Lection 8

Antsypov Alexander

Multitasking and Multithreading

Multitasking refers to a computer's ability to perform multiple jobs concurrently

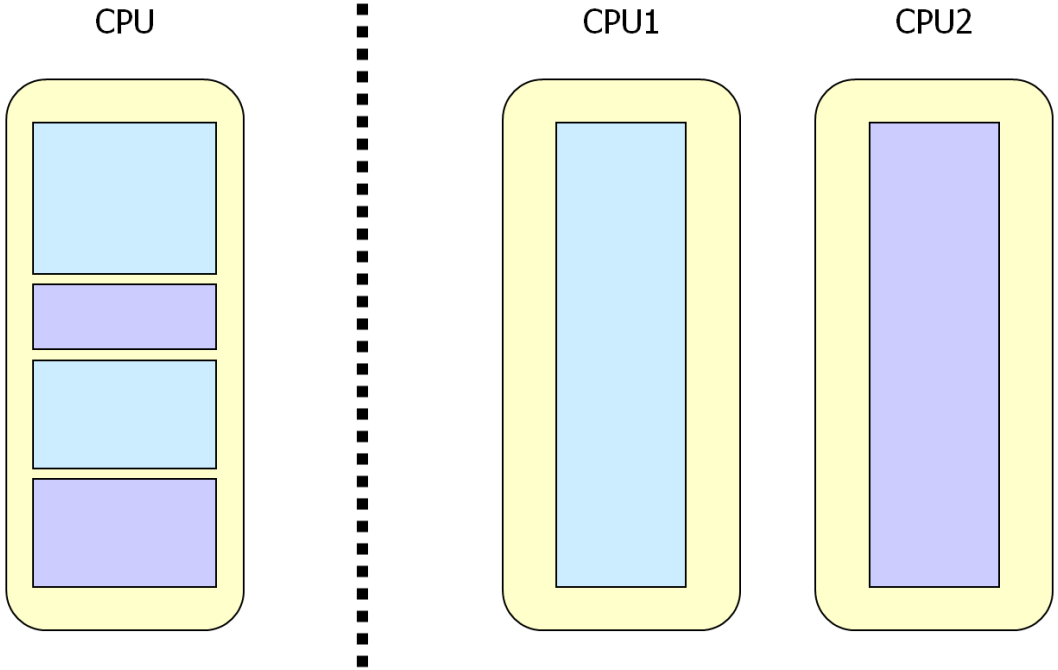
more than one program are running concurrently, e.g., UNIX

A **thread** is a single sequence of execution within a program

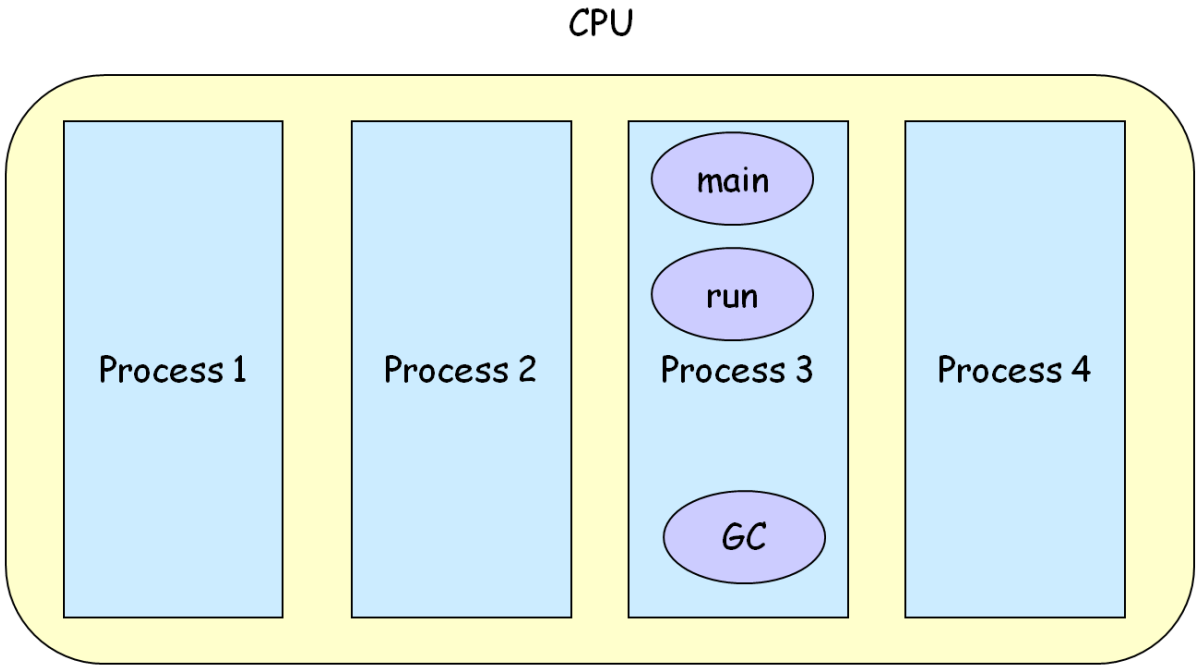
Multithreading refers to multiple threads of control within a single program

each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads and Processes



What are Threads Good For?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- Some APIs and systems demand it: Swing.

Application Thread

When we execute an application:

- The JVM creates a Thread object whose task is defined by the main() method
- It starts the thread
- The thread executes the statements of the program one by one until the method returns and the thread dies

Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap)
- Two different threads can act on the same object and same static fields concurrently

Creating Threads

There are two ways to create our own Thread object

1. *Subclassing the Thread class and instantiating a new object of that class*
2. *Implementing the Runnable interface*

In both cases the run() method should be implemented

Extending Thread

```
public class ThreadExample extends Thread {
    public void run () {
        for (int i = 1; i <= 100; i++) {
            System.out.println("Thread: " + i);
        }
    }
}
```

Thread Methods

void start()

- Creates a new thread and makes it runnable
- This method can be called only once

void run()

- The new thread begins its life inside this method

void stop() (deprecated)

- The thread is being terminated

Thread Methods

yield()

- Causes the currently executing thread object to temporarily pause and allow other threads to execute
- Allow only threads of the same priority to run

sleep(int m)/sleep(int m,int n)

- The thread sleeps for m milliseconds, plus n nanoseconds

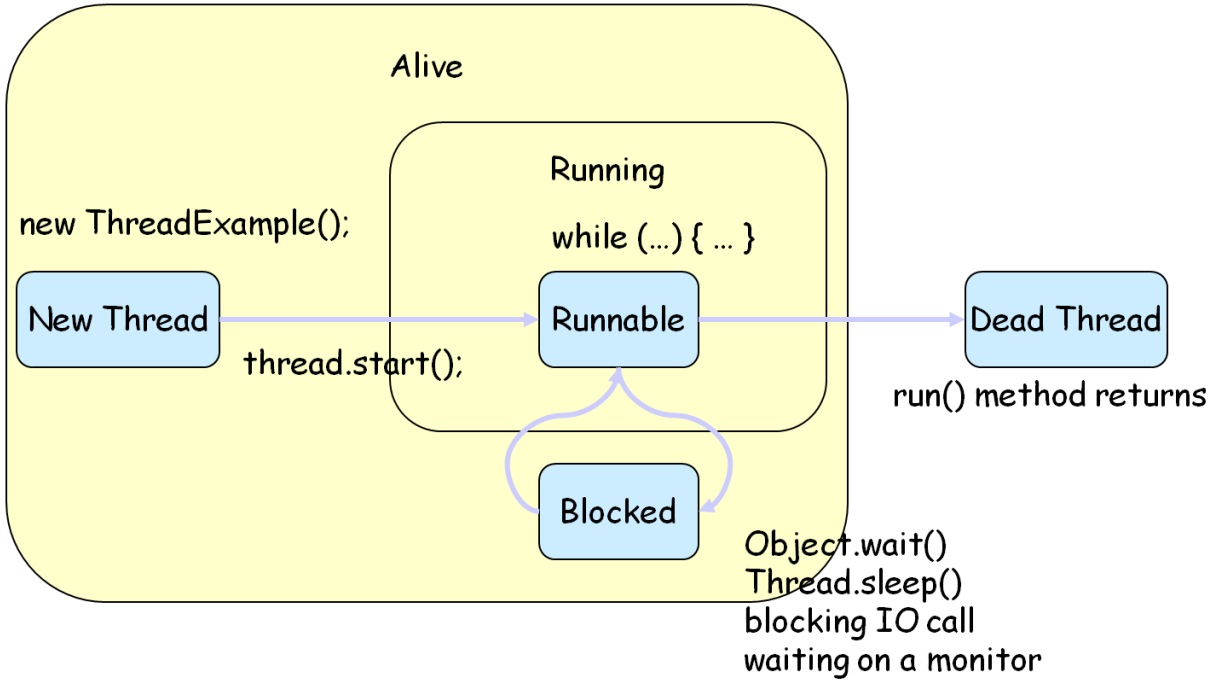
Implementing Runnable

```
public class RunnableExample implements Runnable {
    public void run () {
        for (int i = 1; i <= 100; i++) {
            System.out.println ("Runnable: " + i);
        }
    }
}
```

Starting the Threads

```
public class ThreadsStartExample {
        public static void main (String argv[]) {
                new ThreadExample ().start ();
                new Thread(new RunnableExample ()).start ();
        }
}
```

Thread State Diagram



Example

```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<500 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
        } }
```

Example

```
public static void main(String args[]) {
    PrintThread1 a = new PrintThread1("*");
    PrintThread1 b = new PrintThread1("-");
    PrintThread1 c = new PrintThread1("=");
    a.start();
    b.start();
    c.start();
}
```

Scheduling

- Thread **scheduling** is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either **preemptive** or **nonpreemptive**

Preemptive Scheduling

- **Preemptive scheduling** – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- **Nonpreemptive scheduling** – the scheduler never interrupts a running thread
- The **nonpreemptive scheduler** relies on the running thread to yield control of the CPU so that other threads may execute

Starvation

- A nonpreemptive scheduler may cause **starvation** (runnable threads, ready to be executed, wait to be executed in the CPU a lot of time, maybe even forever)
- Sometimes, starvation is also called a **livelock**

Time-Sliced Scheduling

- **Time-sliced scheduling** – the scheduler allocates a period of time that each thread can use the CPU

when that amount of time has elapsed, the scheduler preempts the thread and switches to a different thread

- **Nontime-sliced scheduler** – the scheduler does not use elapsed time to determine when to preempt a thread

it uses other criteria such as priority or I/O status

Java Scheduling

- Scheduler is preemptive and based on priority of threads
- Uses **fixed-priority scheduling**:

Threads are scheduled according to their priority w.r.t. other threads in the ready queue

Java Scheduling

- The highest priority runnable thread is always selected for execution above lower priority threads
- When multiple threads have equally high priorities, only one of those threads is guaranteed to be executing
- Java threads are guaranteed to be preemptive- but not time sliced

Thread Priority

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority

Thread Priority

- The priority can be adjusted subsequently using the setPriority() method
- The priority of a thread may be obtained using getPriority()
- Priority constants are defined:

MIN_PRIORITY=1

MAX_PRIORITY=10

NORM_PRIORITY=5

Some Notes

- Thread implementation in Java is actually based on operating system support
- Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level

Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits

ThreadGroup

The ThreadGroup class is used to create groups of similar threads. Why is this needed?

“Thread groups are best viewed as an unsuccessful experiment, and you may simply ignore their existence.”

Joshua Bloch, software architect at Sun

Server

```
import java.net.*;import java.io.*;
class HelloServer {
        public static void main(String[] args) {
                int port = Integer.parseInt(args[0]);
                try {
                        ServerSocket server =  new ServerSocket(port);
                } catch (IOException ioe) {
                        System.err.println("Couldn't run server on port " + port);
                        return;
                }
```

Server

```
  while(true) {
      try {
            Socket connection = server.accept();
            ConnectionHandler handler =
            new ConnectionHandler(connection);
            new Thread(handler).start();
  } catch (IOException ioe1) {
  }
}
```

Connection Handler

```
// Handles a connection of a client to an HelloServer.
// Talks with the client in the 'hello' protocol
class ConnectionHandler implements Runnable {
    // The connection with the client
    private Socket connection;

    public ConnectionHandler(Socket connection) {
        this.connection = connection;
    }
```

Connection Handler

```
public void run() {
        try {
                BufferedReader reader = new BufferedReader(
                new InputStreamReader(connection.getInputStream()));
                PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(connection.getOutputStream()));
        String clientName = reader.readLine();
        writer.println("Hello " + clientName);
        writer.flush();
        } catch (IOException ioe) {}
        }
}
```

Client side

```
import java.net.*; import java.io.*;
// A client of an HelloServer
class HelloClient {
        public static void main(String[] args) {
                String hostname = args[0];
                int port = Integer.parseInt(args[1]);

                Socket connection = null;
                try {
                        connection = new Socket(hostname, port);
                } catch (IOException ioe) {
                        System.err.println("Connection failed");
                        return;
                }
```

Client side

```
try {
        BufferedReader reader =
        new BufferedReader(new InputStreamReader(connection.getInputStream()));
        PrintWriter writer =
        new PrintWriter(new OutputStreamWriter(connection.getOutputStream()));

        writer.println(args[2]); // client name
        String reply = reader.readLine();
        System.out.println("Server reply: "+reply);
        writer.flush();
        } catch (IOException ioe1) {        }
}
```

Concurrency

An object in a program can be changed by more than one thread

Q: Is the order of changes that were performed on the object important?

Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object
- Both threads “race” to store their value