Programming Paradigms

Concurrency (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Quiz: Java Concurrency

What does this Java code print?

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

Quiz: Java Concurrency

What does this Java code print?

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

`raiseFlag`: executed in concurrent thread

Quiz: Java Concurrency

What does this Java code print?

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

Shared variable accessed by two threads

Quiz: Java Concurrency

What does this Java code print?

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

Problem: No synchronization. Hence, main thread may read old value

Quiz: Java Concurrency

What does this Java code print?

```java
class Warmup {
  static boolean flag = true;
  static void raiseFlag() {
    flag = false;
  }
  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`!

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		T2
	L = false	
tas (L) ✓	L = true	
		tas (L) ✗
		tas (L) ✗
		tas (L) ✗
		tas (L) ✗
critical section { ⋮		
L = false		
		tas (L) ✓
	L = true	
		⋮ } critical section

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 next**
- **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");
                System.out.println("c");
                barrier.await();
            } catch (Exception e) { return; }
        }
    }
    public static void main(String[] args) {
        barrier = new CyclicBarrier(3);
        for (int i = 0; i < 3; i++) {
            new Thread(new Worker()).start();
        }
    }
```

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");
                System.out.println("c");
                barrier.await();
            } catch (Exception e) { return; }
        }
    }
    public static void main(String[] args) {
        barrier = new CyclicBarrier(3);
        for (int i = 0; i < 3; i++) {
            new Thread(new Worker()).start();
        }
    }
```

Subset of possible outputs: aaabbbccc, aaabcbcbc, aaabbccbc

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

Order of executed instructions under sequential 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

Under relaxed memory model:

xa = 0 ib = false

→ Cores read "old" values

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 = true;
  static void raiseFlag() {
    flag = false;
  }
  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`!

Overview

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

Synchronization Constructs in PLs

- **Various PL constructs to synchronize concurrent threads**
  - Monitors
  - Conditional critical regions
  - Synchronization in Java
  - Transactional memory
  - Implicit synchronization

Monitors

- **Object with operations, internal state, and condition variables**
  - Only one operation is active at any given time
  - Calls to a busy monitor:

    Delayed until monitor free
  - Operations may wait on a condition variable
  - Operations may signal a condition variable to allow others to resume

Example: Bounded Buffer

```
monitor bounded_buf
  buf : array [1..SIZE] of bdata
  next_full, next_empty : integer := 1, 1
  full_slots : integer := 0
  full_slot, empty_slot : condition

  fun insert(d : bdata)
    if full_slots = SIZE
      wait(empty_slot)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    full_slots +:= 1
    signal(full_slot)

  fun remove() : bdata
    if full_slots = 0
      wait(full_slot)
    d : bdata := buf[next_full]
    next_full := next_full mod SIZE + 1
    full_slots -:= 1
    signal(empty_slot)
    return d
```

Conditional Critical Regions

- **Syntactically delimited critical section**
  - Permitted to access a protected variable
  - Condition that must be true before entering the region
- **Syntax (pseudo code):**

```
region protected_var when condition do
  // ...
end region
```

Synchronization in Java

- **Every object can serve as a mutual exclusion lock**
- **`synchronized` keyword to acquire and release locks**
    - `synchronized` blocks: Define a critical section
    - `synchronized` methods: Entire method is a critical section

Demo

- **Synchronized.java**

Synchronization in Java (2)

- **Code in a critical section can**
  - ... wait for another thread:

```
while (!someCondition) {
  wait();
}
```

  - ... signal another thread that it can proceeed:

```
notify();
```

Synchronization in Java (2)

- **Code in a critical section can**
  - ... wait for another thread:

```
while (!someCondition) {
  wait();
}
```

Releases the lock and waits

  - ... signal another thread that it can proceeed:

```
notify();
```

Synchronization in Java (2)

- **Code in a critical section can**
  - ... wait for another thread:

    ```
    while (!someCondition) {
      wait();
    }
    ```

  - ... signal another thread that it can proceeed:

    ```
    notify();
    ```

Wakes up one of the threads that wait in a critical section with the same lock as that hold by the current thread

Synchronization in Java (2)

- **Code in a critical section can**
  - ... wait for another thread:

```
while (!someCondition) {
  wait();
}
```

  - ... signal another thread that it can proceeed:

```
notify();
```

While loop needed: Threads may also be woken up for spurious reasons or after a delay

Synchronization in Java (3)

- **Java memory model: Each Java thread may buffer or reorder its writes until**
  - ... it writes a `volatile` variable,
  - ... it releases a lock (e.g., leaves a `synchronized` block or `wait`s)
- **Must use some synchronization to ensure threads writes become visible**

Example

```
class Warmup {
  static boolean flag = true;
  static void raiseFlag() {
    flag = false;
  }
  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`!

Example

Fix: Make field `volatile`

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

Code will always print `false`

Transactional Memory

- **Atomicity without locks**

```
atomic {
  // critical section
}
```

- **PL implementation will**
  - ... speculatively execute the code block
  - ... check for conflicts, i.e., concurrent accesses to shared data
  - ... commit the results if no conflict
  - ... roll back (and try again later) otherwise

Implicit Synchronization

- **Compiler determines dependencies between concurrently executed code fragments**
  - Automatically add synchronization whenever needed
  - Parallelize independent code fragments
- **Extremely difficult in practice**
  - Auto-parallelization remains an open challenge

Quiz: Concurrency

Which of the following is true?

- PLs with a memory models are sequentially consistent.
- A relaxed memory model allows writes to be re-ordered.
- Test-and-set is used to implement spin locks.
- Client code of a monitor must use barriers to ensure correctness.

Quiz: Concurrency

Which of the following is true?

- ~~PLs with a memory models are sequentially consistent.~~
- A relaxed memory model allows writes to be re-ordered.
- Test-and-set is used to implement spin locks.
- ~~Client code of a monitor must use barriers to ensure correctness.~~

Overview

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