

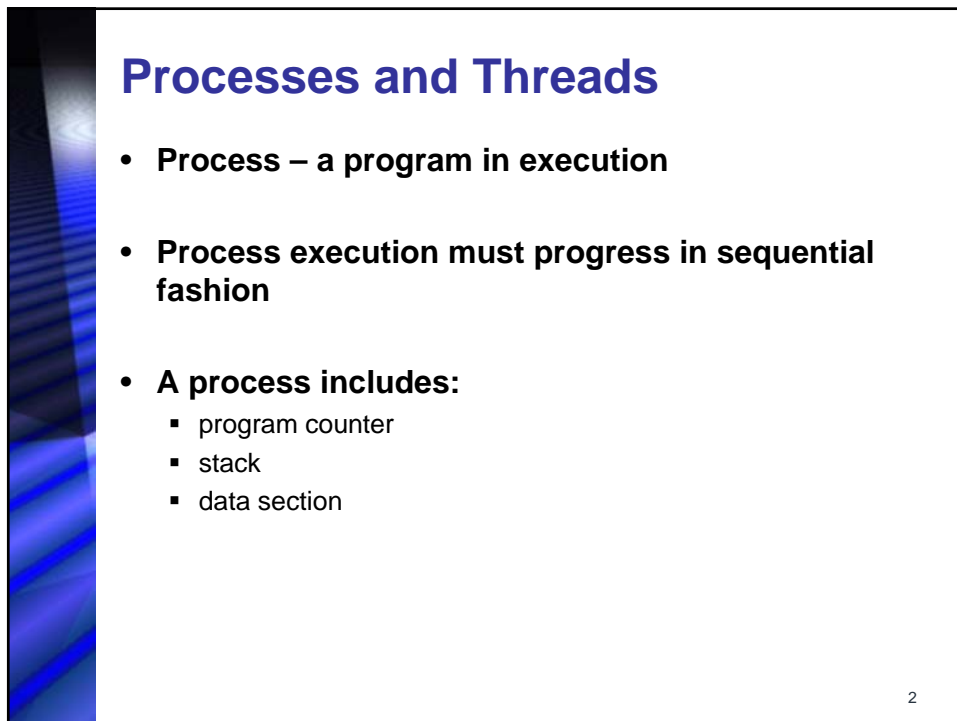
A slide with a blue and black background featuring a 3D rendering of several spheres. The text is overlaid on this background.

CSC 124
<http://faculty.cua.edu/elsharkawy/csc-124.htm>

Multi-Threaded Programming

Instructor
Sameh Elsharkawy, Ph.D.
Elsharkawy@cua.edu

 <http://faculty.cua.edu/elsharkawy/csc-124.htm> © 2007 Pearson Addison-Wesley. All rights reserved

A slide with a white background and a blue and black decorative border on the left side. The text is centered on the white background.

Processes and Threads

- **Process** – a program in execution
- **Process execution must progress in sequential fashion**
- **A process includes:**
 - program counter
 - stack
 - data section

2

Process Control Block (PCB)



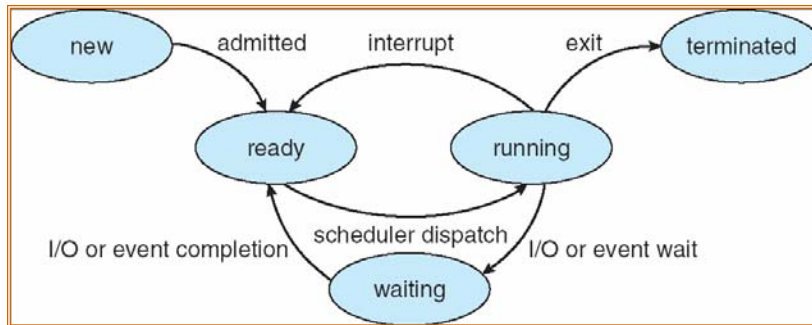
3

Processes and Threads

- **As a process executes, it changes *state***
 - **new:** The process is being created
 - **running:** Instructions are being executed
 - **waiting:** The process is waiting for some event to occur
 - **ready:** The process is waiting to be assigned to a process
 - **terminated:** The process has finished execution

4

Diagram of Process State



5

Processes and Threads

- **Processes do not directly share memory**
 - Separate virtual memory space
 - Processes can communicate by sending events and messages
 - A special area in memory can be characterized as shared memory
- **Sharing of processor (and other resources) among processes is controlled by the operating system**

6

Processes and Threads

- A process can be partitioned into multiple sub-processes (*Threads*)
- A *thread* is a program unit that is executed independently of other parts of the program
- Threads of the same process share the process's memory space and resources

7

Processes and Threads

- A program is responsible for scheduling its own threads
 - Usually uses support methods from the OS
 - In case of Java, thread scheduling is controlled by the virtual machine
 - The Java Virtual Machine executes each thread in the program for a short amount of time
 - This gives the impression of *parallel* execution

8

Steps to Running a Threads

- Implement a class that extends the **Thread** class
- Place the code for your task into the **run** method of your class
- Create an object of your subclass
- Call the **start** method of your class to start the thread
- When a **Thread** object is started, the code in its **run** method is executed in a new thread

9

GreetingThread Outline

- A program to print a time stamp and "Hello World" once a second for ten seconds

```
public class GreetingThread extends Thread
{
    public void run()
    {
        //thread action
        . . .
    }

    //variables used by the thread action
    . . .
}
```

10

Thread Action for GreetingThread

- Print a time stamp
- Print the greeting
- Wait a second

11

GreetingThread

- We can get the date and time by constructing Date object
`Date now = new Date();`
- To wait a second, use the sleep method of the Thread class
`sleep(1000);`
- A sleeping thread can generate an **InterruptedException**
 - Catch the exception
 - Terminate the thread

12

GreetingThread run Method

```
public run( )
{
    try
    {
        //thread action
    }
    catch (InterruptedException exception)
    {
        //cleanup, if necessary
    }
}
```

13

To Start the Thread

- Construct an object of your thread class
GreetingThread t = new
GreetingThread("Hello World");
- Call the start method
t.start();

14

Thread Scheduling

- The thread scheduler runs each thread for a short amount of time called a *time slice*
- Then the scheduler picks another thread from those that are *runnable*
- A thread is *runnable* if it is not asleep or blocked in some way
- There is no guarantee about the order in which threads are executed

15

Terminating Threads

- A thread terminates when its *run* method returns
- Do not terminate a thread using the deprecated *stop* method
- Instead, notify a thread that it should terminate

```
t.interrupt();
```

16

Terminating Threads

- A thread's `run` method should check occasionally whether it has been interrupted
 - Use the `isInterrupted` method
 - An interrupted thread should release resources, clean up, and exit
- The `sleep` method throws an `InterruptedException` when a sleeping thread is interrupted
 - Catch the exception
 - Terminate the thread

17

Terminating Threads

```
public void run(  
{  
    try  
    {  
        for (int i = 1; i <= REPETITIONS &&  
            !isInterrupted(); i++)  
        {  
            //do the work  
        }  
    }  
    catch (InterruptedException exception)  
    {  
        // Handle early termination case  
    }  
    //cleanup  
}
```

18

Corrupting Data with Unsynchronized Threads

- When threads share a common object, they can conflict with each other.
- In this example, a `DepositThread` and a `WithdrawThread` both manipulate a single `BankAccount`

19

Method of `DepositThread`

```
public void run()
{
    try
    {
        for (int i = 1; i <= REPETITIONS &&
            !isInterrupted(); i++)
        {
            account.deposit(amount);
            sleep(DELAY);
        }
    }
    catch (InterruptedException exception)
    {
    }
}
```

20

Example

- Create a **BankAccount** object
- Create a **DepositThread t0** to deposit \$100 into the account for 10 iterations
- Create a **WithdrawThread t1** to withdraw \$100 from the account for 10 iterations
- The result should be zero, but sometimes it is not

21

Example: Error Scenario

- The first thread **t0** executes the lines
`System.out.println("Depositing " + amount);`
`double newBalance = balance + amount;`
- **t0** reaches the end of its time slice and **t1** gains control
- **t1** calls the withdraw method which withdraws \$100 from the balance variable.
- Balance is now -100
- **t1** goes to sleep

22

Example: Error Scenario

- `t0` regains control and picks up where it left off.
- `t0` executes the lines

```
System.out.println(" new balance is " +  
newBalance);  
balance = newBalance;
```
- The balance is now 100 instead of 0 because the deposit method used the OLD balance
- This is called a **race condition**.

23

Race Conditions

- Occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled
- It is possible for a thread to reach the end of its time slice in the middle of a statement
- It may evaluate the right-hand side of an equation but not be able to store the result until its next turn
- Example:
 - BankAccountThreadTest.java*
 - DepositThread.java*
 - WithdrawThread.java*
 - BankAccount.java*

24

Solving the Race Condition Problem

- A thread must be able to **lock** an object temporarily
- When a thread has the object locked, no other thread can modify the state of the object.
- In Java, use **synchronized** methods to do this.
- Tag all methods that contain thread-sensitive code with the keyword **synchronized**

25

Synchronized Methods

```
public class BankAccount
{
    public synchronized void deposit(double amount)
    {
        . . .
    }

    public synchronized void withdraw(double amount)
    {
        . . .
    }

    . . .
}
```

26

Synchronized Methods

- **By declaring both the deposit and withdraw methods to be synchronized**
- **Our program will run correctly**
- **Only one thread at a time can execute either method on a given object**
- **When a thread starts one of the methods, it is guaranteed to execute the method to completion before another thread can execute a synchronized method on the same object.**

27

Synchronized Methods

- **By executing a synchronized method:**
 - **The thread acquires the object lock.**
 - **No other thread can acquire the lock.**
 - **No other thread can modify the state of the object until the first thread is finished**

28

Visualization of Synchronized Thread Behavior

- Imagine the object is a restroom that only one person can use at a time
- The threads are people
- If the restroom is empty, a person may enter
- If a second person finds the restroom locked, the second person must wait until it is empty
- If multiple people want to gain access to the restroom, they all wait outside
- The people may not form an orderly queue;
- A randomly chosen person may gain access when the restroom becomes available again

29

Deadlocks

- A deadlock occurs if no thread can
- proceed because each thread is waiting
- for another to do some work first
- **BankAccount example**

```
public synchronized void withdraw
    (double amount)
{
    while (balance < amount)
        //wait for balance to grow
        . . .
}
```

30

Deadlocks

- The method can lead to deadlock
- The thread sleeps to wait for balance to grow, but it still has the lock
- No other thread can execute the synchronized deposit method
- If a thread tries to call deposit, it is blocked until the withdraw method exits
- Withdraw method can't exit until it has funds available
- **DEADLOCK**

31

Avoiding Deadlocks

- The **wai t** method temporarily releases the object lock and deactivates the thread
- This gives a chance for another thread to gain access to the shared resource lock and change the condition that lead to the waiting and deadlock
- **BankAccount:**
 - Another **deposi t** method can enter and increase the account balance

32

withdraw Method to Avoid Deadlock

```
public synchronized void withdraw
(double amount)
throws InterruptedException
{
    while (balance < amount)
        wait();
}
```

33

Wait and notifyAll

- A thread that calls `wait` is in a blocked state
- It will not be activated by the thread scheduler until it is unblocked
- It is unblocked when another thread calls `notifyAll`
- When a thread calls `notifyAll`, all threads waiting on the object are unblocked
- Only the thread that has the lock can call `notifyAll`
- Example:
 - BankAccount with deadlock avoidance

34

The *Thread* Class

- **Hierarchy:**
 - [java.lang.Object](#)
 - java.lang.Thread
- **Implemented Interfaces:**
 - [Runnable](#)
- **Every thread has a priority**
 - Threads with higher priority are executed in preference to threads with lower priority.
 - Threads are created with initial priority equal to that of creating code's thread.
 - Initial priority can be changed.

35

The *Thread* Class

- **The Java Virtual Machine continues to execute threads until either of the following occurs:**
 - The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
 - All threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

36