JAVA CONCURRENCY FRAMEWORK

Kaushik Kanetkar

Old days

One CPU, executing one single program at a time

No overlap of work/processes

Lots of slack time

CPU not completely utilized

What is Concurrency

Concurrency is the ability to run several parts of a program or several programs in parallel.

Concurrency can highly improve the throughput of a program if certain tasks can be performed asynchronously or in parallel.

NEED FOR CONCURRENCY

- Better resource utilization
- Simpler program design
- More responsive programs

Better resource utilization

Imagine an application that reads and processes files from the local file system. Lets say that reading of file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

5 seconds reading file A
2 seconds processing file A
5 seconds reading file B
2 seconds processing file B
---------- 14 seconds total

Most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. So, by better utilization:

5 seconds reading file A
5 seconds reading file B + 2 seconds processing file A
2 seconds processing file B
---------- 12 seconds total

Hence, concurrency gives a faster performance.

Simpler program design

- Single threaded application, you would have to keep track of both the read and processing state of each file.
- Multi-threaded: you can start two threads that each just reads and processes a single file.

    - Reading / processing from 10 files using single thread : 10 such functions + files.

    - Reading / processing from 10 files using concurrency: 1 such function where 10 threads can act on it.

More responsive programs

Imagine a server application that listens on some port for incoming requests. when a request is received, it handles the request and then goes back to listening. The server loop is sketched below:

```
while(server is active)
{
 listen for request process
request
}
```

If the request takes a long time to process, no new clients can send requests to the server for that duration. Only while the server is listening can requests be received.

An alternate design would be for the listening thread to pass the request to a **worker thread**, and return to listening immediately. The worker thread will process the request and send a reply to the client.

```
while(server is active)
{
 listen for request process
request
 hand request to worker
thread }
```

AMDAHL'S LAW

- Performance gain using Amdahl's Law:

If F is the percentage of the program which can not run in parallel and N is the number of processes then the maximum performance gain is

$$1/ (F+ ((1-F)/n)).$$

Issues with concurrency

- More complex design

Though some parts of a multithreaded applications is simpler than a singlethreaded application, other parts are more complex. Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

Issues with concurrency

- Context switching overhead

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute.

Context switching is not cheap

Issues with concurrency

- Increased resource comsumption

A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread.

The framework:

- **Task Scheduling**: The Executor is a framework for handling the invocation , scheduling and execution of tasks.
- **Concurrency**: Using classes like map, lists and queues.
- **Atomic variables**: Classes for atomic manipulation of single variables provide higher performance.
- **Locking**: Implementing locking mechanism using the sychronized keyword.
- **Timers**: Accurate timing measurements upto nanoseconds usually for timeouts.

Advantages of the framework:

- **Reusability:** Many commonly used classes are implemented.
- **Better Performance**: A highly optimized approach with faster responses.
- **Higher reliability**: With all locks, and synchronizing mechanisms, it is highly reliable.
- **Maintainability and scalability**: The programs are easy to handle and maintain for further development.
- **Better productivity**: Easier to debug.

PROCESSES VS THREADS

- Process: A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.

- Threads: threads are so called lightweight processes which have their own call stack but an access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data

Creating and Starting Threads

```
Thread thread = new Thread();  // Creates a thread
thread.start();                // Starts the thread
```

Ways to point to the code for a thread to start:

- **Thread Subclass**
- **Runnable Implementation**

Thread Subclass

To create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start().

```
public class MyThread extends Thread
{ public void run()
{ System.out.println("MyThread running"); } }
```

To create and start the above thread:

```
MyThread myThread = new MyThread();
myThread.start();
```

Runnable Implementation

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread.

```
public class MyRunnable implements Runnable
{ public void run()
{ System.out.println("MyRunnable running"); } }
```

To create and start the above thread:

```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Subclass or Runnable method ?

- Both methods work.
- Runnable method preferred .
- Easy to queue up the Runnable instances (from the thread pool) until a thread from the pool is idle. This is a little harder to do with Thread subclasses.

Common pitfall:
Calling run() instead of start()

```
Thread newThread = new Thread(MyRunnable());
thread.run();                                  //should be start();
```

- run() method is executed by the thread that created the thread.
- So, necessary to call start() and not run()

Race Conditions and Critical sections

- More than one thread writing to shared resources.
- ‘A’ thread reads from shared “data” , while B does some writing on the “data”. Now when ‘A’ reads again from “data” it has corrupt data in hand.
- Code section that leads to race conditions is called a critical section

Thread safety and shared resources

- Code that is safe to call my multiple threads simultaneously is thread safe.
- Thread safe code cause no race conditions.
- Multiple threads updating shared resources leads to race conditions.

Race Conditions and Critical sections

- **Local variables:** Stored in stack of each thread. So these are thread safe !

```
public void someMethod()
{
 long threadSafeInt = 0;
 threadSafeInt++;
 }
```

Race Conditions and Critical sections

- **Local Object references:** Stored in stack of each thread. Although stored in the “shared” heap, it is also thread safe !

```
public void someMethod()
{
 LocalObject localObject = new LocalObject();
localObject.callMethod();
method2(localObject);
 }
public void method2(LocalObject localObject)
{
 localObject.setValue("value");
 }
```

Race Conditions and Critical sections

- **Object members:** Stored in heap alongwith other objects. Two threads calling a method on the same object and updating it- Not Thread safe !

```
public class NotThreadSafe
{ StringBuilder builder = new StringBuilder();
public add(String text)
{ this.builder.append(text);
}}
```

If two threads call the add() method simultanously **on the same NotThreadSafe instance** then it leads to race conditions

Synchronized blocks

- Block of methods or blocks of code can be marked as synchronized in order to avoid race conditions.

- Following is a synchronized block:

```
public void add(int value)
{ synchronized(this)
{
this.count += value;
}
}
```

Synchronized blocks

- Synchronized construct takes an object in parantheses.
- Object taken in parantheses in called as the monitor object.
- Code synchronized on the monitor object.
- Only one thread can execute inside a code block synchronized on the same monitor object.

Thread signaling

- The purpose of thread signaling is to enable threads to send signals to each other indicating data is ready.

1. Signaling via Shared objects

A simple way for threads to send signals to each other is by setting the signal values in some shared object variable.

2. Busy wait

A thread which is to process the data is waiting for data to become available for processing. When a certain function returns true, it can then proceed.

Thread signaling

3. Wait(), notify(), notifyAll()

The waiting thread with wait() becomes inactive until it receives a notify() call. (Better CPU utilization)

4. Missed signals

A call to notify() before a call to wait() results in missed calls. Such calls need to be stored for the system to know.

5. Spurious wakeups

Threads may wake up without any notify(). Guarding required to avoid such wakeups.

Thread signaling

6. Multiple threads waiting for same signals
Many threads waiting for notify() call but only one should be passed through.

7. Don’t call wait() on constant objects
A wait() on the first instance of an object may risk being woken up by the notify() on some other constant object.

Deadblocks

- A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding.

Thread 1 locks A, waits for B Thread 2 locks B, waits for A

- A needs a unlock from B , but be is itself locked. B needs unlock from A, but A is also locked !!

Deadblocks prevention

1. **Lock Ordering**

Multiple threads needing locks but obtaining in different order.
Make sure the locks are obtained in the same order.

2. Lock Timeout

Thread attempting to obtain a lock will wait only a certain amount of time and then give up.

Deadblocks prevention

3. Deadlock Detection

Used wherever lock ordering is not possible and lock timeout is not feasible.

Thread obtaining a lock is noted in a data structure of threads and locks.

Can detect if a deadlock can happen !

Semaphores

A thread synchronization construct that can be used either to send signals between threads to avoid **missed signals**, or to guard a **critical section** like you would with a **lock**.

1. **Simple semaphore**

Take() method makes signal=true and calls to notify().
Release() method will call wait() only if take() has been called.

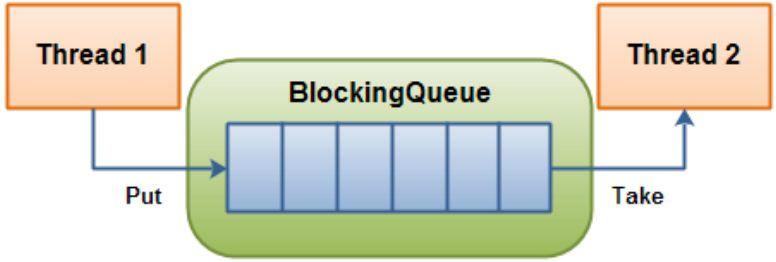
Semaphores

2. Using semaphores for signaling.
Take() method makes signal=true and calls to notify().
Release() method will call wait() only if take() has been called.

3. Counting Semaphore
Counts the number of signals sent using the number of take() calls.

Blocking Queue

A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue.



A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue

Thread pools

- Thread Pools are useful when you need to limit the number of threads running in your application at the same time.
- Performance overhead in creating a new thread with respect to stack.
- Instead of starting a new thread, the task can be passed on the thread pool.
- An idle thread from the pool will pick up the task.
- Used in multi-threaded servers. The threads in the thread pool will process the requests concurrently.

Code walkthrough:

- Following is a code which creates 10 threads and runs them.
- One can see which thread is running currently.
- It is done using the getName() method call.
- When you see the output , you will realize that even “main” is a thread.

```
public class ThreadExample
{
 public static void main(String[] args){
System.out.println(Thread.currentThread().getName()
);
for(int i=0; i<10; i++)
{ new Thread("" + i)
{ public void run()
{ System.out.println("Thread: " + getName() + "
running");
 }
 }.start();
 }
 } }
```

Output:

main
thread: 0 running
thread: 1 running
thread: 2 running
thread: 3 running
thread: 4 running
thread: 5 running
thread: 6 running
thread: 7 running
thread: 8 running
thread: 9 running

If the start() calls to thread are commented out, then none of the threads will run , next slide ->

```
public class ThreadExample
{
 public static void main(String[] args){
System.out.println(Thread.currentThread().getName()
);
for(int i=0; i<10; i++)
{ new Thread("" + i)
{ public void run()
{ System.out.println("Thread: " + getName() + "
running");
 } };
//.start();
 }
 } }
```

Output:

main

Thus only the main thread is running and none of the 10 threads have been given a start().