Programming Paradigms

Concurrency (Part 4)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

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

  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):**

```
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 = 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

Fix: Make field `volatile`

```
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);
  }
}
```

Code will always print `true`

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?

- 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.

Please vote via Ilias.

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.~~

Please vote via Ilias.

Overview

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