

CSCI 340
Lecture # 11
Lecturer: Dr. Simina Fluture

Topics: **Review of Threads, Java Threads**
 Thread Concept
 Kernel Thread /User Thread
 Thread state diagram
 Operations on threads

Readings: **Web and Class notes**
 Textbook: related topics

Threads

A traditional process (also named a heavyweight process) is a single task with one single thread.

A thread is also called a lightweight process (LWP), and may consist of a program counter, register set and a stack space. All threads in a process share the same address space.

A task consists of a collection of resources like: **main memory (code section, data section), I/O devices, files.**

Multithreading refers to the ability of an operating system to support multiple threads within a single process.

Some operating systems support threads internally (kernel level threads, or in Unix terminology *Bound threads*) through system calls, while others (user level threads, or in Unix terminology *Unbound threads*) support them above the kernel, using library calls.

Java Threads

Java provides support for threads at the language level. Java provides a set of APIs to manage threads.

State Diagram, Operations on threads

Creation of a thread: brings the thread into the new state.

New: an object for the thread is created.
 no system resources have been allocated yet.

Start a thread. Make a thread runnable.
Resources are allocated to the thread; the thread goes into the **Runnable** state.

Two ways of providing the run() method for a thread:
Subclassing the thread class and overriding the run() method.

Class A extends Thread {

```
    Public void run( ) {  
  
        //code  
  
    }  
}
```

CSCI 340
Lecture # 11
Lecturer: Dr. Simina Fluture

Implementing the Runnable interface.

Class B implements Runnable {

```
    Public void run() {  
        //code  
    }  
}
```

Blocked state: (not runnable)

Reasons

waits for an event (for a specific condition to be True). For example calls a **join** method on another thread object whose thread has not yet terminated.
waits for the completion of an I/O.
waits for the lock on a synchronized method.
waits for a fixed amount of time to elapse.

Methods

suspend() suspends execution of the currently running thread. (the method is **deprecated**, deadlock for monitors)

join() waits for this thread to die.

System call for an I/O

wait() on an object.

sleep() puts the currently running thread to sleep for a specified amount of time (milliseconds)

For the *wait()* and *sleep()* methods, if the thread that is interrupted is blocked, the method that blocked the thread throws an InterruptedException object.

Dead state: the thread exits (terminates).

If the thread terminates normally – the run method terminates.

If the thread terminates abnormally – *stop()* (**deprecated**)

system.exit(0)	indicates that system terminated normally
system.exit(1)	there is an error

isAlive() returns a Boolean value that determines if a Thread is in the Dead state or not.

Scheduling in Java

Java uses a preemptive priority CPU scheduling algorithm.

MIN_PRIORITY(1)

MAX_PRIORITY(10)

DEFAULT_PRIORITY(5)

t.setPriority()

t.getPriority()

Windows95/NT (of JDK 1.1) implements time slicing, while Solaris 2.x (of JDK 1.1) does not.