Parallel Programming
Exercise Session 9

Feedback: Exercise 8

Essential

Race Conditions

- Race condition occurs if multiple accesses can happen concurrently and at least one access is a write

Thread 1	Thread 2
x=23;	y=42;
r1 = x;	r2 = y;
v = r1;	w = r2;
z = 2;	

- Thread 1 accesses x, v, z
- Thread 2 accesses y, w
- No race condition, no bad interleaving possible

Essential

Race Conditions

- Race condition occurs if multiple accesses can happen concurrently and at least one access is a write

Thread 1	Thread 2
x = 23;	y = 42;
r1 = x;	r2 = y;
v = r1;	w = r2;
y = 2;	z = 2;

- Thread 1 accesses x, v, y
- Thread 2 accesses y, w, z
- Both write to y! Result depends on interleaving!

Relations

- For a set S we can define a mathematical relation R for members of S

- Example:
  - Set of natural numbers, relation “greater or equal than”
  - Set of all humans, relation “knows”

Relations

- Relations can have different properties

- Example:
  - Transitivity: a R b and b R c implies a R c

  - The “greater than” relation is transitive.
  - The ”knows” relation is not transitive.

Transitive Closure

- For a relation R, the smallest relation which contains R and is transitive, is called the transitive closure of R.

- Example:
  - For the set of airports in the world, we can define the relation “offers direct flight to”
  - This is (probably) not transitive (show why)
  - The transitive closure also has meaning: ”can fly from a to b with stops”

Relations and Code

- When we execute code, ”actions” happen, i.e., a variable gets read or written
- We can define relations for these actions, such as “is executed before”
  - Easy to check because it is a local property
  - Can build the transitive closure if we want to know if actions are ordered!
  - Not all actions are ordered!

Program Order

Essential

- Not all actions are ordered!

```
S1: a=23;
S2: x=3;
S3: if (x==3) {
S4:   b += 1;
    } else {
S5:     b *= 2;
S6:     x = 0;
    }
S7: x=4;
```

Not in program order!

Program Order

- Action in mutually exclusive code paths are not in program order
- Actions in different threads are not in program order!

- But ordering was good for proofs!
- Want to allow the compiler / hardware to reorder sometimes for performance.

- Solution: Let compiler reorder whenever it is not “observable” – need to define a subset of special actions which are visible across threads

Synchronization Actions

- Solution: Let compiler reorder whenever it is not “observable” – need to define a subset of special actions which are visible across threads

  - For this lecture, the most important synchronization actions are
    - Start/End of a thread
    - Read/Write of a volatile or atomic variable
    - Acquire / release of a monitor

Synchronizes With Relation

- The variable x is initially 0.
- Thread A writes x=5
- Thread B reads/prints the value of x

- We could see 0 -> then we expect that B executed before A
- We could see 5 -> then we expect that A executed before B

- The synchronizes with relation says a read of a volatile must return the last value written to it. It synchronizes with that last write across threads!

Synchronizes With Relation

- If we combine (the transitive closure of) program order and "synchronizes with" we get the "happens before" order
- Any output/result we see in a Java Program must be consistent with this happens before order

Essential

Java Memory Model Takeaways

- If a variable is not declared volatile you must use the happens-before order to reason about possible values -> requires thought
- If a (primitive) variable is volatile it behaves like an atomic register

- Can sometimes gain performance (and maintain correctness) by not marking everything volatile.
- But you are using Java, is performance really your focus? ☺ - if in doubt declare shared variables as volatile

Java Memory Model Extra Resources

- The Java Memory Model, by Manson, Pugh and Adve
- Java Language and Virtual Machine Specification

Lecture Recap

Atomic operations

- An atomic action is one that effectively happens at once i.e. this action cannot stop in the middle nor be interleaved
- It either happens completely, or it doesn't happen at all.
- No side effects of an atomic action are visible until the action is complete

Essential

Atomic registers

- Atomic registers => support read and write, nothing else
- Usually we think of reads / writes as atomic, i.e., if we write a line such as x=1 in pseudocode we assume it happens atomically and is globally visible.

- This is not true in Java (unless x is e.g., AtomicInteger)
- Volatile makes it globally visible (but not atomic in all cases)

Essential

Atomic registers

- An operation such as x++ (with x being an atomic register) is NOT atomic!
    - Three steps: v = read(x), increment v, write(x, v)

- Problem with atomic registers:
    - Need O(n) space to synchronize n threads -> bad
    - Fix: support more than read/write in an atomic operation

Hardware support for atomic operations

Different atomic operations have been proposed, unclear which is best

- Test-And-Set (TAS)
- Compare-And-Swap (CAS)
- Load Linked / Store Conditional
- http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

Hardware Semantics

boolean TAS(*memref* s)

atomic

```
if (mem[s] == 0) {
        mem[s] = 1;
        return true;
} else

return false;
```

int CAS (*memref* a, int old, int new)

atomic

```
oldval = mem[a];

if (old == oldval)

        mem[a] = new;

return oldval;
```

Locks with atomics

- Now we can implement locks for n threads using a single variable:

  - Lock: while (!TAS(l)) {}
  - Unlock: mem[l] = 0

Bus Contention

- TAS/CAS are read-modify-write operations:
  - Processor assumes we modify the value even if we fail!
  - Need to invalidate cache
  - Threads serialize to read the value while spinning

TATAS

- Idea: Use normal operation to read first, try TAS only if first read returns 0
- Helps a bit. But what about the case where we see 0 first, then 1 in TAS? Can this happen?
  - Yes, and the more threads the more likely ☺

Exponential Backoff

- Idea: Each time TAS fails, wait longer until you re-try
- Works well, must tune parameters (how long to wait initially, when to stop increasing)
- Same concept in networks, people talking in a high-latency zoom call, etc.

Deadlock

- Circular dependency between resources/lock and threads
- Nobody can make progress

- Avoid by introducing global order in which locks are taken
  - Cannot have circles now since all dependencies go “in one direction”
- Or by not using locks at all! (Lock-free, wait-free, more later)

Essential

java.util.concurrent.atomic.AtomicBoolean

```
boolean set();

boolean get();

boolean compareAndSet(boolean expect, boolean update);

boolean getAndSet(boolean newValue);
```

atomically set to value `update` iff current value is `expect`. Return true on success.

sets `newValue` and returns previous value.

Progress Conditions

- Freedom of deadlock

  At least one thread is guaranteed to proceed into the critical section at some point

- Freedom of starvation

  All threads are guaranteed to proceed into the critical section at some point

- Freedom of starvation => Freedom of deadlock

Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

Peterson Lock

- Prove deadlock freedom and starvation freedom
- Hint: Only prove starvation freedom. Deadlock freedom will follow from that.

Fairness

- Intuitive explanation: If thread A calls lock() before thread B, then thread A should enter the critical section (CS) before thread B

- More formally: Divide the protocol into a doorway interval D and a waiting interval W

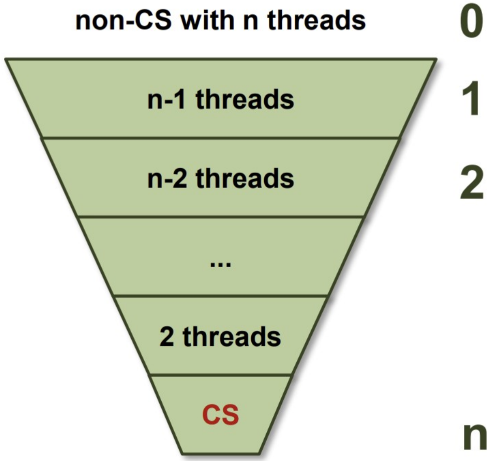
- D has a finite number of steps
- W has an unbounded number of steps

Fairness

- So called “first-come-first-served”

- Explanation of notation on blackboard
- Is the Peterson lock fair?
- What is the doorway interval D and the waiting interval W?

Filter Lock




```
int[] level(#threads), int[] victim(#threads)


lock (me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```

Other threads are at same or higher level

And I have to wait

Filter lock is not fair

- What is the doorway section and what is the waiting section?
- Remember the number of steps in the doorway should be bounded

- Doorway section = first two instructions in the first for-loop iteration

- Now find an execution that proves that the filter lock is not fair
- Hint: Use 3 threads

Assignment 9: Overview

- Analyzing locks
- Atomic operations

Analyzing locks

- The sample code represents the behavior of a couple that are having dinner together, but they only have a single spoon.

- Prove or disprove that the current implementation provides mutual exclusion.
    - HINT: Use State space diagram

Atomic operations

- In this task, we will see and analyze:
  - the usage of atomic operations to perform concurrency control, and
  - the cost of using them when having data contention

- For more details, please refer to the assignment sheet

Kahoot!

https://create.kahoot.it/details/6c967a18-d238-46fc-9e7b-bcde3d3ac187