Programming Paradigms

Lecture 17:

Concurrency (Part 2)
Wake-up Exercise

What may this Java code print?

```java
final int[] a = {1, 2};
Thread t1 = new Thread(new Runnable() {
    public void run() {
        synchronized (a) {
            a[0]++; a[1]++;  
        }
    }
});
t1.start();

Thread t2 = new Thread(new Runnable() {
    public void run() {
        a[0]++; a[1]++;  
    }
});
t2.start();
t1.join();
t2.join();
System.out.println(a[0]+", "+a[1]);
```
Wake-up Exercise

What may this Java code print?

```java
final int[] a = {1, 2};
Thread t1 = new Thread(new Runnable() {
    public void run() {
        synchronized (a) {
            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.

https://ilias3.uni-stuttgart.de/vote/0ZT9
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
def test_and_set(L):
    # Pseudo code
    while not test_and_set(L):
        # nothing (spin)
```

Problem: Repeated writes when lock is already acquired harms performance ("contention")
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:

  ```java
  // 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
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

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
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
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();
        }
    }
}

What are the possible outputs of this code?
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

https://ilias3.uni-stuttgart.de/vote/0ZT9
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: \( \text{inspected} = \text{false} \)
\[ x = 0 \]

**Core A:**
1. \( \text{inspected} = \text{true} \)
2. \( x_a = x \)

**Order of executed instructions**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**Core B:**
3. \( x = 1 \)
4. \( i_b = \text{inspected} \)

**Final values**

<table>
<thead>
<tr>
<th>( x_a )</th>
<th>( i_b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
</tr>
</tbody>
</table>

**Under relaxed memory model:**
\( x_a = 0 \) and \( i_b = \text{false} \)
 neste read old values

**Under sequential consistency**
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

```java
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!
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

```plaintext
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

entry 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)

entry 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):

```plaintext
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:
    ```java
    while (!someCondition) {
      wait();
    }
    ```
  - ... signal another thread that it can proceed:
    ```java
    notify();
    ```
Synchronization in Java (2)

Code in a critical section can

- wait for another thread:

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

- signal another thread that it can proceed:

```java
notify();
```

Releases the lock and waits
Synchronization in Java (2)

- Code in a critical section can
  - ... wait for another thread:
    ```java
    while (!someCondition) {
        wait();
    }
    ```
  - ... signal another thread that it can proceed:
    ```java
    notify();
    ```

Wakes up all 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:
    ```java
    while (!someCondition) {
      wait();
    }
    ```
  - ... signal another thread that it can proceed:
    ```java
    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 waits)

- **Must use some synchronization to ensure threads writes become visible**
Example

```java
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!
Example

```java
class Warmup {
    static volatile 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);
    }
}
```

Fix: Make field `volatile`

Code will always print `true`
Transactional Memory

- **Atomicity without locks**
  ```c
  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?

■ Barriers are a form of busy-wait synchronization.

■ Memory models specify that the PL is sequentially consistent.

■ A conditional critical region can emit and receive signals by other threads.

■ Writes to fields are always visible to other threads in Java.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Concurrency

Which of the following is true?

- Barriers are a form of busy-wait synchronization.
- Memory models specify that the PL is sequentially consistent.
- A conditional critical region can emit and receive signals by other threads.
- Writes to fields are always visible to other threads in Java.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

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