Problems with Concurrency

February 19, 2014
Overview

- problems with concurrency
  - interleavings
  - race conditions
  - deadlocks

- More examples
  - GUI

- source of problems
  - non-determinism
  - deterministic execution model
Overview

Problems
General ideas
Shared variable
Access
interleavings
Race condition
Race condition: example
Solution: locks
Java structured locks
(synchronized)
Problem with locks
Shared state in structures
Monitors
Monitor example (1)
Monitor example (2)

More examples

Core of the problem
■ common observation: “parallel programming is difficult”
■ typical programming models used:
  ◆ shared memory (locks, monitors, Clojure refs, STM)
  ◆ message passing (actors, Clojure agents)
■ nowadays many suggest start with declarative model (1st part of this course)
  ◆ think (declarative = pure functional)
  ◆ “Intel® Array Building Blocks software implements a deterministic parallel programming model”
  ◆ Scala book suggests to start with val not var
Shared variable

- memory hierarchy (registers, L1/L2 caches, memory)
- 2 or more threads
  - change the same variable:
    1. read value from memory to register
    2. change value in register
    3. write register value back to memory
  - do not synchronize anyhow

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=x+2</td>
<td>x=x+3</td>
</tr>
</tbody>
</table>

- may happen that the value is incremented by only 2 or 3
Access interleavings

**Overview**

**Problems**

General ideas

Shared variable

Access interleavings

Race condition

Race condition: example

Solution: locks

Java structured locks

(synchronized)

Problem with locks

Shared state in structures

Monitors

Monitor example (1)

Monitor example (2)

More examples

Core of the problem

---

**Access interleavings**

- **normal (expected) execution (assume x=5)**
  
  1. thread 1: reads 5
  2. thread 1: increments its register by 2
  3. thread 1: writes 7 back
  4. thread 2: reads 7
  5. thread 2: increments its register by 3
  6. thread 2: writes 10 back

- **undesirable *interleaving* of steps may have**
  
  - x=7 or x=8 at the end

- **race condition** – different scheduling may give different result from the same input

- **idea** – limit allowed interleavings (with locks)
Race condition

**description**

- undesirable
- same program and same input may give different output

**typical characteristics**

- happens rarely
- usually under extreme conditions
  - server heavy load, etc
- not visible during development
- extremely difficult to debug (does not repeat)
Race condition: example

- several \((N)\) threads increment `count` variable
- the result may be less than \(N \times 10000000\)

◆ luckily easy to repeat..

```java
static volatile int count = 0;
/** Creates new thread that increments the variable */
static Thread newThread() {
    Thread thread = new Thread() {
        public void run() {
            for (int i = 0; i < 10000000; i++) {
                count++;
            }
        }
    };
    thread.start();
    return thread;
}
```
Solution: locks

- limit access to a shared variable to 1 thread at a time
  - a thread
    1. takes the lock
    2. changes the variable
    3. releases the lock
  - another thread blocks until the lock is released

- Java `synchronized(var) {}` block
  - `critical section` – structured take/release
  - less flexible (impossible to lock/release from separate functions)
  - less error-prone (cannot forget to release, e.g. on exception)
Java structured locks (synchronized)

- Every object has *monitor* as a part of it!
- Monitor is a lock with the possibility of wait/notify
- Using it just as a lock
  - critical section with `synchronized(obj) {}`
    - monitor acquired on entry, released on exit
    - code section, synchronizing on the same instance
      - `obj` can be entered by one thread at a time
    - other threads sleep before the code section until
      the monitor is released
  - `synchronized void fun() {}`
    - is like using `synchronized(this)` for the whole method
  - *reentrant lock* – one thread can acquire the same
    monitor many times (nested synchronized calls/blocks on the same object)
Problem with locks

**Overview**

**Problems**
- General ideas
- Shared variable
- Access
- interleavings
- Race condition
- Race condition: example
- Solution: locks
- Java structured locks
  (synchronized)

**Problem with locks**
- Shared state in structures
- Monitors
- Monitor example (1)
- Monitor example (2)

**Core of the problem**

- 2 threads; 2 locks: A,B
  1. thread 1: takes A
  2. thread 2: takes B
  3. thread 1: blocks on B
  4. thread 2: block on A

- *deadlock* – variant of race condition
  - blocks occasionally
  - if always blocks – easy to spot and fix

*Race condition is the source of the problem.*
Shared state in structures

- several shared variables
- example: linked list with explicit `size` variable
  1. read `size` variable
  2. traverse to the end of the list
  - add new element to the list
  3. increment and write back `size` variable

- `size` variable and the list elements may go out of sync
- concurrent access of non thread-safe structures
  - asking for trouble!
  - none of `java.util.*` are thread-safe
Monitor is a lock that in addition to locking allows to wait for condition:

- for example, lock a message queue; if it is empty
  - release the queue,
  - wait on the queue until some other thread puts a message and notifies us.
- If the two operations are not atomic we may end up waiting on non-empty queue.
- wait() – *atomically*: releases objects monitor and puts thread into sleep.
  - Must be in synchronized block!
- notify(), notifyAll() – wakes up one/all waiting threads
Monitor example (1)

- **consumer** – thread that reads values from the queue
  - sleeps if it is empty

```java
class Consumer extends Thread {
    ArrayList<String> buffer = new ArrayList<String>();
    volatile boolean finish = false; // notice volatile!

    public void run() {
        while (!finish) {
            try {
                consume();
            } catch (InterruptedException e) {}
        }
        System.out.println("Finishing");
    }

    private void consume() throws InterruptedException {
        synchronized (buffer) {
            // use buffer object as a monitor for itself!
            if (buffer.isEmpty()) {
                buffer.wait(); // atomically: release 'buffer' and sleep
            } else {
                System.out.println("Got " + buffer.get(0));
                buffer.remove(0);
            }
        }
    }
}
```

Overview

Problems

General ideas
Shared variable
Access
interleavings
Race condition
Race condition: example
Solution: locks
Java structured locks
(synchronized)
Problem with locks
Shared state in structures

Monitors

Monitor example (1)
Monitor example (2)

More examples

Core of the problem
Overview

Problems

General ideas

Shared variable

Access

interleavings

Race condition

Race condition: example

Solution: locks

Java structured locks

(synchronized)

Problem with

locks

Shared state in

structures

Monitors

Monitor example

(1)

Monitor example

(2)

More examples

Core of the

problem

---

producer – puts values to the queue

notifies the consumer in case it is waiting

```java
public class MonitorExample {
    public static void main(String[] args) throws InterruptedException {
        Consumer consumer = new Consumer();
        consumer.start();
        synchronized (consumer.buffer) {
            // do not forget to acquire the monitor!
            consumer.buffer.add("Hello");
            consumer.buffer.notify();
        }
        Thread.sleep(1500);
        consumer.buffer.add("consumer"); // bug!!, changing shared object
        consumer.finish = true; // first(!) set the finish flag
        consumer.interrupt(); // consumer may be sleeping in wait()!
    }
}
```
Overview

Problems

More examples
GUI state as shared state
Example: search files
Single thread solution
Two thread solution
Two threads with invoke
Ooops: how easy to overlook
Aggregate queue
Bounded queue

Core of the problem

More examples
GUI state as shared state

- consider GUI as a shared state
  - window/component states and dimensions, etc
  - e.g., new element in GUI table may cause
    - appearance of scrollbars
    - re-layout of many other elements

- still wonder why? (Swing’s Threading Policy)
  All Swing components and related classes, unless otherwise documented, **must** be accessed on the *event dispatching thread*. 
Example: search files

- Example from the 1st lab
  - search files and add them to the list

- Example for the 2nd lab
  - search files and add file *extensions* to the list
  - show file count and sort
Single thread solution

■ “search files” button press
■ handled in AWT

_event dispatching thread_

◆ continue searching for files in this thread
◆ update Swing list from this thread

■ usually Swing components updates are postponed

■ no other Swing components react during this time

◆ application freezes, unable even to stop the search!
Two thread solution

- new thread is created
  - start searching for files in the new thread
  - update Swing list from the new thread

- AWT event dispatching thread continues execution
  - may handle “stop search” button press

- lots of race conditions in GUI
  - solution occasionally works (with small directories)
  - unacceptable as may cause unexpected errors
Two threads with invoke

- similar to the previous, but
  - update Swing list from the AWT thread
    - create Runnable, put to the AWT event queue

```java
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        ThreadSearchListener.super.addElement(file);
    }
});
```

* Overview
  - Problems
  - More examples
    - GUI state as shared state
      - Example: search files
    - Single thread solution
    - Two thread solution
  - Two threads with invoke
    - Ooops: how easy to overlook
    - Aggregate queue
    - Bounded queue
  - Core of the problem
Overview

Problems

More examples
GUI state as shared state
Example: search files
Single thread solution
Two thread solution
Two threads with invoke

Ooops: how easy to overlook

Aggregate queue
Bounded queue

Core of the problem

- code given on the 1st lab
  - in “finally”: GUI is called by the new thread
  - may work in 99.95% of the runs
  - once application will crash or some buttons remain disabled

```java
thread = new Thread(new Runnable() {
    public void run() {
        try {
            searchFiles(rootDir);
        } catch (ControlException exc) {
            System.out.println(exc);
        }
        finally {
            main.searchButton.setEnabled(true);
            main.stopButton.setEnabled(false);
            isStopSearch = false;
        }
    }
});
```
Aggregate queue

Overview

Problems

More examples
GUI state as shared state
Example: search files
Single thread solution
Two thread solution
Two threads with invoke
Ooops: how easy to overlook

Aggregate queue
Bounded queue
Core of the problem

- in previous solution: for every new file found
  - put new Runnable to the list
  - in AWT thread
    - sort and update the Swing list

- make more efficient
- now: process files by chunks
  - add new files to the aggregateQueue
  - schedule update for AWT thread
    - collects all files from the aggregateQueue and updates the Swing list
Bounded queue

Overview

Problems

More examples

GUI state as shared state
Example: search files
Single thread solution
Two thread solution
Two threads with invoke Ooops: how easy to overlook
Aggregate queue

in previous solution: aggregate queue grows indefinitely
now: stop file search thread if the aggregateQueue is full

◆ use Java wait()/notify()
Overview

Problems

More examples

Core of the problem

Non-determinism
Non-observable
non-determinism
Observable
non-determinism
Deterministic
execution model
(1)
Deterministic
execution model
(2)
Non-determinism

- race condition – non-deterministic behavior
- two kinds of non-determinism

1. *observable* – program may give different result
2. *non-observable* – program may execute differently, but this does not affect the result

- only the 1st kind causes race conditions
- the difference may be vague at the moment, but important for
  - checkpointing (save the state and restart a program), program debugging
  - or reasoning about program behavior in general!
Non-observable non-determinism

- Assume all variables = 0 at start

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=5</td>
<td>y=6</td>
</tr>
<tr>
<td>z=x*4</td>
<td>y=y+t</td>
</tr>
</tbody>
</table>

- If thread scheduler runs the 1st thread first then the states are
  \[ s_0 = \{ x = 5, z = 0, y = 0 \}, s_1 = \{ x = 5, z = 20, y = 0 \}, s_2 = \{ x = 5, z = 20, y = 6 \} \]

- If thread scheduler runs the 2nd thread first then consequent states are different

  ◆ but the result is always the same!
Observable non-determinism

- an example with shared variable (see lecture first example)
- an example with *non-deterministic action* on input channel

<table>
<thead>
<tr>
<th>process 1</th>
<th>process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>if inbox is empty then f1()</td>
<td>send m to process1</td>
</tr>
<tr>
<td>else f2()</td>
<td></td>
</tr>
</tbody>
</table>

- depending on network delays, etc, the result of the whole program may be different
- notice, if one could *not* branch on empty inbox
  - i.e. had to wait until the message comes
Deterministic execution model (1)

- single-assignment “variables” (now unvariable)
  - reassigning a variable
    - causes exception (Oz)
    - not possible by language semantics (Haskell)

- using value of unassigned variable is not possible
  - blocks until it is assigned (Oz)
  - evaluates when values is needed (Haskell)

```
x=5
y=6
z=x*y
t=y+z
```
Deterministic execution model (2)

- divide between 2 threads

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=5</td>
<td>y=6</td>
</tr>
<tr>
<td>z=x*y</td>
<td>t=y+z</td>
</tr>
</tbody>
</table>

- program execution is *deterministic* evaluation of values from other values
  - the result is always the same