CSE 223: Programming -2
12-Concurrency Design Patterns

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Concurrency Patterns

- These patterns involve coordinating concurrent operations and address two primary problems
  - Shared resources
  - Sequence of operations
- List
  - Single threaded execution
  - Guarded suspension
  - Balking
  - Scheduler
  - Read/Write lock
  - Producer-consumer
  - Two-phase termination

Threads & Synchronization

Threads & Synchronization

- Multi-threaded programming is a conceptual paradigm for programming, where programs are divided into two or more *light-weight* processes (*threads*) which can be run in '**parallel**'.
- In traditional single-thread system, event loop with *polling* is used. In these systems, a single thread of control sits in an *infinite* loop, polling a single event queue to decide what to do next. Until the *event handler* returns, nothing else can happen in the system.
- In *multi-threading*, if a task can be described as independent thread, we can simplify the programming and get more CPU utilization by automatically switching from one thread (that is going to wait) to another ready to run one.

Threads & Synchronization

- In order to have real multi-threading programming, the operating system must support threading.
  - Note: Multitasking OS may not support threading (each task is considered as one thread).
- Java provides full support of threading and allows the entire environment to be asynchronous.
- Java runtime utilizes priorities (1-10) to specify the priority of one thread relative to another. The highest-priority thread that is ready to run is always given the CPU and my *preempt* a lower priority thread.
- *Synchronization and mutual exclusion* for shared resources is implemented using a variant of the monitor model.
  - Each object has its own implicit monitor that can be entered by calling one of the object's *synchronized* methods. Once inside the monitor, no other thread can call any synchronized method on the same object. Other threads have to wait (*blocked*) until some other thread explicitly *notifies* it to come out.

Threads & Synchronization

- Thread class encapsulates all of the control one needs over threads.
  - There are methods on a Thread object that control whether the thread is
    - Running
    - Sleeping
    - Suspended
    - Stopped
  - The static method Thread.currentThread() always returns a handle for the current thread.
- Runnable interface
  - When a new instance of Thread is created, we need to tell it what code to run inside the new thread of control.
  - A thread can start on any object that implements the *Runnable* interface.
  - The Runnable interface has only single method called "*run*" which is the start code of the associated thread.

Threads & Synchronization

```
class ThreadDemo implements Runnable{
    ThreadDemo(){
     Thread ct=Thread.currentThread();
     System.out.println("current Thread: "+ct);
     Thread t=new Thread(this,"Demo Thread
     System.out.println("Thread created: " + t);
     t.start();
     try{ Thread.sleep(3000);
     } catch(InterruptedException e){
          System.out.println("interrupted");}
     System.out.println("exiting main thread");
    }
    public void run(){
     try{ for (int i= 5; i>0; i--) {
          System.out.println(" "+ i );
          Thread.sleep(1000);}
     } catch(InterruptedException e){ System.out.println("interrupted");}
     System.out.println("exiting child thread");
    }
    public static void main(String args[]){
     new ThreadDemo();
    }
}
```

```
current Thread:Thread[main,5,main]
Thread created:Thread[Demo Thread,5,main]
          5
          4
          3
exiting main thread
          2
          1
exiting child thread
```

One possible output

Synchronization

```
class Callme{
        void call(String msg){
                System.out.print('[" + msg);
                try Thread.sleep(1000); catch(Exception e);
                System.out.println("]");
                }
}
class Caller implements Runnable{
        String msg;
        Callme target;
        public Caller(Callme t, String s){
                target = t;
                msg = s;
                new Thread(this).start();
        }
        public void run(){ target.call(msg);}
}
class Synch{
        public static void main(String args[]) {
        Callme target = new Callme();
        new Caller(target,"Hello");
        new Caller(target,"Synch");
        new Caller(target,"World");
        }
}
```

[Hello[Synch[World]
]
]

The synchronized Statement

- To synch objects that was not designed for multithread access, we can use the synchronized statement.

  synchronized (object) statement;

- Example:
  - the previous example could be synchronized by leaving the call method unsynchronized and use the synchronized statement inside Caller's run method instead.

```
public void run(){
        synchronized(target){
        target.call(msg);
        }
}
```

Inter-thread Communication

- Communication & synch between threads are done in Java using wait, notify and notifyAll methods that are implemented as final methods in Object.
- These methods must only be called from inside of synchronized methods.
  - **wait**: forces the current thread to give up the monitor and go to sleep.
  - **notify**: wakes up the first thread that called wait on the same object.
  - **notifyAll**: wakes up all threads that are waiting on the same object. The highest priority thread that wakes up will run first.

Single threaded execution

Single threaded execution

- Intent
  - To prevent concurrent access to data or other resources → Critical Section.
- Synopsis
  - Synchronize on a same object the portions of code that should not be concurrently executed.
- Example
  - Several threads set and get a datum held by an object. The time of an access is non deteministic.
- Solution

```
class HoldDatum {
    private Datum datum;
    public synchronized void set (Datum d) { datum = d; }
    public synchronized Datum get () { return datum; }
}
```

  - A resource such as an I/O stream may be accessed by a single synchronized method.

Guarded Suspension

Guarded Suspension

- Intention
  - A method should wait to execute until a precondition holds.
- Synopsis
  - Use an object's methods wait and notify.
- Example
  - A queue is used by two threads to exchange objects. A thread can take an object from the queue only if the queue contains elements.
- Solution

```
synchronized void guardedMethod () {
    while (! precondition) { wait (); }
        ... // code requiring precondition
}
```

  - Any method that might change the precondition looks like:

```
synchronized void stateChangingMethod () {
    ... // code possibly changing precondition
    notify ();
}
```

- Both methods are synchronized on the same object.
- See also54
  - Balking (is similar to **guardedMethod**, but it returns rather than waiting)

Balking

Balking

- Intention
  - Disregard the call to a method of an object, if the state of the object is not appropriate to execute the call.
- Synopsis
  - Test the object's state, in a synchronized block for thread safety, and return if the state is inappropriate.
- Example
  - A toilet with an automatic flusher should not begin a flush cycle when another cycle is in progress.
- Solution

```
void balkingMethod () {
    synchronized (this) {
        if (! some_condition) return;
        else
            ... // code possibly changing some_condition
    }
    ... // do whatever
}
```

  - The synchronized block ensures thread safety.
- See also:
  - Guarded Suspension → it waits rather than returning

Scheduler

Scheduler

- Intention
  - implement a general purpose thread scheduling policy.
- Synopsis
  - Use two classes: a general **Scheduler**, which assign a resource to a thread, and a specialized **ScheduleOrdering**, which selects one thread among many to execute.
- Example
  - Entries to log are queued while waiting to be printed. Queueing minimizes the chance of printing entries in the wrong termporal order.
- Solution
  - **Request**: implements **SchedulingOrdering**. Objects encapsulate a request for a Processor
  - **Processor** performs the computation abstracted by a Request. It asks the Scheduler the ok to process. It informs the scheduler when the process is completed.
  - **Scheduler** schedules Requests to the Processor. It relies on interface ScheduleOrdering, implemented by each Request, for the ordering.

Scheduler

- **ScheduleOrdering** Interface used to decide the order in which Requests a processed. Keeps the Scheduler independent of the details of a Request.

- See also
  - ReadWriteLock (is a specialized form of Scheduler)

- **Processor**

```
void process (Request r) {
    scheduler.enter (r);
    ... // process this request
    scheduler.done ();
}
```

- **Scheduler** defines a variable shared by its enter and done methods

```
Thread runningThread;
```

Scheduler

- The **Scheduler's** enter method looks like

```
void enter (Request r) {
    Thread thisThread = Thread.currentThread ();
    synchronized (this) {
        if (runningThread == null) {    // processor is free
            runningThread = thisThread;
            return;
        }
        ... // place thisThread in a waiting list;
    }
    synchronized (thisThread) {
        while (thisThread != runningThread)
            thisThread.wait();
    }
    synchronized (this) {
        remove thisThread from the waiting list;
    }
}
```

- The **Scheduler's** enter method looks like

```
void done () {
    synchronized (this) {
        if (there are no waiting threads) {
            runningThread = null;
            return;
        }
        runningThread = first thread in waiting list;
    }
    synchronized (runningThread) {
        runningThread.notifyAll ();
    }
}
```

Read/Write Lock

Read/Write Lock

- Intention
  - To allow concurrent read access to an object, but exclusive write access.
- Synopsis
  - Use a specialized Scheduler whose method enter is replaced by two methods: readLock and writeLock.
- Example
  - An on-line auction system where remote users get and set bids.

Read/Write Lock

- Method *readLock* looks like:

```
void synchronized readLock () {
    while (pendingWriters + activeWriters > 0) wait ();
    activeReaders++;      // being reading
}
```

- Method *writeLock* looks like:

```
void synchronized writeLock () {
    pendingWriters++;     // block readers
    while (activeReaders + activeWriters > 0) wait ();
    pendingWriters--;
    activeWriters++;      // being writing
}
```

- Method *done* looks like:

```
void synchronized done () {
    if (activeWriters > 0)  activeWriters--;
    else if (activeReaders > 0)  activeReaders--;
    else error ();
    notifyAll ();
}
```

- Note that *activeWriters* is either 0 or 1.

Producer-Consumer

Producer-Consumer

- Intention
  - To coordinate the asynchronous production and consumption of information.
- Synopsis
  - Producers and consumers exchange information via a queue. The code to pull information from the queue is guarded.
- Example
  - A trouble-ticket dispatching system receives tickets from clients.
  - Dispatchers pull tickets and forward them to the appropriate troubleshooter.
- Solution
  - **Producer** supplies objects representing the information used by the Consumer and places them in the Queue. Production and consumption of objects are asynchronous.
  - **Queue** holds produced objects that cannot be consumed immediately.
  - **Consumer** pulls from the Queue and uses the object produced by the Producer. If the Queue is empty it waits.

Producer-Consumer

- See also:
  - GuardedSuspension (it is used by this pattern)
  - Filter (it is a simple form of this pattern)
  - Scheduler (this pattern is a special case of scheduling)
- The **producer** is a thread whose run methods looks like:

```
public void run () {
    for (;;) {
        Object object = ... // produce an object
        synchronized (queue) {
            queue.enqueue (object);
            queue.notify ();
        }
    }
}
```

- The **consumer** is a thread whose run methods looks like:

```
public void run () {
    for (;;) {
        synchronized (queue) {
            while (queue.isEmpty ()) { queue.wait (); }
            Object object = queue.dequeue ();
        }
        ... // consume the object
    }
}
```

Two-Phase Termination

Two-Phase Termination

- Intention
  - To orderly shutdown a thread.
- Synopsis
  - Set a latch that a thread checks at strategic points of its execution.
- Example
  - A server that assigns a thread to each client must handle requests to terminate a client's thread or its own execution.
- Solution

```
public void run () {
    initialize ();
    while (! isInterrupted ()) {
        ... // whatever execution
    }
    shutdown ();
}
```

Method *isInterrupted* tests the interrupted status of a thread. The interrupted status is set when a thread's method interrupt is called and is cleared in various situations.