Managing the Java Thread Lifecycle: Java Thread Interrupts vs. Hardware/OS Interrupts

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Know various ways to stop Java threads
  • Stopping a thread with a volatile flag
  • Stopping a thread with an interrupt request
  • Learn the patterns of interrupting Java threads
• Understand differences between a Java thread interrupt & a hardware/OS interrupt
Java Thread Interrupts vs. Hardware/OS Interrupts
Java Thread Interrupts vs Hardware/OS Interrupts

- Interrupts at the hardware or OS layers have several properties

Interrupts at the hardware or OS layers have several properties

- **Asynchronous**
  - Can occur essentially anytime & are independent of the instruction currently running

Interrupts at the hardware or OS layers have several properties

- **Asynchronous**
  - Can occur essentially anytime & are independent of the instruction currently running
  - A program needn’t test for them explicitly since they occur “out-of-band”
Interrupts at the hardware or OS layers have several properties:

- Asynchronous
- Preemptive
  - Pause (& then later resume) the execution of currently running code without its cooperation

See en.wikipedia.org/wiki/Preemption_(computing)
This example shows how to catch the UNIX SIGINT signal

```c
void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("can't catch SIGINT\n");

    for (; ;)
        sleep(10);

    return 0;
}
```

See [www.thegeekstuff.com/2012/03/catch-signals-sample-c-code](http://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code)
This example shows how to catch the UNIX SIGINT signal. It occurs asynchronously.

```c
void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    if (signal(SIGINT, sig_handler)
        == SIG_ERR)
        printf("can't catch SIGINT\n");

    for (; ;)
        sleep(10);

    return 0;
}
```

The SIGINT interrupt is typically generated by typing ^C in a UNIX shell.
Java Thread Interrupts vs Hardware/OS Interrupts

- This example shows how to catch the UNIX SIGINT signal
  - It occurs asynchronously
  - It preempts the current instruction

```c
void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    if (signal(SIGINT, sig_handler)
        == SIG_ERR)
        printf("can't catch SIGINT\n");

    for (;;)
        sleep(10);

    return 0;
}
```
Java Thread Interrupts vs Hardware/OS Interrupts

- This example shows how to catch the UNIX SIGINT signal
  - It occurs asynchronously
  - It preempts the current instruction
  - It needn’t be tested for explicitly

```java
void sig_handler(int signo) {
    if (signo == SIGINT)
        printf("received SIGINT\n");
}

int main(void) {
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("can't catch SIGINT\n");

    for (; ;)
        sleep(10);

    return 0;
}
```
Java Thread Interrupts vs Hardware/OS Interrupts

- Asynchronous & preemptive interrupt handling make it hard to reason about programs

See [en.wikipedia.org/wiki/Unix_signal#Risks](en.wikipedia.org/wiki/Unix_signal#Risks)
Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.

- Race conditions

**Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly**

See [en.wikipedia.org/wiki/Race_condition#Software](en.wikipedia.org/wiki/Race_condition#Software)
Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.

- Race conditions
- Re-entrancy problems

A non-entrant function cannot be interrupted in the middle of its execution & then safely called again before its previous invocations complete execution

See [en.wikipedia.org/wiki/Reentrancy_(computing)](en.wikipedia.org/wiki/Reentrancy_(computing))
Java Thread Interrupts vs Hardware/OS Interrupts

- Asynchronous & preemptive interrupt handling make it hard to reason about programs, e.g.
  - Race conditions
  - Re-entrancy problems
  - Non-transparent restarts

See en.wikipedia.org/wiki/PCLSRing#Unix-solution:_restart_on_request

* e.g., an I/O operation returns the # of bytes transferred & it is up to the application to check this & manage its own resumption of the operation until all the bytes have been transferred
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts

See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
  - i.e., they don’t occur at an arbitrary point & don’t pause (& later resume) running code
Java Thread Interrupts vs Hardware/OS Interrupts

• Java thread interrupts differ from hardware or operating system interrupts, e.g.
  • Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
  • A program must test for them explicitly

```java
void processNonBlocking()
{
    ...
    while (true) {
        ...
            // Do some long-running computation
        if (Thread.interrupted())
            throw new InterruptedException();
    ...
```
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than asynchronous & preemptive
- A program must test for them explicitly
  - i.e., `InterruptedException` is (usually) thrown synchronously & is handled synchronously

```java
void processNonBlocking()
{
    ...
    while (true) {
        ... // Do some long-running computation
        if (Thread.interrupted())
            throw new InterruptedException();
    ...
}
```
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
- A program must test for them explicitly
- Certain operations cannot be interrupted
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & *non-preemptive* rather than *asynchronous* & *preemptive*
- A program must test for them explicitly
- Certain operations cannot be interrupted, e.g.
  - Blocking I/O calls that aren’t invoked on “interruptable channels”

```java
static class SleeperThread extends Thread {
    public void run() {
        int c;
        try {
            c = System.in.read();
        }
        ...
    }
}
```

See bugs.java.com/bugdatabase/view_bug.do?bug_id=4514257
Java Thread Interrupts vs Hardware/OS Interrupts

- Java thread interrupts differ from hardware or operating system interrupts, e.g.
  - Delivery is *synchronous* & non-preemptive rather than asynchronous & preemptive
- A program must test for them explicitly
- Certain operations cannot be interrupted, e.g.
  - Blocking I/O calls that aren’t invoked on “interruptable channels”
  - Waiting to acquire an “intrinsic lock”

```java
void someMethod() {
    synchronized (this) {
        ...
    }
}

synchronized void anotherMethod() {
    ...
}
```

End of Managing the Java Thread Lifecycle: Java Thread Interrupts vs. Hardware/OS Interrupts