Concurrency And Its Challenges

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering

AMRITA VISHWA VIDYAPEETHAM

Jul - Dec 2020

Contents

1. Limits of functional verification
2. Concurrent systems
3. Concurrent programming
4. Concurrent programming with Java threads
5. Challenges in concurrent systems
6. Effects of interleaving
7. Shared access to data: An Example in Java
8. Broad goals of concurrent systems

Limits of functional verification

Some programs are supposed to run forever.

- Embedded systems
- Real-time systems
- Event driven applications

Termination is a bug!

Digital clock

```
while( true ) {
    Update hh:mm:ss
}
```

Traffic signal

```
while( true ) {
    Red    → Orange
    Orange → Green
    Green  → Red
}
```

Search engine

```
while( true ) {
    Receive keywords
    Perform search
    Send search results
}
```

Will input-output style of verification work for these scenarios?

In concurrent systems, computations are carried out simultaneously.

- Computations happen in overlapped time periods.
- Two or more tasks make progress simultaneously.
- Some from of interdependency between tasks exists.
- Physically there may be a single processor only.

Term	What it means	Remarks
Sequential	1111111111 222222222	Task 1 followed by Task 2
Parallel	222222222 1111111111	Tasks 1 and 2 on physically different processors
Concurrent	111 22 11 2222 111 222 11	Tasks 1 and 2 interleaved (single processor mostly)*

*Parallel also means concurrent but not vice-versa.

Concurrent programming

Most programming languages support concurrent programming through threads.

- A program can have multiple threads.
- Each thread has its own execution flow.
- Threads can run independent of each other.
- Only local variables are separate for each thread.
- Rest of the data are accessed and modified by all threads.
- Sequential programs can be viewed as programs with one thread (main).
- Threads can be created dynamically or a pool of threads can be maintained.
- Threads can be created using constructs provided by programming languages.
- A thread scheduler allocates CPU to each thread for certain time slices in a round-robin fashion.

Java supports basic to sophisticated threading libraries for concurrent programming. We will look at the basic library.

1. Go to https://swaminathanj.github.io/oop/26_Multithreading.html.
2. Try the first program on creating threads.
   - Glance through the source code and get a high-level understanding.
   - Create a project in Eclipse and add Main.java & NewThread.java to it.
   - Run the program once and check the console output.
   - Run the program 3 or 4 times and observe the output each time.
   - Create second instance of NewThread in main() and run.
3. Some notes about the code.
   - The program has 2 threads: Main and NewThread.
   - Although Main creates NewThread instance, both run independently.
   - The local variable i in Main's main() method is different from the local variable i of NewThread's run() method.
   - nt.start() puts the thread instance nt in ready state. The thread scheduler decides when to run().

Challenges in concurrent systems

Concurrent systems throw some inherent challenges to programming.

1. Thread scheduling is not our control. Leads to race conditions.
   - Any interleaving is possible during an execution.
   - Each execution results in a different interleaving.
     - Behavior is unpredictable.
     - Difficult to debug or reproduce bugs.
2. Threads share access to common data.
   - Simultaneous access to shared data causes incorrect results.

   Synchronization constructs are used to mitigate "undesirable" interleavings and "incorrect" access to a good extent, provided they are used properly.
   - But this can give rise to problems such as deadlock or starvation.
3. Interleaving complexity is exponential.
   - Testing gets exponentially harder compared to sequential programs.
     - Path complexity compunded by interleaving complexity.
4. Most concurrent systems are non-terminating.
   - Functional verification is not practical.

Effects of interleaving

Let's study the effects of interleaving with a simple example.

- Let x be the shared variable with initial value 0.

Thread 1	Thread 2
read x x = x + 1	read x x = x − 1

Possible interleavings	Result
1. read x, x = x + 1, read x, x = x − 1	x = 0, 1, 0 ✓
2. read x, x = x − 1, read x, x = x + 1	x = 0, -1, 0 ✓
3. read x, read x, x = x − 1, x = x + 1	x = 0, -1, 1 X
4. read x, read x, x = x + 1, x = x − 1	x = 0, 1, -1 X
5. read x, read x, x = x + 1, x = x − 1	x = 0, 1, -1 X
6. read x, read x, x = x − 1, x = x + 1	x = 0, -1, 1 X

(1) Number of possible interleavings increases exponentially as number of threads or statements within threads increase. (2) Some interleavings produce incorrect results.

Shared access to data: An Example in Java

A counter that is incremented and decremented by different threads.

1. Go to https://swaminathanj.github.io/oop/26_Multithreading.html.
2. Try the third program on counter - increment & decrement by different threads.
   - Create a project in Eclipse and add Counter.java, Incrementer.java, Decrementer.java & CounterTest.java to it.
   - Replace while(true) with for (int i=0;i<10000;i++) in both Incrementer and Decrementer threads.
   - Run the program multiple times. Check if the counter value gets a value of 1 or -1 finally (instead of 0) in some run. Under extremely rare circumstances, it might happen.
   - Introduce synchronized keyword as given in the description. Now, for no run, you should see the above issue.
3. You can also try the next example on Producer-Consumer problem.
   - Observe the effect of with and without synchronized.
   - The keyword synchronized enforces mutual exclusion.
   - The condition variable flag and the methods wait() & notify() ensures both threads take turns in tight way.

Broad goals of concurrent systems

In the light of above discussion, lets define the goals of concurrent programs.

1. Safety: Nothing bad will happen.
   - In sequential programs, safety implies final state is correct.
   - In concurrent programs, safety implies the system does not get into unsafe state at any point during the execution.
2. Liveness: Something good will happen eventually.
   - In sequential programs, liveness implies no infinite loops and program terminates.
   - In concurrent programs, liveness implies no deadlocks or program does not block.
3. Fariness: All get to progress.
   - In sequential programs, the concept of fairness is not applicable.
   - In concurrent programs, fairness implies no unbounded wait or starvation.