# Programming Paradigms

# Concurrency (Part 3)

**Prof. Dr. Michael Pradel**

**Software Lab, University of Stuttgart**

**Summer 2020**

# Overview

- Introduction
- Concurrent Programming Fundamentals
- Implementing Synchronization ←
- Language-level Constructs

# Synchronization

- **Two high-level goals**
  - Make some operation atomic: Multiple instructions of a thread appear to other threads as always executing together
    - Mutually exclusive locks: Ensure that only one thread enters a critical section at a time
  - Condition synchronization: Delay some operation until some precondition holds

# Synchronization vs. Parallelism

- **Inherent trade-off in concurrent software**
  - Synchronization is needed to ensure correctness of computation
  - Synchronization reduces the amount of possible parallelism

# Busy-Wait Synchronization

- **Spin locks**
  - Provide mutual exclusion
- **Barriers**
  - No thread continues until all threads have reached a specific point

# Spin Lock

- **Goal: Ensure mutual exclusion**
- **In principle: Can implement with only load and store operations**
  - But: Super-linear time and space requirements
- **In practice: Implemented using special hardware instructions**
  - Read, modify, and write a memory location as one atomic step

# Test-and-Set

- **Instruction that**
  - sets a boolean variable to true and
  - returns whether it was false before
- **Spin-lock implementation:**

```
// Pseudo code
while not test_and_set(L)
  // nothing (spin)
```

# Test-and-Set

- **Instruction that**
  - sets a boolean variable to true and
  - returns whether it was false before
- **Spin-lock implementation:**

```
// Pseudo code
while not test_and_set(L)
  // nothing (spin)
```

**Problem: Repeated writes when lock is already acquired harms performance (“contention”)**

## Example:

| T1 | L = false | T2 |
| --- | --- | --- |
| tas(L) | | |
| | L = true | |
| CS { ⋮ | | tas(L) |
| | | tas(L) |
| | | tas(L) |
| L = false | | |
| | | tas(L) ✓ |
| | L = true | |

# Test and Test-and-Set

- Avoid **contention caused by repeated writes**
- Spin-lock implementation:

```
// Pseudo code
boolean L = false

procedure acquire_lock(L)
  while not test_and_set(L)
    while L
      // nothing (spin)

procedure release_lock(L)
  L = false
```

# Test and Test-and-Set

- Avoid contention caused by repeated writes
- Spin-lock implementation:

```
// Pseudo code
boolean L = false

procedure acquire_lock(L)
  while not test_and_set(L)
    while L
      // nothing (spin)

procedure release_lock(L)
  L = false
```

When another threads holds the lock, reads repeatedly (which is fast due to caching)

# Barrier

- **Goal: Ensure that all threads finish one phase before entering the**
- **Implementation based on atomic fetch-and-decrement**
  - Shared counter initialized to n
    - n .. number of threads
  - Decrement when a thread reaches the barrier
  - Last to arrive flips a shared boolean, which all others are waiting for

# Barrier: Pseudo Code

```
integer n = // nb of threads
boolean sense = true
local_sense = true // thread-local variable

procedure barrier()
  local_sense = not local_sense
  if fetch_and_decrement(count) == 1
    count = n
    sense = local_sense
  else
  repeat
    // spin
  until sense == local_sense
```

# Barrier: Pseudo Code

Shared flag to indicate whether all threads can proceed

```
integer n = // nb of threads
boolean sense = true
local_sense = true // thread-local variable

procedure barrier()
  local_sense = not local_sense
  if fetch_and_decrement(count) == 1
    count = n
    sense = local_sense
  else
  repeat
    // spin
  until sense == local_sense
```

# Barrier: Pseudo Code

```
integer n = // nb of threads
boolean sense = true
local_sense = true // thread-local variable

procedure barrier()
   local_sense = not local_sense
   if fetch_and_decrement(count) == 1
      count = n
      sense = local_sense
   else
   repeat
      // spin
   until sense == local_sense
```

**Local and global flag are the same means all threads can proceed**

# Barrier: Pseudo Code

```
integer n = // nb of threads
boolean sense = true
local_sense = true // thread-local variable

procedure barrier()
  local_sense = not local_sense
  if fetch_and_decrement(count) == 1
    count = n
    sense = local_sense
  else
  repeat
    // spin
  until sense == local_sense
```

Reinitialize for next iteration

# Barrier: Pseudo Code

```
integer n = // nb of threads
boolean sense = true
local_sense = true // thread-local variable

procedure barrier()
  local_sense = not local_sense
  if fetch_and_decrement(count) == 1
    count = n
    sense = local_sense
  else
  repeat
    // spin
  until sense == local_sense
```

**Allow other threads to proceed**

# Quiz: Barriers in Java

```
class Barrier {
    static CyclicBarrier barrier;
    static class Worker implements Runnable {
        public void run() {
            try {
                System.out.println("a");
                barrier.await();
                System.out.println("b");
                barrier.await();
            } catch (Exception e) { return; }
        }
    }
    public static void main(String[] args) {
        barrier = new CyclicBarrier(4);
        for (int i = 0; i < 4; i++) {
            new Thread(new Worker()).start();
        }
    }
}
```

*Please vote in Ilias.*

# Quiz: Barriers in Java

```
class Barrier {
    static CyclicBarrier barrier;
    static class Worker implements Runnable {
        public void run() {
            try {
                System.out.println("a");
                barrier.await();
                System.out.println("b");
                barrier.await();
            } catch (Exception e) { return; }
        }
    }
    public static void main(String[] args) {
        barrier = new CyclicBarrier(4);
        for (int i = 0; i < 4; i++) {
            new Thread(new Worker()).start();
        }
    }
}
```

**Only possible output: aaaabbbb**

*Please vote in Ilias.*

# Memory Consistency

- **When multiple locations are written concurrently, when do the writes become visible to other threads?**
- **Most programmers expect sequential consistency**
  - Each thread's instructions execute in the specified order
  - Shared memory behaves like a global array: Reads and writes are done immediately

# Relaxed Memory Models

- **In practice: Some reads and writes may occur "out of order"**
  - Ensuring sequential consistency: Inefficient
  - Instead, hardware and compilers reorder and delay some instructions
  - E.g., store into location that is not in CPU cache
    - Takes hundreds of cycles to complete
    - Processor completes it "in the background"
    - Loads on same core see it via write buffer

Initially: inspected = false
x = 0

**Core A**

1 inspected = true
2 xa = x

**Core B**

3 x = 1
4 ib = inspected

Under relaxed memory model:

xa = 0 and ib = false

Cores read old values

| Order of executed instructions under sequ. consistency | | | | Final values xa | ib |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 0 | true |
| 1 | 3 | 4 | 2 | 1 | true |
| 1 | 3 | 2 | 4 | 1 | true |
| 3 | 4 | 1 | 2 | 1 | false |
| 3 | 1 | 2 | 4 | 1 | true |
| 3 | 1 | 4 | 2 | 1 | true |

# Memory Models of PLs

- **Different hardware: Different reordering behavior**
- **PLs want to provide the same guarantees everywhere**
- **PLs defines their own memory model**
  - E.g., Java memory model or C11 memory model
  - PL implementation: Add fences, i.e., instructions to synchronize memory accesses

# Java Memory Model

- **By default, writes to shared objects are not immediately visible to other threads**
  - Other thread may read any old value
- **Enforce visibility by explicit synchronization**
  - Mark fields as `volatile`
  - Order write and read via `synchronized` block

# Example (Again)

```
class Warmup {
  static boolean flag = false;
  static void raiseFlag() {
    flag = true;
  }
  public static void main(String[] args)
      throws Exception {
    ForkJoinPool.commonPool()
      .execute(Warmup::raiseFlag);
    while (!flag) {};
    System.out.println(flag);
  }
}
```

**Code may hang forever, print `true`, or print `false`!**

# Quiz: Java Memory Model

## What may this Java code print?

```
final int[] a = {1,2};
Thread t1 = new Thread(new Runnable() {
    public void run() {
        synchronized(a) {
            a[0]++; a[1]++;
        }
    }
});
Thread t2 = new Thread(new Runnable() {
    public void run() {
        a[0]++; a[1]++;
    }
});
t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(a[0]+", "+a[1]);
```

*Please vote via Ilias.*

# Quiz: Java Memory Model

## What may this Java code print?

```
final int[] a = {1,2};
Thread t1 = new Thread(new Runnable() {
    public void run() {
        synchronized(a) {
            a[0]++; a[1]++;
        }
    }
});
Thread t2 = new Thread(new Runnable() {
    public void run() {
        a[0]++; a[1]++;
    }
});
t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(a[0]+", "+a[1]);
```

**Anything between 1, 2 and 3, 4 is possible: Access to `a` isn't properly synchronized.**

*Please vote via Ilias.*