Operating Systems and Networks

Lecture 08: Threads in Java (continue)

InterruptedException in run method

Example:
The printer CoolPrinter which reads files from a PrintSpooler (a queue of documents). The CoolPrinter has two mode of operation the normal mode and repair mode. ServcingManager periodically stops the CoolPrinter for checking and servicing. Create coolPrinter and run in on a thread. Let servicingManger interrupt it for the repair. (see coolPrinter folder)

Important part of the code

```
public void run(){
    boolean isInRepairMode = false;
    while(!isInRepairMode){
        ... try {
            KeepPrinting();
        } catch (InterruptedException e) {
            isInRepairMode = true;
            Reparing();
        }
    }
}
```

InterruptedException in run method

- This pattern, known as Guarded Block, with a boolean variable to control stopping, is very common in properly written run methods, and you should learn it.
- Note how we don't need to call interrupted() any more. Calling isInRepairMode on every repetition allows the method to find out if the interrupt status is set.

InterruptedException in other methods

- When you call sleep in methods other than run, quite often you don't know how to stop the thread - because you don't know what threads might call the method.
- In the situation where the InterruptedException arises, ask yourself, Can the method still perform its usual task?
- If not, then the method should throw an exception - that is what exceptions are for. Usually you just declare the method as throwing InterruptedException, and then you don't need a try-catch round Thread.sleep().

InterruptedException in other methods

- If the method can successfully perform its usual task, despite the InterruptedException, then it is usually more convenient not to rethrow the exception.
- If you don't, you must make sure the interrupt status is set again. This is so that when control returns to the run method it can detect the interrupt.
InterruptedException in other methods

- Use the following code.
  ```java
  try {
    Thread.sleep(...);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
  ```

  Note: The static method
  Thread.currentThread() returns a reference to the thread currently executing.

Deprecated methods and stop()

- Thread class has a method stop() that forces the thread to stop executing.
- stop() is deprecated as it is unsafe. While a thread is updating an object, it may temporarily put the object in a bad state with its invariants broken.
- There is no safe way to force a thread to stop.

Many Thread in action

- Threads consume a lot of system resources:
  - Each thread uses memory
  - Creating and destroying threads takes a lot of time.
  - This can be a big overhead if you have lots of small tasks that need to be run on separate threads.
  - Therefore it is desirable to be able to limit the total number of threads, and to maximize the use of each thread once it has been created.

Example: Multi-threaded client/server

Thread pools

- A thread pool keeps a fixed set of threads that are available for running a succession of short tasks.
- Each thread in the pool is presented as a run- object (i.e. an instance of a class that implements Runnable)
- Think of the threads in the pool as being like counter staff in a post office, and the run-objects as being like customers, each with its own task to perform.
- The threads take on any tasks that come along, and when all the threads are busy the tasks wait in a queue.

Thread pools via java.util.concurrent

- A thread pool with nThreads threads in it, use this static factory method from java.util.concurrent.Executors:

  ```java
  public static ExecutorService newFixedThreadPool(int nThreads)
  ```

  which results in a Thread pool. Then you can call many methods, for example

  ```java
  void execute(Runnable command)
  ```

  "command" is task to be submitted to the pool.
Thread pools via java.util.concurrent (continue)
If any threads are available to run it, one of them does so. Otherwise the task waits in a queue until a thread is free. Throws RejectedExecutionException if the thread is shut down. void shutdown() shuts the pool down. No new tasks will be accepted, though tasks still being run, or in the queue, will be completed.

Example: 3 waiters in a busy café
Four groups of customers have just arrived in a restaurant, sitting on four tables. There are two waiters. Each table spend up to 10 min (OK! to speed up 10 sec) to decide and order their food. Using Thread pool write a program that simulates the two waiters serving a four tables. ((see the folder restaurant for a solution))

Interacting threads
Aim: study threads which interact with each other by working on the same data, ensuring they don’t modify the same piece of data simultaneously.
Mutual exclusion
Let us see it with the help of an example

Flight booking system
Consider a flight with 10 seats and a number of travel agents each trying to book two seats on the flight.
1) Write a class Flight with two integer value attributes for total number of seats and number of booked seats (make them static to simplify)
2) Write a class TravelAgent which is implemented as a thread. TravelAgent checks if there are two seats available and books them.
3) In Flight, create 10 TravelAgent and let them book. At the end make sure that all threads join().
4) Print the number of booked seats. (see folder flight_travelagent_01 for a solution)

Flight booking system (ctd)
- Although each travel agent checks for availability of seat, we end up with too many seats booked.
- Tidy up the code by refactoring the run() method of TravelAgent by extracting method. Then include some print statement (or use a debugger) to explore why. (see folder flight_travelagent_2)

Challenges of concurrency
- The problem is known as a race condition or data race.
How to achieve mutual exclusion:
- Lock variables
- Strict Alternation
- sleep and wake up
- Semaphore
- Monitors
**Race condition**

- The problem is known as a *race condition* or *data race*.
- It is caused because threads/processes can access shared resources (shared variables, files, …)
- The key to avoiding race condition is to ensure *mutual exclusion*.
- Mutual exclusion: no two thread can access the *critical region* simultaneously.
- Critical region (also *section*) part of the program where shared memory is accessed.

**Question:** what is the critical section in case of the Flight example?

**Race condition (ctd)**

Mutual exclusion solves the race condition problem. But to have concurrent programs with correct behaviour the following conditions must satisfy:

1. No two thread may be simultaneously inside their critical regions
2. No assumption may be made about speed or number of CPUs
3. No thread running outside its critical region may block other thread
4. No thread should have to wait forever to enter its critical region

**Achieving mutual exclusion**

We shall examine the following:

- Lock variables
- Strict Alternation
- sleep and wake up
- Semaphore
- Monitors

**Lock variables**

- consider having a shared (lock) variable, \( v =0 \)
- \( v=0 \) means no thread in the critical region (CR)
- \( v=1 \) means some thread is in CR

When a thread wants to enter the CR, it checks \( v \)

- If \( v=0 \), threads sets \( v \) to 1 (obtain locks) and enters CR. Threads sets \( v \) to 0 before leaving CS (release lock)
- If \( v=1 \), the thread waits.

**Question:** would this work?

**Lock variables (ctd)**

The same problem as before: Assume a thread checks the value of lock and finds out it is 0. But before setting it to 1, another thread reads the value (which is still 0), sets the value to 1 and enters CR. Then, when the first thread starts again (it sets the value again to 1) ending up with both threads in the CR.

**Question:** how about we read the lock once and then checking it before storing? It won’t help

We need a mechanism that ensure things are atomic. How about threads taking turn?

**Strict alternation**

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
</tr>
</thead>
</table>
| While (true){
while (turn !=0) {
/* loop here to discover when turn change to 0 */
} enter CR;
turn = 1;
execute non CR } | While (true){
while (turn !=1) {
/* loop here to discover when turn change to 1 */
} enter CR;
turn = 0;
execute non CR |
Strict alternation (ctd)

- A lock that uses busy waiting (turn) is called a spin lock.
- Question: Are all four conditions for correct concurrent behaviour met?
- Answer: condition 3 can be violated. But it ensures the other three condition are met.

Work it out!

Condition 3 can be violated.

- Thread1 leaves its CR, sets turn = 1
- Thread2 enters its CR
- Thread2 finishes its CR
- Now, both threads in nonCR and turn = 0
- Thread1 execute its loop, exiting CR & setting turn= 1
- Now, both threads in nonCR and turn = 1
- Thread1 finishes its nonCR fast, goes back to busy loop
- At this point, Thread1 is not permitted to enter CR because turn=1

So thread2 which is not in its CR is blocking Thread1

Busy waiting

- All above methods require busy waiting.
- Other methods based on busy waiting
  - Dekker’s mutual exclusion algorithm (60s)
  - Peterson’s algorithm (1981)
  - Test and Set Lock (TSL)
- Negative aspects of busy waiting
  - wasting CPU resources
  - Priority Inversion Problem
  - Avoid waste of resources “Sleep and wakeup”

Priority inversion problem 4 processes/threads

- H: high priority process, i.e. scheduled to run when ready
- L: Low priority process
- Suppose L is executing its CR
- H becomes available (for example finishing I/O)
- H is now busy waiting and can not execute it CR, because L is in CR & L can not run to leave its CR because H has the priority and must run

Busy waiting (ctd)

- Avoid waste of resources Sleep and wakeup
- Sleep is a system call that allows the caller to block itself, until another thread wakes it up (wakeup call)
- We must also take care of the case that the thread is not sleep and it receives a wakeup call
- This leads to the idea of semaphores

Summary

- Example of race condition
- Four main conditions to ensure correct concurrent programs
- Using locks – requires ensuring atomic action
- strict alternation (not all four conditions are satisfied)
- Busy waiting and its problems