Detecting data races on shared memory machines, for programs with locks and condition variables

Parosh Aziz Abdulla
Uppsala University, Sweden.
<parosh@it.uu.se>

Frédéric Haziza
Uppsala University, Sweden.
<daz@it.uu.se>

October 13, 2008

Abstract

This document represents the notes following the discussions in the research group, concerning the verification of multi-threaded programs using pthreads, mutexes (simple locks) and condition variables.

1 Introduction

A race condition is a situation in a shared-variable concurrent program in which one process changes a variable that another process has previously read and that other process doesn't get notified of the change.(1)

(1) Fred: or just doesn't see the change

A process can, for example, write a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. Another example: when a process checks a variable and takes action based on the content of the variable, it is possible for another process to "sneak in" and change the variable in between the check and the action in such a way that the action is no longer appropriate.

Alternatively, one can define a race condition as the possibility of incorrect results in the presence on unlucky timing in concurrent programs, that is, getting the right answer relies on lucky timing.

Race condition are of particular interest because they can lead to some rather devious bugs. These bugs are extremely hard to track since they are non-deterministic and difficulty reproducable. The kind of errors caused by race condition are very subtle and often manifest themselves in the form of corrupted or incorrect variable data. Unfortunately, if often means that the error won't crash the system immediately, but will wait until some other code is executed and which relies on the data to be correct, making the process of locating the original race condition even more difficult.

To avoid data corruption or incorrectness, the programmer uses synchronization techniques to constraint all possible process inter-leavings to only the desirable ones. Race condition usually lead to an incorrectly synchronized program.

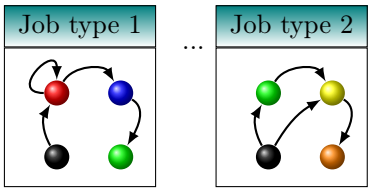
A precise way to detect race conditions would be to search the state space (of a *model* of the program) for a state where multiple threads try to access and change the same variable. The technique is sound, but more importantly it seems to be much more precise than other techniques (static analysis, dynamic detection, ...). It indeed detects the race conditions themselves instead of detecting violations of the locking discipline that can be used to prevent race conditions.

2 Snapshot

We describe in the section the information we require in order to depict a snapshot of the system at a given time.(2)

(2) Fred: Define system? Use *model* instead?

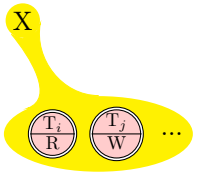
The system contains multiple separate threads of control and each thread is assigned a particular task, so-called *job type*, represented by a finite state automaton. We can assume, to start with, that the system has a finite number of job types.



The system (ie the whole program) is associated with a set of shared variables, so-called *Globals*. The Globals might be read and written by any threads. A thread has a unique id and is associated with a set of local variables, so-called *Locals* (such as its stack).

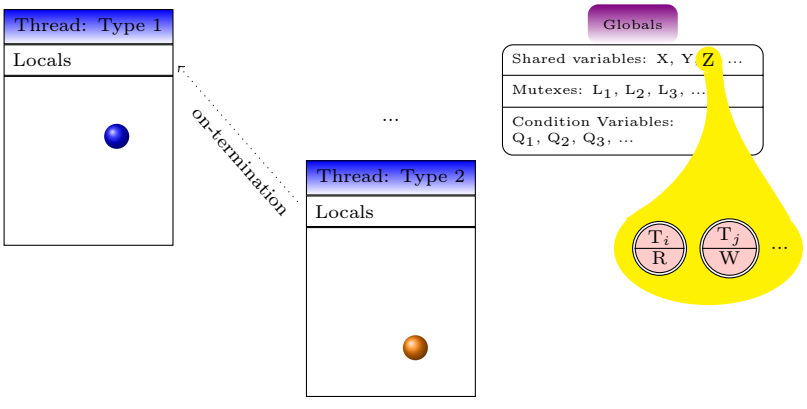
The communication between the threads is achieved by means of shared memory. Synchronization (i.e. constraining all possible thread inter-leavings to only the desirable ones) is achieved by means of mutexes and condition variables. A mutex is a simple lock with 2 states: taken or free. Any thread requesting to take the lock when the lock is taken will wait until the lock is freed. Condition variables represent queues of all the sleeping threads on an associated predicate.

Each shared variable is associated with a set of pairs, so-called *access set*. A pair represents the thread id accessing that variable and a token corresponding to the type of access: either read (R) or write (W). If no thread is accessing the variable, the access set is empty.



A thread can voluntarily stop its activity and resume it only when another thread terminates (either normally or abnormally). An *on-termination* association is set up and the sleeping thread gets notified and awaken when the awaited thread terminates.

So, a snapshot of the system would be depicted as follows: the Globals and a collection of thread with a given job type, a current state and Locals, with eventual on-termination associations. The Globals contains the mutexes, condition variables and all shared variables, with their respective access set.



3 Actions on the snapshot

Any high-level language will be somehow compiled or interpreted to run as machine instructions. The instruction set architecture (ISA) are often a relatively small set of opcodes matching to the assembly language. An ISA includes:

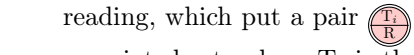
- Arithmetic (such as *add* and *sub*)
- Logic instructions (such as *and*, *or* and *not*)
- Data instructions (such as *move*, *input*, *output*, *load*, and *store*)
- Control flow instructions (such as *goto*, *if ... goto*, *call*, and *return*)

Arithmetic and logic instructions are simply pure CPU operations. We are insterested in instructions touching the main memory and instructions controlling them.

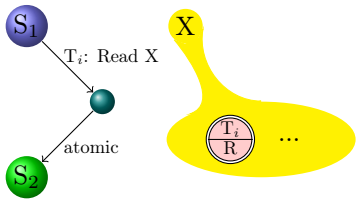
The language we allow abstracts away the CPU operations and narrows down the operations to a set of movers, i.e. loading from and storing to the main memory and operations related to thread bookkeeping. Operations like $x := y$ will be considered in our model as a 2-steps (atomic) operation composed of a load (of y) and a store (of x). Operations like $x := 3 * y$ will be considered as a 3-steps operation composed of a load (of y), a pure CPU operation, and a store (of x).

Abstracting away all instructions gives us a set of atomic movers from main memory (the Loads) and a set of movers to main memory (the Stores). We can abstract away the data itself. We will keep the control flow instructions to reflect them in the job types.

For the shared variables, we have then 2 actions: read and write.

- Read:
  Atomic action which we divide in two steps. The first step is a request for reading, which put a pair 

in the associated set, where T_i is the reading thread. The second step is to remove the pair (and perform the read atomically).

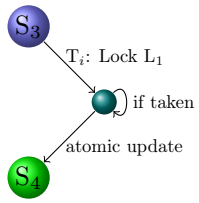


- Write:
  Similar to the read but a pair 

is inserted in the access set.

For the mutexes, we allow 3 actions: lock, unlock and try-lock. Each mutex is a special shared variable which value is either *taken* or *free*.

- Lock: (or grab, take, request, ...)
  The lock action starts by moving the current thread to a temporary state. If the mutex is free, the variable is atomically updated to taken, and the thread moves out of the temporary state. If the mutex is taken, the thread doesn't exit the temporary state. Note that this models well the fact that only one can succeed in locking the mutex.

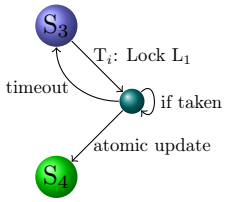


- Unlock:
  If the mutex was taken, resetting it to free will release the mutex, allowing other blocked threads to compete for the mutex. If it was free, nothing happens.[3]

[3] Fred: But still something fishy here...

- Lock with timer: (or try-lock)

  Behaves like the lock, but the request can timeout and the thread moves back to its original state.



For a thread, we allow creation and termination. We also allow a thread to wait for another one.[4]

[4] Fred: or several other ones?

- Creation:
  We add another thread entity to the collection with a new job type, a start state, some initialization for the locals. Note that all of the initialization steps can be parametrized.[5]

[5] Fred: multiple creation and join temporarily on hold

- Join(id):

  A Current thread ⋯⋯ on termination ⋯⋯ $\text{Thread}_{requested\ id}$ association is added and the calling thread goes to an intermediary sleeping state.

- Termination:
  The thread is removed from the collection.[6] . If an on-termination link was present, notification is send to all other joined threads, moving them out of their intermediary sleeping state.

[6] Fred: Its id is not recylcled. Other threads might still try to communicate with it, which should result in some sort of failure

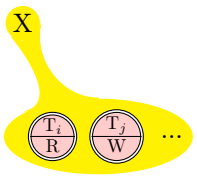
For the condition variables:[7]

[7] Fred: todo

- Wait
- Signal
- Broadcast

4 Bad states

A data race occurs when multiple threads access a shared variable and at least one has the intention of changing it, without any synchronization (i.e. ordering) constraints. Consequently, a bad state is one configuration that contains a variable where its access set has more than 2 pairs and one of the pairs contains a write token.



5 Ordering

We define in this section how two snapshots of the system can be compared.

6 Computing Pre

We define in this section how predecessors are computed.