Multithreaded Programming
Part 3

Prof. Siddharth Shah
Department of Computer Engineering
Dharmsinh Desai University

Outline

- Interthread Communication
- Deadlock
- Suspending, Resuming and Stopping Threads
- Using Multithreading

Interthread Communication - 1

- As discussed earlier, multithreading divides your tasks into discrete, logical units.
- Threads also provide a secondary benefit: they do away with polling.
- Polling is usually implemented by a loop that is used to check some condition repeatedly.
- Once the condition is true, appropriate action is taken.
- This wastes CPU time.

Interthread Communication - 2

- For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

Interthread Communication - 3

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.
- These methods are implemented as final methods in Object, so all classes have them.
- All three methods can be called only from within a synchronized context.

Interthread Communication - 4

- Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:
- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).
- Additional forms of wait( ) exist that allow you to specify a period of time to wait.
- **notify( )** wakes up a thread that called wait( ) on the same object.
- **notifyAll( )** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

Deadlock - 1

- A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected.
- However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock - 2

- Deadlock is a difficult error to debug for two reasons:
- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

Suspending, Resuming and Stopping Threads - 1

- Prior to Java 2, a program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by Thread, to pause, restart, and stop the execution of a thread.
- Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they are deprecated and should not be used.
- **suspend()** can sometimes cause serious system failures like deadlock.
- **resume(),** no resume without suspend()
- **stop()** can sometimes cause serious system failures like leave a data in corrupted state.

Suspending, Resuming and Stopping Threads - 2

- A thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.
- Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread.
- As long as this flag is set to “running,” the run( ) method must continue to let the thread execute.
- If this variable is set to “suspend,” the thread must pause.
- If it is set to “stop,” the thread must terminate.
- Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

Using Multithreading

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. With the careful use of multithreading, you can create very efficient programs.
- However if you create too many threads, you can actually degrade the performance of your program rather than enhance it. As some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program.
- To create compute-intensive applications that can automatically scale to make use of the available processors in a multi-core system, consider using the new Fork/Join Framework.