

Concurrency And Its Challenges

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering



Jul - Dec 2020

- 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 An Example in Java
- 8 Broad goals of concurrent systems

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 form of interdependency between tasks exists.
- Physically there may be a single processor only.

Term	What it means	Remarks
Sequential	1111111111 2222222222	Task 1 followed by Task 2
Parallel	2222222222 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.

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.

- ❶ Go to https://swaminathanj.github.io/oop/26_Multithreading.html.
- ❷ 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.
- ❸ 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()**.

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 data (except local variables).

- Simultaneous access to shared data causes incorrect results.

Synchronization constructs are used to mitigate "undesirable" interleavings and "inappropriate" access to a good extent, provided they are used properly.

③ Interleaving complexity is exponential.

- Testing gets exponentially harder compared to sequential programs.
- **Path complexity x interleaving complexity \Rightarrow Unimaginable!**

④ Most concurrent systems are **non-terminating**.

- Functional verification is not practical.

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

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

Thread 1

```
read x
x = x + 1
```

Thread 2

```
read x
x = x - 1
```

Possible interleavings

- 1 read x , $x = x + 1$, read x , $x = x - 1$
- 2 read x , $x = x - 1$, read x , $x = x + 1$
- 3 read x , read x , $x = x - 1$, $x = x + 1$
- 4 read x , read x , $x = x + 1$, $x = x - 1$
- 5 read x , read x , $x = x + 1$, $x = x - 1$
- 6 read x , read x , $x = x - 1$, $x = x + 1$

Result

- $x = 0, 1, 0$ ✓
- $x = 0, -1, 0$ ✓
- $x = 0, -1, 1$ ✗
- $x = 0, 1, -1$ ✗
- $x = 0, 1, -1$ ✗
- $x = 0, -1, 1$ ✗

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

A counter that is incremented and decremented by different threads.

- ➊ Go to https://swaminathanj.github.io/oop/26_Multithreading.html.
- ➋ Try the third program on counter - increment & decrement by different threads.
 - Glance through the source code and get a high-level understanding.
 - Create a project in **Eclipse** and add **Counter.java**, **Incrementer.java**, **Decrementer.java** & **CounterTest.java** to it.
 - **Run** the program multiple times. Check if the counter value between two adjacent prints is > 1 in some run. You may or may not get it.
 - Introduce **synchronized** keyword as given in the description. Now, for no run, you should see the above issue.
- ➌ 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**. i.e. only one method in Counter can be active at any instant.

In the light of above discussion, let's define the goals of concurrent programs.

1. **Safety**: Nothing bad will not 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 execution.

2. **Liveness**: Something good will happen eventually.

- In sequential programs, liveness implies **no infinite loops** and **program terminates**.
- In concurrent programs, liveness implies the program is not deadlocked and progresses forward.

3. **Fairness**: All get to progress.

- In sequential programs, the concept of fairness is not applicable.
- In concurrent programs, fairness implies no unbounded wait or starvation.