COP 3330, Spring 2013

Threads - 2

Instructor : Arup Ghosh

04-17-13

School of Electrical Engineering and Computer Science
University of Central Florida

Pausing a Thread

- Call Thread.sleep() to temporarily suspend the current thread
- Pass it the number of milliseconds that you'd like the thread to stop

Creating threads and starting threads are not the same

- A thread doesn't actually begin to execute until another thread calls the start() method on the Thread object for the new thread.
- The Thread object exists before its thread actually starts, and it continues to exist after its thread exits. This allows you to control or obtain information about a thread you've created, even if the thread hasn't started yet or has already completed.

Ending threads

- A thread will end in one of three ways:
  - The thread comes to the end of its run() method.
  - The thread throws an Exception or Error that is not caught.
  - Another thread calls one of the deprecated stop() methods. Deprecated means they still exist, but you shouldn't use them in new code and should strive to eliminate them in existing code.

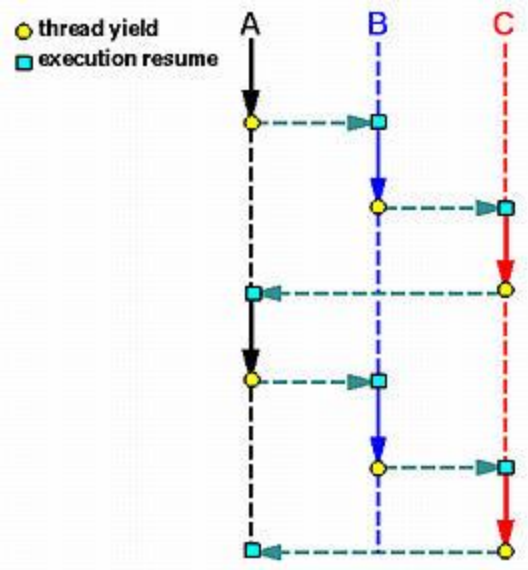
- When all the threads within a Java program complete, the program exits.

Joining with Threads

- The Thread API contains a method for waiting for another thread to complete: the join() method. When you call Thread.join(), the calling thread will block until the target thread completes.
- Thread.join() is generally used by programs that use threads to partition large problems into smaller ones, giving each thread a piece of the problem. The example at the end of this section creates ten threads, starts them, then uses Thread.join() to wait for them all to complete.

Yielding

- `Thread.yield()` lets other threads have their turn, without actually entering a sleep state
- This is basically equivalent to `Thread.sleep(0)`, except that you don't need to deal with any `InterruptedException`s that might be thrown



Thread Scheduling

- Threads can't all be running at exactly the same time, so they need to share
- Java uses round-robin scheduling
  - The scheduler lets one thread use the processor for a little while
  - Then the scheduler takes the processor back from the thread and lets a different thread use it
  - The scheduler goes through the threads circularly, so it's fair to everyone

Every Java program uses threads

- Every Java program has at least one thread -- the main thread. When a Java program starts, the JVM creates the main thread and calls the program's main() method within that thread.
- The JVM also creates other threads that are mostly invisible to you -- for example, threads associated with garbage collection, object finalization, and other JVM housekeeping tasks.
- Other facilities create threads too, such as the AWT (Abstract Windowing Toolkit) or Swing UI toolkits, etc.

Thread States

- Threads can be in different states depending on the circumstances:
- Running
  - The thread has control of the CPU and is doing work
  - When its time expires, it goes to a ready state

Thread States

- Ready
  - The thread is waiting for its turn to run
  - Eventually the scheduler will dispatch the thread and it enters a running state
- Sleeping
  - The thread is done working for now and has taken itself out of the rotation
  - Eventually, the thread will wake up and enter a ready state

Thread States

- Blocking
  - The thread needs to do I/O operations
  - I/O is slow so the thread is essentially asleep until the I/O operation completes
  - When the I/O operation completes, the thread enters a ready state

Thread States

- Waiting
  - The thread is waiting until an event occurs
  - If the thread is notified (using notify() or notifyAll()) then it becomes ready
  - If the thread is interrupted, it enters a ready state and receives an exception

All threads live in the same memory space

- As we discussed earlier, threads have a lot in common with processes, except that they share the same process context, including memory, with other threads in the same process.
- This is a tremendous convenience, but also a significant responsibility. Threads can easily exchange data among themselves simply by accessing shared variables (static or instance fields), but threads must also ensure that they access shared variables in a controlled manner, lest they step on each other's changes.

Deadlock

- Deadlock occurs when two (or more) threads are waiting on each other to finish what they're doing
- Example
  - Thread A controls the printer, but is waiting for thread B to give up control of a file it wants to print
  - Thread B controls a file, but is waiting for thread A to give up control of the printer

Deadlock

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

-- Kansas Legislature

Race Conditions

- A race condition occurs when 2 or more threads are able to access shared data and they try to change it at the same time.
- In a Race Condition, the result of the program depends on the scheduling of events
- This is usually undesirable
- This is frequently the result of two different threads trying to work with the same variable

Race Conditions

- Race conditions sometimes occur because a thread may be interrupted at any time, even in the middle of evaluating an expression

Synchronization for controlled access

- The Java language provides two keywords for ensuring that data can be shared between threads in a controlled manner: synchronized and volatile.

- Synchronized has two important meanings: it ensures that only one thread executes a protected section of code at one time (mutual exclusion or mutex), and it ensures that data changed by one thread is visible to other threads (visibility of changes).

Ensuring visibility of changes to shared data

- Processors can use caches to speed up access to memory (or compilers may store values in registers for faster access). On some multiprocessor architectures, if a memory location is modified in the cache on one processor, it is not necessarily visible to other processors until the writer's cache is flushed *and* the reader's cache is invalidated.

- This means that on such systems, it is possible for two threads executing on two different processors to see two different values for the same variable! This sounds scary, but it *is* normal. It just means that you have to follow some rules when accessing data used or modified by other threads.

Ensuring visibility of changes to shared data

- Volatile is simpler than synchronization and is suitable only for controlling access to single instances of primitive variables -- integers, booleans, and so on. When a variable is declared volatile, any write to that variable will go directly to main memory, bypassing the cache, while any read of that variable will come directly from main memory, bypassing the cache.

- This means that all threads see the same value for a volatile variable at all times. Without proper synchronization, it is possible for threads to see stale values of variables or experience other forms of data corruption.

Example

- Calculating Primes
- This example uses a background thread to calculate primes, then sleeps for ten seconds. When the timer expires, it sets a flag.