Advanced Programming

Week 3

Multithreading

- Introduction
- Multi-threading
- Life cycle of a Thread
- Creating and Executing Threads
- Thread Scheduler

Tilahun Melak(PhD)

September, 2025

Objectives

At the end of this lecture students will be able to :

- Explain multi-threaded programing
- Explain the Life Cycle of a Thread
- Create threads
- Discuss Thread Scheduling

Introduction

- We can walk, talk, breathe, see, hear, smell... all at the same time
- Computers can do this as well - download a file, print a file, receive email, run the clock
- Only computers that have multiple processors can truly execute multiple instructions concurrently
- Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once.
- Java makes concurrency available to you through the language and APIs.

What is a Thread?

- A thread is a control/flow/path of execution that exists within a process.
- Thread is portion of a program that can execute concurrently with other threads.
- Each thread is a statically ordered sequence of instructions.
  - Sequential flow of control within a program
- Threads are being extensively used to express concurrency on both single and multiprocessors machines.
  - Sharing a single CPU between multiple tasks (or "threads") in a way designed to minimize the time required to switch tasks.
    - accomplished by sharing as much as possible of the program execution environment between the different tasks.

What is a Thread?

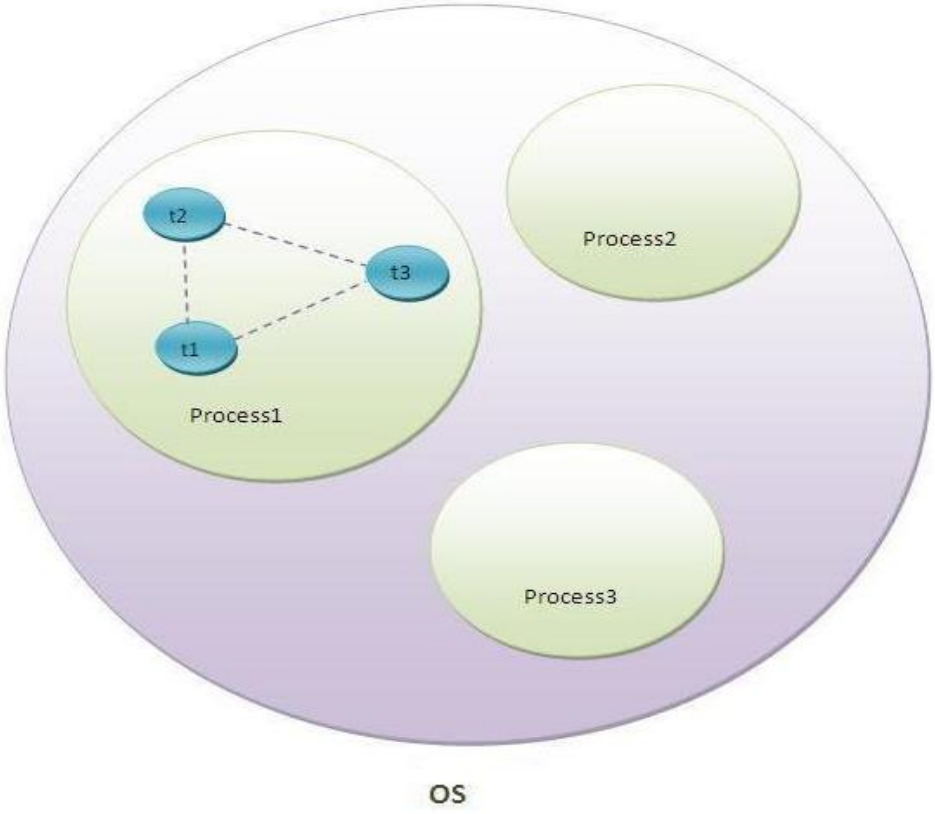
- Programming a task having multiple threads of control is called **Multithreading** or **Multithreaded Programming**.
- A multithreaded application contains separate threads of execution, where each thread
  - Has its own method-call stack and program counter
  - execute concurrently with other threads.
  - shares application-wide resources such as memory with other threads.
- In single-threaded applications lengthy activities must complete before others can begin which leads to poor responsiveness.

What is a Thread?

- In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute concurrently and the application can operate more efficiently.
- Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed (because, for example, it is waiting for the result of an I/O operation), another can use the processor.
- The Java Virtual Machine (JVM) creates threads to run a program, the JVM also may create threads for performing housekeeping tasks such as garbage collection

What is a Thread?

- As shown in the figure, thread is executed inside the process.
- There is context-switching between the threads.
- There can be multiple processes inside the OS and one process can have multiple threads.



Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts (10th ed.). Wiley.

Multi-threading

- **Multithreading**
  - executing multiple threads simultaneously.
- A thread is the smallest unit of execution (lightweight sub-process).
- Both multiprocessing and multithreading enable multitasking.
- **Why multithreading?**
  - Threads share memory (no separate allocation).
  - Saves memory and reduces context-switching time vs. processes.
- **Applications:**
  - game development, animations, real-time systems.

Schildt, H. (2018). Java: The complete reference (11th ed.). McGraw-Hill Education.

Why Multithreading?

- Keeps applications responsive during long-running tasks.
- Allows cancellation of independent tasks.
- Handles problems that are naturally parallel.
- Enables monitoring of resources (e.g., databases).
- Utilizes multiple processors efficiently.
- Supports simultaneous execution of multiple operations.

Schildt, H. (2018). Java: The complete reference (11th ed.). McGraw-Hill Education.

Life cycle of a Thread (Thread States)

- A thread occupies one of several thread states.
  - Born (New) state
    - Thread just created
    - A thread begins its life cycle in the new state.
    - It remains in this state until the start() method is called on it.
  - Runnable(Ready state)
    - When the program starts the call thread (start method) it enters the *ready* state.
    - After invocation of start() method on new thread, the thread becomes runnable.
    - Highest-priority ready thread enters running state

Life cycle of a Thread (Thread States) Cntd..

- Running state
  - A thread is in running state if the thread scheduler has selected it.
  - System assigns processor to thread (thread begins executing)
  - When the thread's quantum expires, the thread returns to the *Ready* state and the operating system dispatches another thread
  - Transitions between the ready and running states are handled solely by the operating system
  - When run method completes or terminates, enters dead state
- Blocked state
  - Entered from running state when
  - waiting on I/O request
  - Wants to enter a synchronized statement
  - cannot use processor, even if available

Life cycle of a Thread (Thread States) Cntd..

- Running state
  - A thread is in running state if the thread scheduler has selected it.
  - System assigns processor to thread (thread begins executing)
  - When the thread's quantum expires, the thread returns to the *Ready* state and the operating system dispatches another thread
  - Transitions between the ready and running states are handled solely by the operating system
  - When run method completes or terminates, enters dead state
- Blocked state
  - Entered from running state when
  - waiting on I/O request
  - Wants to enter a synchronized statement
  - cannot use processor, even if available

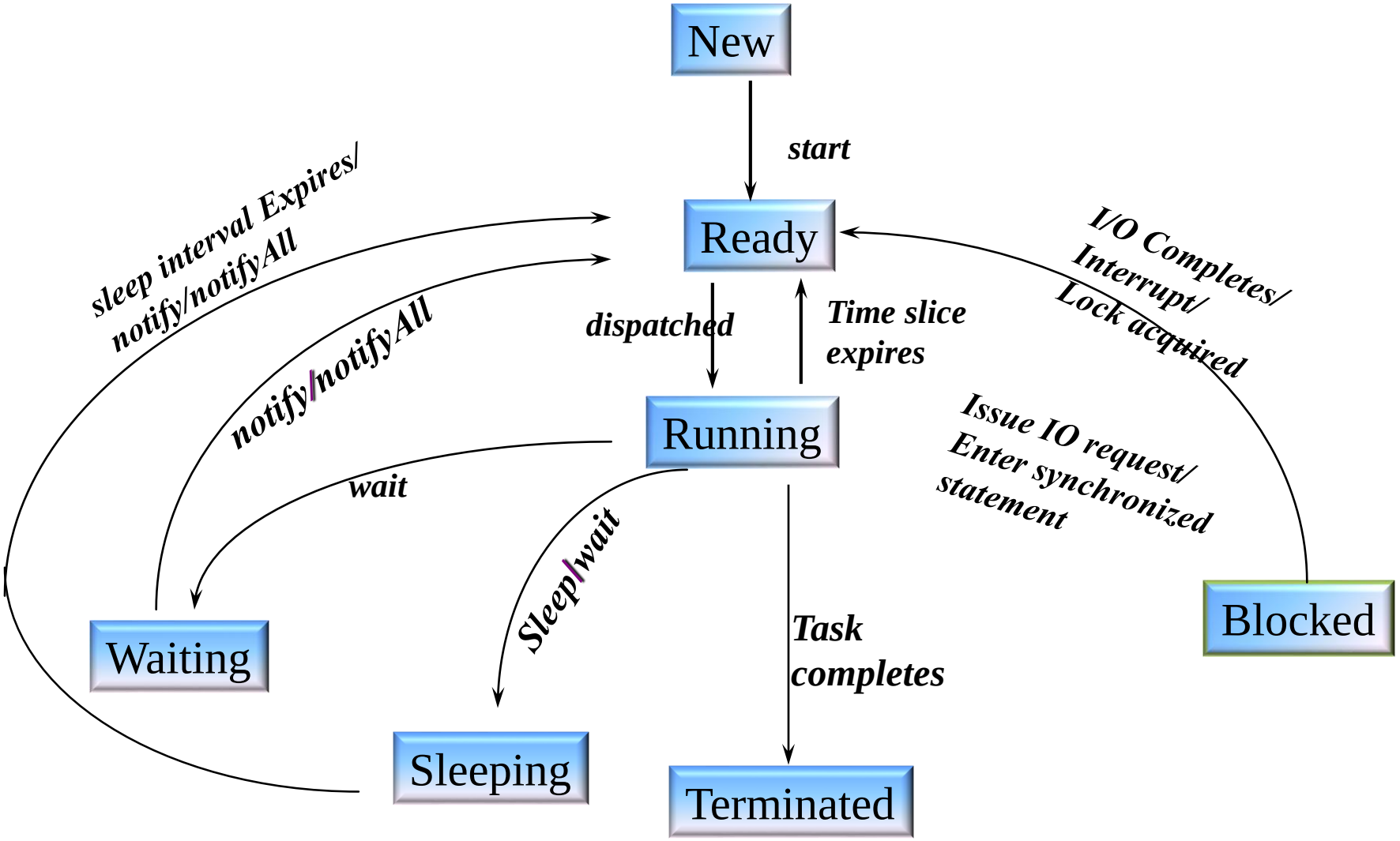
Life cycle of a Thread (Thread States) Cntd..

- Sleeping (Timed waiting) state
  - Entered from running state when
    - sleep method called or
    - wait is called with a time interval.
  - Cannot use processor , even if available
  - transitions back to the *Ready* state when that time interval expires or when it is notified by another thread
- Waiting state
  - Entered from running state when calls wait, the thread enters a waiting state for the particular object on which wait was called.
  - One waiting thread becomes ready when object calls notify
  - notifyAll - all waiting threads become ready
  - cannot use a processor, even if one is available.
  - In this stage the thread is still alive

Life cycle of a Thread (Thread States) Cntd..

- Terminated(Dead) state
  - Thread marked to be removed by system
  - A thread enter the terminated when it complete its task.

Life cycle of a Thread (Thread States) Cntd..



Oracle. (n.d.). Thread states. In Java Platform, Standard Edition Documentation.

Creating and Executing Threads

- To implement multithreading, Java defines two ways by which a thread can be created.
- **Implementing the Runnable interface**
  - Runnable
    - represents a "task" that can be executed concurrently with other tasks
    - declares method run in which you place the code that defines the task to perform.
  - Define a class that implements Runnable

```
class Task implements Runnable{
    public void run(){//define the task here}
}
```

  - Instantiate the Thread class
    - invoke Thread constructor with an instance of this Runnable class
  - Then, call *start* method of the Thread instance

```
public static void main(String [] args){
    Thread t = new Thread(new Task());
    t.start(); }
```

Example: Thread with Runnable Interface

```
class Task1 implements Runnable
{
public void run()
{
System.out.println("thread is running…");
}
public static void main(String args[])
{
Thread t1 =new Thread(new Task1());
t1.start();
 }
}
```

Creating and Executing Threads Cntd..

- **By Extending Thread class**
  - Define a subclass of java.lang.Thread
    - Define a *run* method

```
class Task extends Thread{
    public void run(){define the task here}
}
```

  - In another thread (e.g., the main), create an instance of the Thread subclass
  - Then, call *start* method of that instance

```
public static void main(String [] args){
    Task t = new Task();
    t.start(); }
```

Example: Thread by Extending Thread

```
class Task2 extends Thread
{
public void run()
{
System.out.println("thread is running…");
}
public static void main(String args[])
{
Task2 t1 =new Task2();
t1.start();
 }
}
```

Thread Class

- Constructors
  - ***Thread(threadName)***
  - ***Thread()***
  - Creates an auto numbered Thread of format Thread-1, Thread-2...
- Methods
  - ***void run()***
    - "Does work" of thread
    - Can be overridden in a subclass of Thread
  - ***Start()***
    - Launches thread, then returns to caller
    - Calls run
    - Error to call start twice for same thread(IllegalThreadStateException)

Thread Class Cntd…

- Methods
  - ***static sleep****(milliseconds)*
    - Thread sleeps for number of milliseconds
    - Can give lower priority threads a chance to run
  - ***interrupt)()***
    - Interrupts a thread
  - ***static interrupted****()*
    - Returns true if current thread interrupted
  - ***isInterrupted()***
    - Determines if a thread is interrupted
  - ***isAlive()***
    - Returns true if start has been called and not dead (run function has not completed)

Thread Class Cntd…

- Methods
  - ***yield ()***
    - Cause the currently running thread to temporarily pause and allows other threads to execute.
  - **setName(threadName)**
    - Set name of the thread
  - **getName()**
    - Returns the name of the thread.

Thread Class Cntd…

- Methods
  - **toString()**
    - Returns thread name, priority, and ThreadGroup
  - **Join()**
    - Wait for a thread to end
    - No argument or 0 millisecond argument means thread will wait indefinitely

      -Can lead to deadlock

Thread Scheduler

- The thread scheduler is a part of the JVM responsible for determining which thread will execute.
- It does not guarantee which runnable thread will be selected at any given time.
- In a single process, only one thread executes at a time.
- Scheduling is typically managed using preemptive scheduling or time slicing.

Oracle. (n.d.). Class Thread (Java Platform, SE 8). Oracle Docs

Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

- The Thread class provides two overloaded methods for sleeping a thread:
  - public static void sleep(long miliseconds)throws InterruptedException
  - public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example: Sleep Method

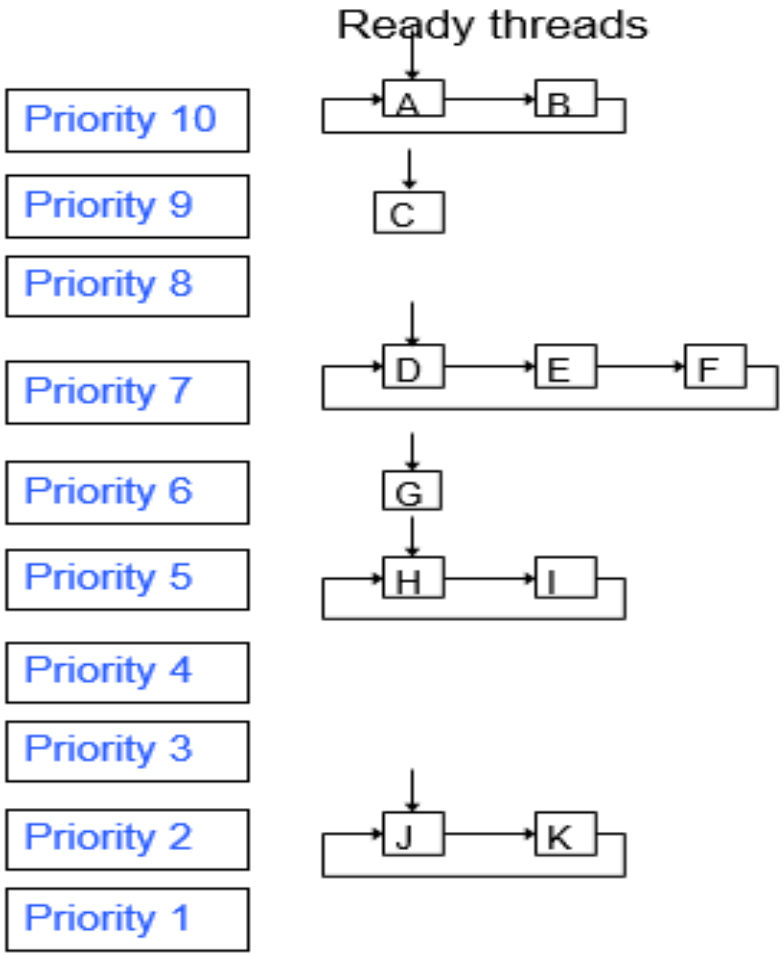
```
public class TestSleepMethod1 extends Thread {
   public void run(){
    for (int i=1; i<5; i++)    {
            try    {
                    Thread.sleep(500);    }
            catch(InterruptedException e)    {
                    System.out.println(e);    }
            System.out.println(i);   }    }
     public static void main(String args[])    {
    TestSleepMethod1 t1=new TestSleepMethod1();
     TestSleepMethod1 t2=new TestSleepMethod1();
    t1.start();
    t2.start();
            }}
```

Thread Priorities and Thread Scheduling

- Each Java thread has a priority that helps the operating system decide the scheduling order.
- Priorities range from MIN_PRIORITY = 1 to MAX_PRIORITY = 10.
- By default, threads are assigned NORM_PRIORITY = 5.
- Generally, higher-priority threads are scheduled before lower-priority ones (preemptive scheduling).
- However, this behavior is not guaranteed—it depends on the JVM implementation and the underlying OS.

Oracle. (2024). Class Thread. In Java Platform, Standard Edition 21 API Specification.

Thread Priorities and Thread Scheduling Cntd…



Oracle. (2024). Class Thread. In Java Platform, Standard Edition 21 API Specification.

Example: Thread Priority

```
public class TestThreadPriority extends Thread {
    public void run(){
    System.out.println("running thread name is:"+Thread.currentThread().getName());
    System.out.println("running thread priority:"+Thread.currentThread().getPriority());
        }
    public static void main(String args[]){
    TestThreadPriority m1=new TestThreadPriority();
    TestThreadPriority m2=new TestThreadPriority();
    m1.setPriority(Thread.MIN_PRIORITY);
    m2.setPriority(Thread.MAX_PRIORITY);
    m1.start();
    m2.start();
    }
}
```

Notes

Thread start() and run() in Java

- The start() method does two things:
  1. Instructs the JVM to create a new thread.
  2. Calls the thread's run() method in that new thread.
- The run() method is similar to the main() method:
  - main() is called by the JVM at program startup.
  - run() defines the starting point of execution for a thread.

Oracle. (n.d.). Concurrency: Creating threads with the Thread class. In Java documentation.

Notes Cntd…

- When the run() method finishes, the thread ends.
- A thread stops only after its run() method completes and returns.
- The code in main() executes within the main thread, created automatically by the JVM.
- A Java program continues running until all threads (including the main thread) finish execution.

Oracle. (n.d.). *Concurrency: Creating threads with the Thread class*. In *Java documentation*.

Summary

- In today's lecture we have discussed about;
  - Multi-threading
  - Life cycle of a Thread
  - Thread Scheduler
  - Examples

References

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts (10th ed.). Wiley.
- Schildt, H. (2018). *Java: The complete reference* (11th ed.). McGraw-Hill Education.
- Oracle. (n.d.). *Thread states*. In *Java Platform, Standard Edition Documentation*.
- Oracle. (n.d.). *Class Thread (Java Platform, SE 8)*. Oracle Docs.
- Oracle. (2024). *Class Thread*. In *Java Platform, Standard Edition 21 API Specification*.
- Oracle. (n.d.). Concurrency: Creating threads with the Thread class. In Java documentation.