CSCI 340
Lecturer: Dr. Simina Fluture

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

Readings: **Java API – Thread Class**
Textbook: related topics

Processes vs Threads

Regular processes also called **heavyweight processes**: one thread and one task.

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

Threads

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.

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

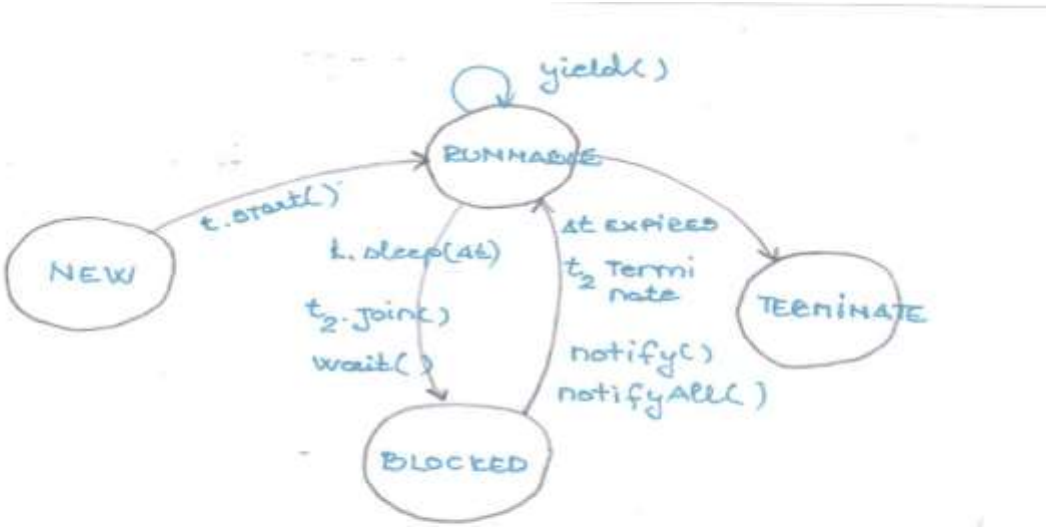
Some operating systems support threads internally (kernel level threads, or in Unix terminology *bound threads*) through system calls, while others (user level threads, or in Unix terminology *unbound threads*) support them above the kernel, using library calls.

Advantages:
Disadvantages:

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: bringing the thread into the new state.

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

Starting a thread: making a thread runnable.
Resources are allocated to the thread.
the thread goes into the **Runnable state**.

They are 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
    }
}
```

Never use t.run(); the start method will call the run method for us.

Implementing the Runnable interface

```
Class B implements Runnable {

        Public void run( ) {

                //code
        }
}
```

Blocked state: (not runnable).

Reasons for a thread to move into the blocked state

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 (that will block a thread)

~~suspend( )~~ suspends execution of the currently running thread. (the method is d**eprecated**, it creates deadlock for monitors)
join( ) waits for this thread to die. Can be used to enforce a sequential order between concurrent threads.
wait( ) on an class object or a notification object.
sleep(time) puts the currently running thread to sleep for a specified amount of time (milliseconds)

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

Dead/terminate state: the thread exits (terminates).
The thread terminates normally when it terminates the run method.
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.
In project 1, *isAlive()* should be used together with *join().*

Scheduling in Java

Java uses a preemptive priority CPU scheduling algorithm.

MIN_PRIORITY(1)
MAX_PRIORITY(10)
NORM_PRIORITY(5)

t.setPriority( )
t.getPriority( )

Some operating systems use time slicing for threads with same priority. Windows implements time slicing, while Solaris 2.x (of JDK 1.1) does not.