock operation in one thread is said to have happened before a lock operation in a different thread if those operations referred to the same lock (in the distributed context a send of a message always happens before a receive of that same message). Happens-before is transitive.*

Two accesses to a shared variable can form a data race if they cannot be orderd using happens-before. Detecting races using happens-before has a number of significant draw-backs:

- One has to instrument the code to track per-thread info about concurrent accesses to **every** shared location;
- Detection depends on scheduler-controlled interleaving of events to elicit races - hence a high false negative rate.

8.2.2 Eraser

Eraser presents a different approach to dynamic race detection. The key idea behind it is to track lock sets that govern access to each shared location. A data race is an access to a shared variable that is not governed by lock(s). It finds more races then happens-before based tools, but still causes 10-30x slow-down.

The lockset algorithm works as follows:

1. $\forall$ shared location v keep *C(v)* - set of **candidate locks**, initially set to contain all locks.
2. $\forall$ accesses to v set $C(v) = C(v) \cap$ set of currently held locks *(lock refinement step)*
3. $If\, C(v) = \emptyset$, show a data race warning

This algorithm can sometimes be **too** strict:

- During intialization, when no locks are usually held;
- Read-shared data that is only written during initialization and is safe without locks throughout the rest of the program;
- Reader-writer locks.

The following refinements fix the abovementioned problems:

- Ignore the initialization part and assume intialization is over when a thread different from the creator accesses shared data;
- Assume every shared location is safe to access without locks until first written;
- Track locks held only when writing separately from usual lock tracking to correctly handle reader-writer locks.

Eraser was originally implemented using ATOM binary rewriting tool on Alpha (modern binary rewriting tools like Pin exist). It kept a shadow word for every word of memory in the data section and on the heap, and instrumented every direct memory access, which cause 10-30x slow-down.

8.3 Races Not Enough

Races-freedom is neither a necessary nor sufficient condition of program correctness! The real goal should be atomicity.