Advances in Programming Languages

Lecture 9: Concurrency Abstractions

Ian Stark

School of Informatics
The University of Edinburgh

Monday 15 October 2018
Semester 1 Week 5

THE UNIVERSITY
of EDINBURGH

https://wp.inf.ed.ac.uk/apl18
https://course.inf.ed.ac.uk/apl

Outline

Outline

Programming-Language Techniques for concurrency

This is the second of a block of lectures looking at programming-language techniques for concurrent programs and concurrent architectures.

- Introduction, basic Java concurrency
- Concurrency abstractions
- Concurrency in different languages
- Current challenges in concurrency

Basic Java Concurrency

Java provides basic concurrency mechanisms as standard.

Threads Encapsulated in class Thread, these can run arbitrary code, share data, spawn subthreads and wait for children.

Scheduler Each Java runtime determines the degree and style of concurrency available on a particular platform.

Locks Every object has an *intrinsic* lock, which **synchronized** methods must acquire before execution to ensure mutual exclusion. Explicit locking allows finer delineation of *critical regions*.

Condition Variables Every object is a *monitor*, with wait() to block and notify()/notifyAll() to communicate between threads.

This language support is enough to write safe concurrent code and implement sophisticated concurrent algorithms. In particular, locks and condition variables avoid the need for busy-waiting and spin-lock loops.

Homework from Thursday

1. Do This

Find out what a *data race* is. What happens to C or C++ code with a data race?

2. Read This

Read the first three sections of the Java Concurrency tutorial.

http://java.sun.com/docs/books/tutorial/essential/concurrency

Processes and Threads • Thread Objects • Synchronization

Have another Java concurrency tutorial to recommend? Great! Post on Piazza or mail me.

Outline

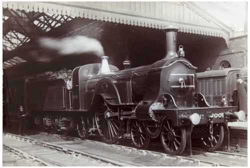
1. Opening
2. Races
3. Helpful Abstractions
4. Closing

Racing

Race Condition

A *race condition* (or just a *race*) in software or hardware is a situation where certain events may happen in different orders and some outcome depends on what that order turns out to be.

This is a very general term. A race is likely to be a problem if the system depends on things happening in one particular order, but there is no way to control that order.



Tony Hisgett, Flickr

ThinCat, Wikipedia

RISK ASSESSMENT —

“Most serious” Linux privilege-escalation bug ever is under active exploit (updated)

Lurking in the kernel for nine years, flaw gives untrusted users unfettered root access.

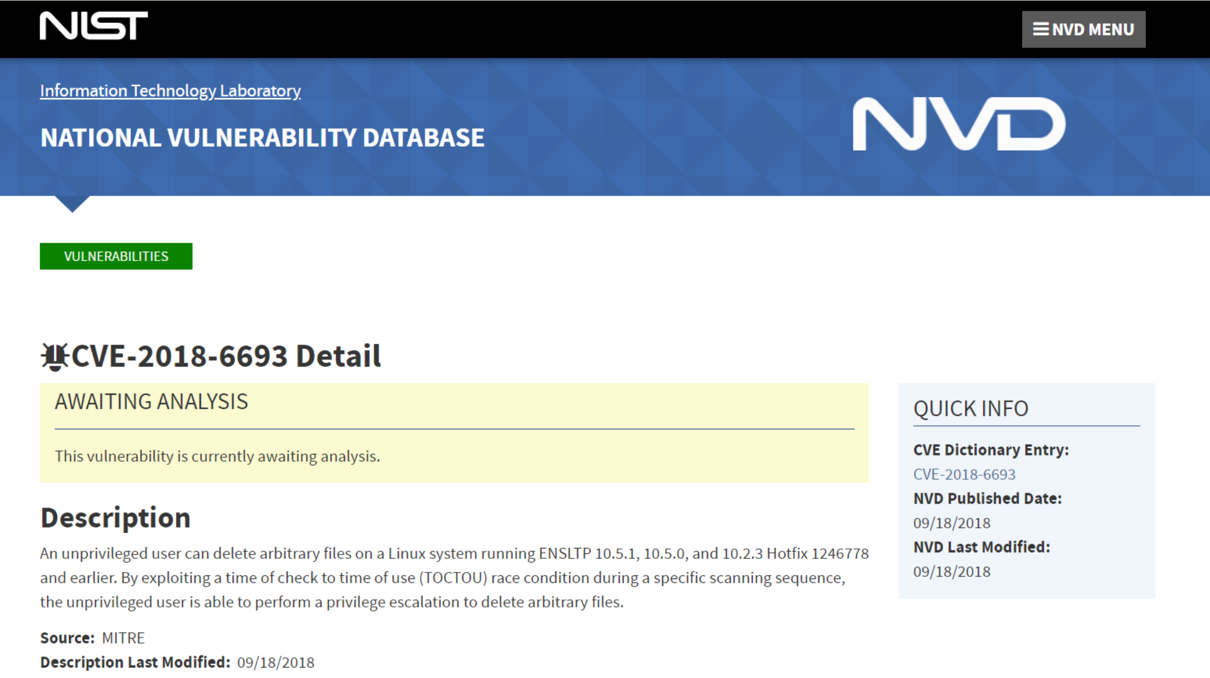
DAN GOODIN - 10/20/2016, 9:20 PM

DIRTY COW

Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel

DIRTY COW

(CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel



NVD MENU

Information Technology Laboratory

NATIONAL VULNERABILITY DATABASE

VULNERABILITIES

CVE-2018-6693 Detail

AWAITING ANALYSIS

This vulnerability is currently awaiting analysis.

Description

An unprivileged user can delete arbitrary files on a Linux system running ENSLTP 10.5.1, 10.5.0, and 10.2.3 Hotfix 1246778 and earlier. By exploiting a time of check to time of use (TOCTOU) race condition during a specific scanning sequence, the unprivileged user is able to perform a privilege escalation to delete arbitrary files.

Source: MITRE

Description Last Modified: 09/18/2018

QUICK INFO

CVE Dictionary Entry:

CVE-2018-6693

NVD Published Date:

09/18/2018

NVD Last Modified:

09/18/2018

McAfee™

Together is power.

McAfee Endpoint Security for Linux Threat Prevention 10.5.0

Racing

Data Race

A *data race* is more precisely defined. It is a situation where a program contains two memory accesses with the following properties:

- They happen in different threads;
- Both target the same memory location;
- At least one is a write operation;
- There is no concurrency control to make sure they don't happen at the same time.

If Java code contains a data race, then some high-level guarantees about sensible multicore memory behaviour are lost.

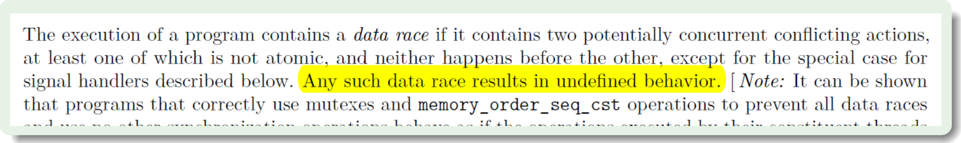
Specifically *sequential consistency*

If C/C++ code contains a data race, then the behaviour of that code is undefined. Really, really undefined: a standards-compliant compiler can do *absolutely anything at all*.

"Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."

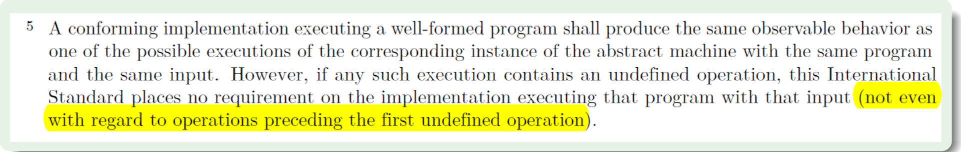
Data Races in the C++ Language Standard

Data Race



The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [*Note:* It can be shown that programs that correctly use mutexes and `memory_order_seq_cst` operations to prevent all data races

Undefined Behaviour



5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

Undefined behavior can result in time travel
Post on “The Old New Thing” blog by Raymond Chen, Microsoft, June 2014

Data Races and Thread Safety

Racy code can lead to consistency problems and other errors, some of which may even depend on scheduler and platform details.

It's important to identify classes that are *thread safe*: where methods run correctly in the presence of other threads, and even when called simultaneously from different concurrent threads.

Java threads and locks make it possible to write such code, and particular idioms or patterns can help to do so correctly. For example:

- An *immutable* object cannot be modified once constructed. Functional languages deal almost exclusively in immutable values; Java uses this pattern in libraries like String.
- Restricting field access to **synchronized** methods that *get*, *set* and *update* values can help to make a class thread safe.

Outline

1. Opening
2. Races
3. Helpful Abstractions
4. Closing

Synchronization Wrappers

The Java Collections class provides several general operations on collections. The synchronizedXYZ(...) wrapper methods return thread-hardened versions of existing collections.

From class java.util.Collections

```
List unsafelist = new ArrayList();

unsafelist.add("This");     // This is only safe if no other
unsafelist.add("That");     // thread can access the list

List list = Collections.synchronizedList(unsafelist); // Thread-safe version of the list

// We can safely start two concurrent threads where both have access to the list
...                      |      ...
list.add("Things");      |      list.add("Thing 1");   // No need to synchronize
int n = list.size();     |      list.add("Thing 2");

// Result n could be 3, 4, or 5, but the list will remain consistent.
```

Concurrent collections

Making more methods **synchronized** may give safer code, but it can also become a serious bottleneck and reduce the benefits of concurrency, or even lead to complete deadlock.

Synchronization wrappers on existing collections can help — they will only add locks where needed, but even this can slow things down.

Java 5 introduced *concurrent collections*, bringing in both new algorithms and new ways to use collections effectively in a threaded environment.

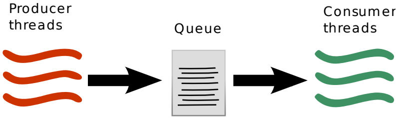
Interfaces

- ConcurrentMap
- BlockingQueue
- BlockingDeque
- TransferQueue

Classes

- ConcurrentHashMap
- ConcurrentLinkedQueue
- CopyOnWriteArrayList
- ArrayBlockingQueue *etc.*

Example: Producer/Consumer Pattern (1/2)



The *producer-consumer pattern* is a way to decouple tasks and achieve scalable parallelism.

A queue allows independent tasks to proceed on each side without interfering. Consumers block when the queue is empty; producers block when the queue is full.

If we just used a regular queue, then simultaneous actions by multiple producer and consumer threads may leave its internal datastructures in an inconsistent state.

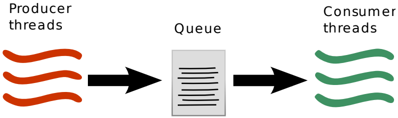
To make this thread-safe we could wrap it to make all uses of the queue **synchronized**. That works, but the access lock is then a bottleneck for responsive concurrency.

A smarter implementation can support simultaneous access to both ends of the queue — for example, with separate locks for adding and removing elements.

Example: Producer/Consumer Pattern (1/2)

Implementations of the Java BlockingQueue provide a highly-concurrent thread-safe queue.

They also support concurrency-aware programming by offering different suites of methods for different concurrency scenarios.



All approaches offer ways to *insert*, *remove* and *examine* queue items. The difference is in what happens when these cannot work because the queue is full or empty.

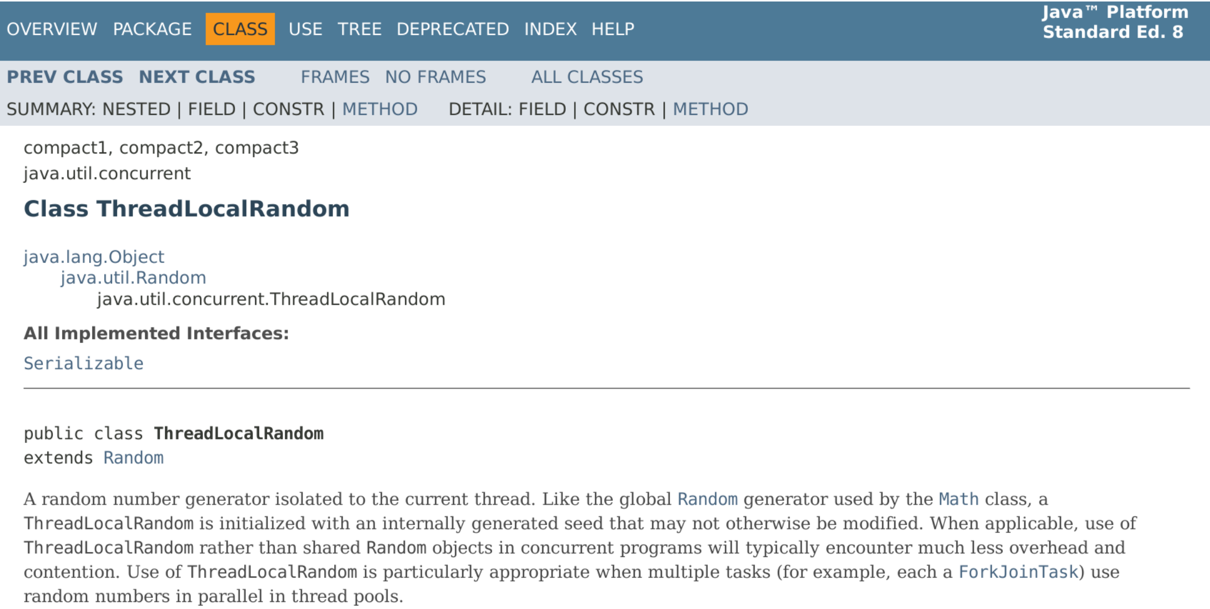
java.util.concurrent.BlockingQueue

		Exception	**Option**	**Block**	**Timeout**
Operation:	Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
	Remove	remove()	poll()	take()	poll(time, unit)
	Examine	element()	peek()	–	–

Quiz

The `java.util.concurrent` package was introduced in Java 5. With the release of Java 7 it included a new class with a special concurrent algorithm to do one of the following. Which one, and why?

- A Write the system clock.
- B Generate a pseudo-random number.
- C Read the current heap size.



compact1, compact2, compact3

java.util.concurrent

Class ThreadLocalRandom

java.lang.Object
  java.util.Random
    java.util.concurrent.ThreadLocalRandom

All Implemented Interfaces:

`Serializable`

`public class` **`ThreadLocalRandom`**
`extends Random`

A random number generator isolated to the current thread. Like the global `Random` generator used by the `Math` class, a `ThreadLocalRandom` is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of `ThreadLocalRandom` rather than shared `Random` objects in concurrent programs will typically encounter much less overhead and contention. Use of `ThreadLocalRandom` is particularly appropriate when multiple tasks (for example, each a `ForkJoinTask`) use random numbers in parallel in thread pools.

More Elaborate Concurrency

The java.util.concurrent package includes a wide range of more sophisticated concurrency idioms that enhance Java's standard threads, locks and monitors.

Locks — Re-entrant locks, read-write locks, condition variables.
Executors — Thread factories, thread pools, alternative scheduling.
Fork/Join — Managing large numbers of concurrent lightweight tasks.
Futures — Asynchronous computations returning values.
Synchronizers — Semaphores, latches, barriers, phasers, exchangers.

Note that this is still a library: all can be implemented using the standard Java concurrency primitives. Compared to writing these yourself, though:

- The library is tried, tested, and maintained;
- The algorithms support a high degree of concurrency.
- On some platforms they may be able to use additional low-level concurrency support.

When Racy Code is Good — From java.lang.String source code (JDK 8)

```java
public final class String {
...
    private final char value[];
...
    private int hash; // Default to 0
...
    public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;

            for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
     }
```

Outline

1. Opening
2. Races
3. Helpful Abstractions
4. **Closing**

Summary

- Java provides threads, locks and monitors as language primitives.
- These are sufficient to write explicitly concurrent code.
- They also allow all kinds of bad things: interference; deadlock; livelock; . . .

- Writing *thread-safe* code is possibly, just tricky.
- Patterns can help: immutability, atomicity, synchronization wrappers.

- Java's concurrent collections are thread-safe *and* add performance.
- Java's concurrency libraries add many more concurrency idioms.

Homework

1. Do this

Find out what the addAndGet method on a Java AtomicLong object does. Why is that useful?

Java 8 introduced a LongAdder class. Find out what it does, and how it can make code faster.

2. Read this

The remaining sections of the Java Concurrency tutorial

http://java.sun.com/docs/books/tutorial/essential/concurrency

- Liveness
- Guarded Blocks
- Immutable Objects
- High Level Concurrency

A simple blocking method

!

```
class Pigeonhole {

  private Object contents = null;

  synchronized void put (Object o) {

    while (contents != null)    // Wait until the pigeonhole is empty
      try { wait(); }
      catch (InterruptedException ignore) { return; }

    contents = o;               // Fill the pigeonhole
    notifyAll();                // Tell anyone who might be interested
  }
 ...
}
```

Optional Coding Exercise !

Extend the Pigeonhole class to include methods to check whether there is anything in the pigeonhole, and to release the contents of the pigeonhole.

Write a PigeonFancier program that:

- Has a fixed number of pigeon-holes which are emptied by some dedicated pigeon-handler threads releasing pigeons after random delays;
- Has a single thread which regularly puts new pigeons into empty holes;
- Make sure the pigeon stuffer doesn't wait too long for any hole to become empty.

References

Brian Goetz, Time Peierls, Joshua Block, Joseph Bowbeer, David Holmes and Doug Lea.
Java Concurrency In Practice.
Addison Wesley, 2006.
Current essential reference for concurrent Java programming.

Doug Lea.
Concurrent Programming in Java: Design Principles and Patterns.
Second Edition. Addison-Wesley, 1999.
The original standard text, describing many of the patterns now implemented inside `java.util.concurrent`, and some of the horrors of the Java Memory Model.