CS331: Java Threads

http://jatinga.iitg.ac.in/~asahu/cs331/

A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline

- Thread Methodology
- Eight Rules for Designing Multithreaded Apps
- Java thread
  - Creation and lifecycle
- Examples
  - VectorSum
- Synchronized
  - Function, block, objects
- **Source Code Available@**

  http://jatinga.iitg.ac.in/~asahu/cs331/

Multicore Difficulties

- Multiprocessors are likely to be cost/power effective solutions
  - Because it share lots of resources
    - ***Personal room is costlier than dormitory***
  - Sharing resource arise many other problems
    - Critical Sections
      - Lock and Barrier Design
    - Coherence
      - Shared data at all placed should be same
    - Consistency
      - Order should be similar to serial (ROB)
    - One processor Interference others
      - Share efficiently using some policy

Threading Methodology

Threading Methodology

- **Does not recommend going straight to concurrency!**
- First produce a tested single-threaded program
  - Use reqs./ design/ implement /test/ tune/ maintenance steps
- Then to create a concurrent system from the former, do

1. Analysis: Find computations that are independent of each other
   1. AND take up a large amount of serial execution time (80/20 rule)
2. Design and Implement: straightforward Test for Correctness: Verify that concurrent code produces correct output
3. Tune for performance: once correct, find ways to speed up

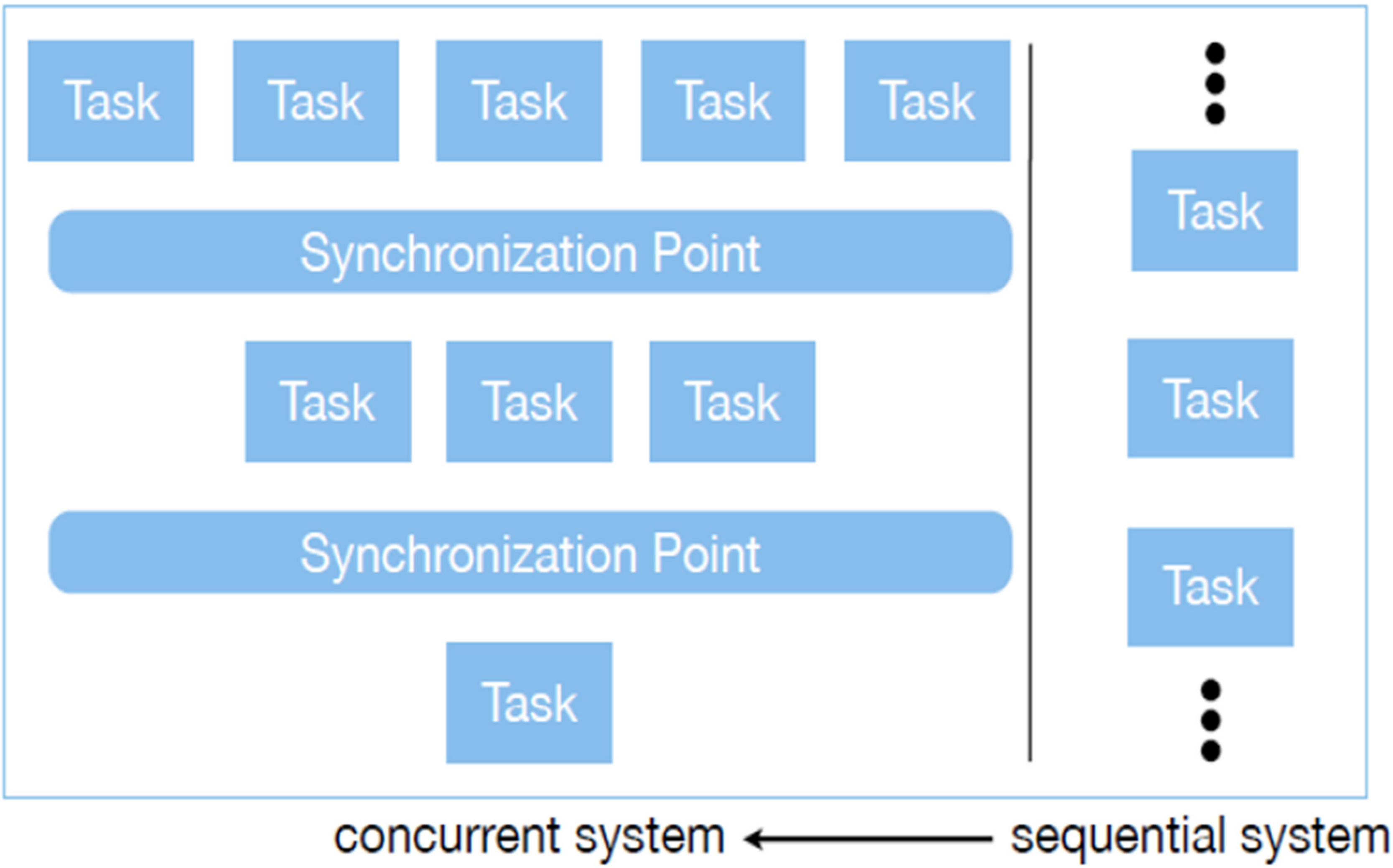
Performance Tuning

- Tuning threaded code typically involves
  - identifying sources of contention on locks (synchronization)
  - identifying work imbalances across threads
  - reducing overhead
- Testing and Tuning
  - Whenever you tune a threaded program, you must test it again for correctness
- Going back further
  - if you are unable to tune system performance,
  - you may have to re-design and re-implement

Design Models

- Two primary design models for concurrent algorithms
- Task Decomposition
  - identify tasks (computations) that can occur in any order
  - assign such tasks to threads and run concurrently
- Data Decomposition
  - program has large data structures where individual data elements can largely be calculated independently
  - data decomposition implies task decomposition in these cases

Task Decomposition



Eight rules of Designing multithreaded APPS

1/8 Rules: Designing MT APPS

- **Identify Truly Independent Computations**
- If you can’t identify (in a single threaded application) computations that can be done in parallel, you’re out of luck
- Some situations that indeed can’t be made parallel

2/8 Rules: Designing MT APPS

- **Implement Concurrency at the Highest Level Possible**
- When discussing "What's Not Parallel" a common refrain was "you can't make this parallel,
  - So see if its part of a larger computation that CAN be made parallel"
- This is such good advice, it was promoted to being a guideline!
  - Two approaches: bottom up, top down

2/8 Rules: Bottom UP

- One methodology says to create a concurrent program
  - Start with a tuned, single-threaded program and
  - Use a profiler to find out where it spends most of its time
- In the bottom-up approach, you start at those “hot spots” and work up; typically, a hotspot will be a loop of some sort
  - See if you can thread the loop
  - If not, move up the call chain, looking for the next loop and see if it can be made parallel...
  - If so, still look up the call chain for other opportunities, first.
- Why? Granularity! You want coarse-grained tasks for your thread

2/8 Rules: Top Down

- With knowledge of the location of the hot spot
- Start by looking at the whole application and see if there are parallelization opportunities on the large-scale structure that contains the hot spot
  - if so, you've probably found a nice coarse-grained task to assign to your threads
  - If not, move lower in the code towards the hot spot, looking for the first opportunity to make the code concurrent

3/8 Rules: Designing MT APPS

- **Plan Early for Scalability**
- The number of cores will keep increasing
- You should design your system to take advantage of more cores as they become available
  - Make the number of cores an input variable and design from there
- In particular, designing systems via data decomposition techniques will provide more scalable systems
  - humans are always finding more data to process!
- More data, more tasks; if more cores arrive, you’re ready

4/8 Rules: Designing MT APPS

- **Make use of Thread-Safe Libraries Wherever Possible**
- First, software reuse!
  - Don’t fall prey to Not Invented Here Syndrome
  - if code already exists to do what you need, use it!
- Second, more libraries are becoming multithread aware
  - That is, they are being built to perform operations concurrently
- Third, if you make use of libraries, ensure they are thread-safe; if not, you’ll need to synchronize calls to the library
  - Global variables hiding in the library may prevent even this, if the code is not reentrant ; if so, you may need to abandon it

5/8 Rules: Designing MT APPS

- **Use the Right Threading Model**
- Avoid the use of explicit threads if you can get away with it
- They are hard to get right
- Look at libraries that abstract away the need for explicit threads
  - OpenMP, Cilk and Intel Threading Building Blocks
  - Scala's agent model, Go's go routines and Clojure's concurrency primitives
- All of these models hide explicit threads from the programmer Right Threading Model

6/8 Rules: Designing MT APPS

- **Never Assume a Particular Order of Execution**
- With multiple threads, as we've seen, the scheduling of atomic statements is nondeterministic
- If you care about the ordering of one thread's execution with respect to another, you have to impose synchronization
- But, to **get the best performance**, you want to **avoid synchronization** as much as possible
- In particular, you want high granularity tasks that don't require synchronization
  - This allows your cores to run as fast as possible on each task they're given

7/8 Rules: Designing MT APPS

- **Use Thread-Local Storage Whenever Possible or Associate Locks with specific data**
- Related to Rule 6; the more your threads can use thread-local storage, the less you will need synchronization
- Otherwise, associate a single lock with a single data item
  - in which a data item might be a huge data structure
- This makes it easier for the developer to understand the system;
  - "if I need to update data item A, then I need to acquire lock A first"

8/8 Rules: Designing MT APPS

- **Dare to Change the Algorithm for a Better Chance of Concurrency**
- Sometimes a tuned, single-threaded program makes use of an algorithm which is not amenable to parallelization
- They might have picked that algorithm for performance reasons
  - Strassen's Algorithm O(n2.81) vs. the triple-nested loop algorithm to perform matrix multiplication O(n3)
- Change the algorithm used by the single-threaded program to see if you can then make that new algorithm concurrent
  - BUT: when measuring speedup, compare to the original!!

Java Threads

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);
        }
    }
}
```

A Runnable Object

- The Thread object's **run()** method calls the Runnable object's **run()** method

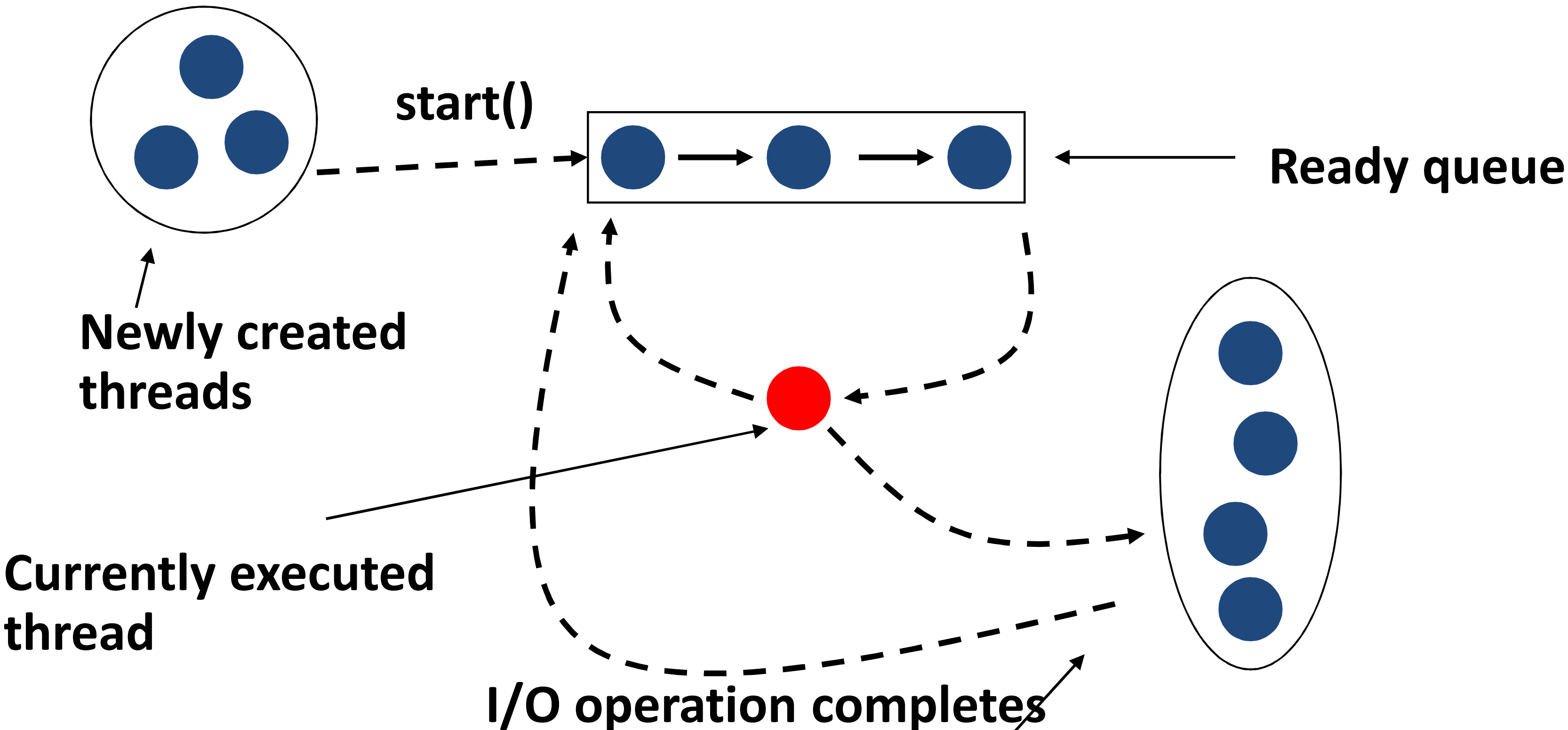
- Allows threads to run inside any object, regardless of inheritance

Example - an applet that is also a thread

Starting the Threads

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

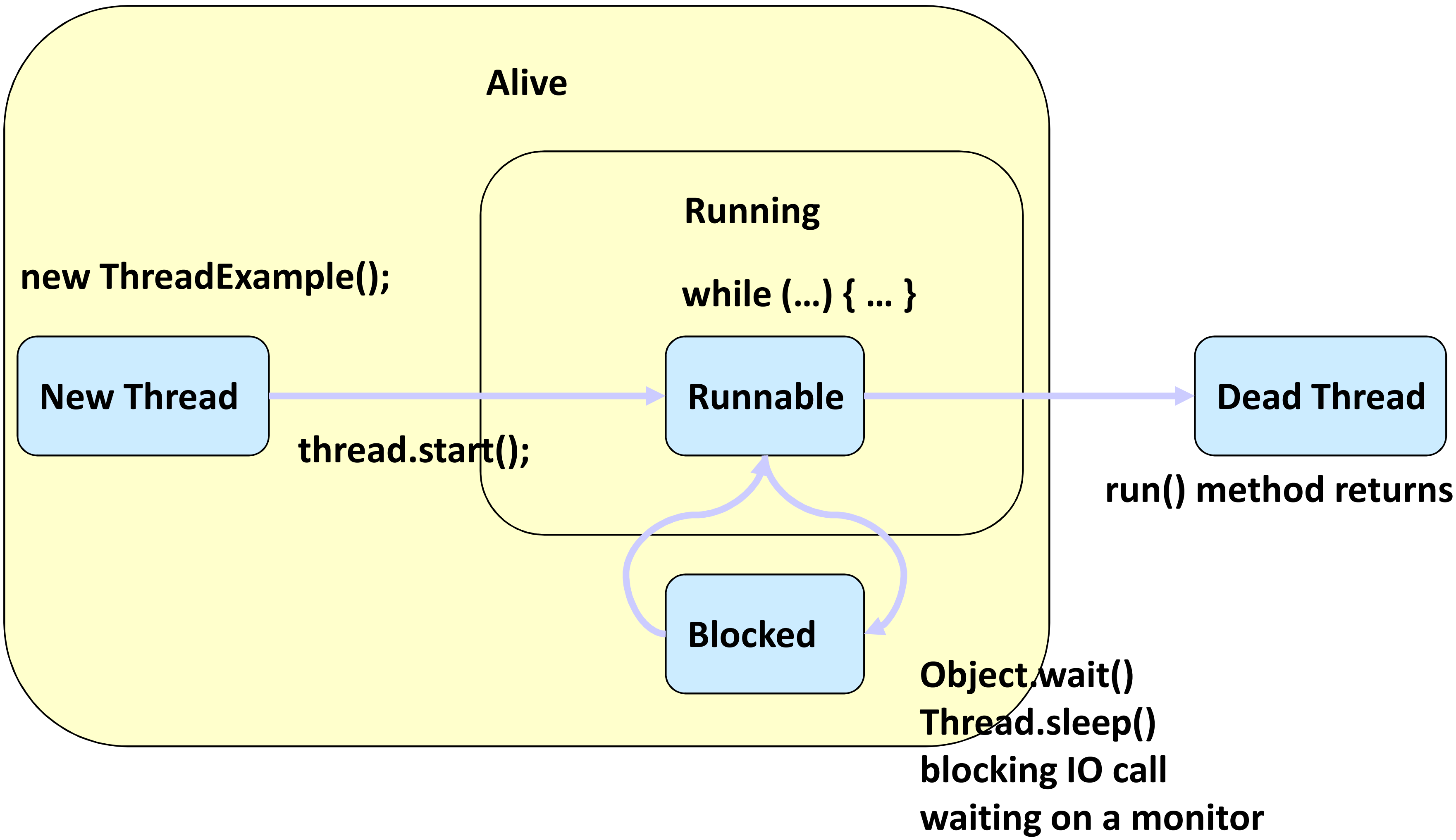
Scheduling Threads



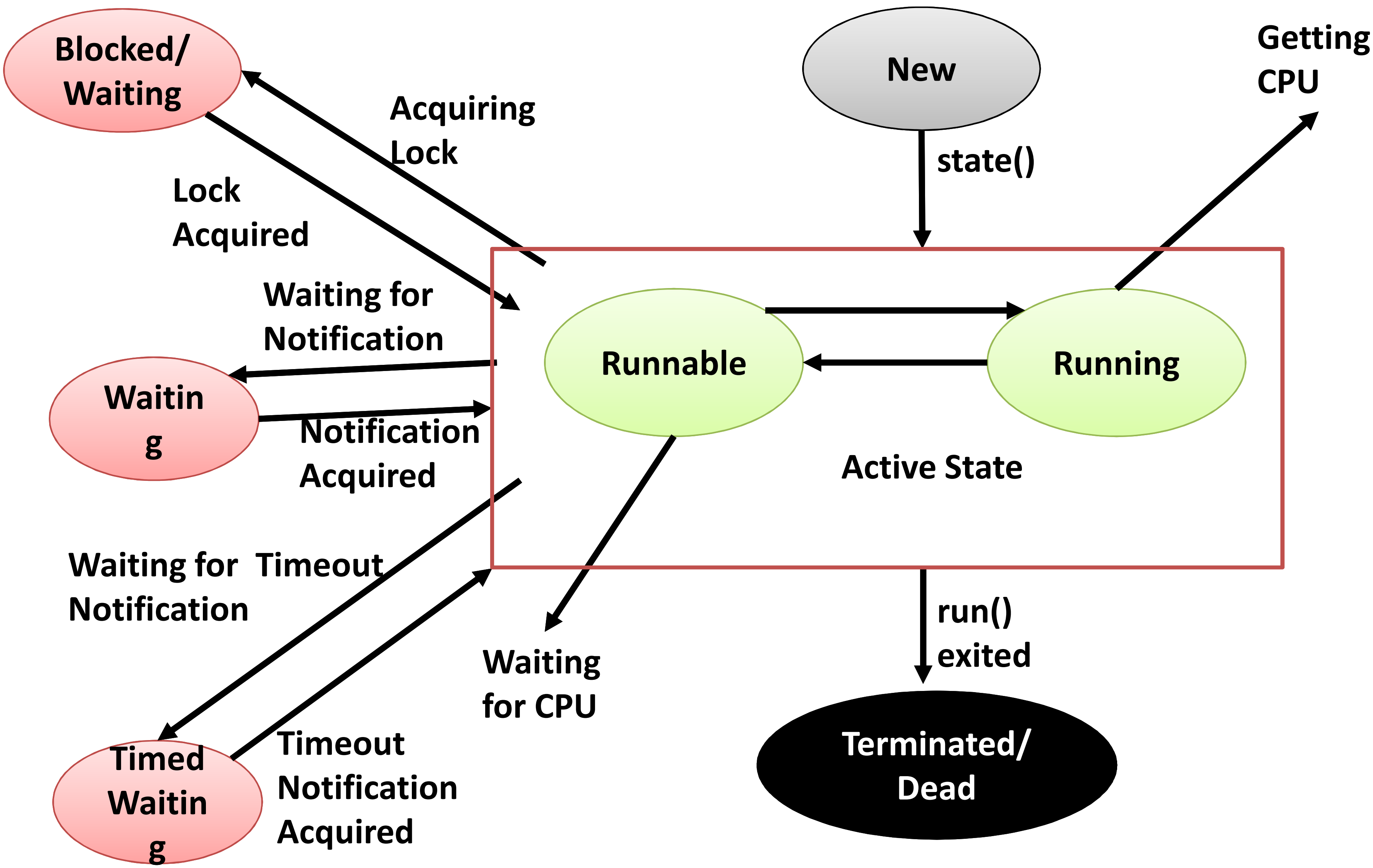
What happens when a program with a ServerSocket calls accept()?

- Waiting for I/O operation to be completed
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

Thread State Diagram



Java thread State: Life cycle



First Example: MainExtend.java

```
public class MainExtend extends Thread {
  public static void main(String[] args) {
    MainExtend thread = new MainExtend();
    thread.start();
    System.out.println("This code is
                         outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

First Example: MainImplRun.java

```
public class MainImplRun implements Runnable {
  public static void main(String[] args) {
    MainImplRun obj = new MainImplRun();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the
thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Example 1

```
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 1 (cont)

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

Java thread Example 2

```
class NewThread implements Runnable {
   Thread t;
   NewThread() {
         t = new Thread(this, "Demo Thread");
         System.out.println("Child thread: " + t);
         t.start();       // Start the thread
         }
public void run() {
         for(int i = 5; i > 0; i--) {
                  System.out.println("Child Thread: " + i);
         }
}
```

Java final keywords

- Keyword **final** : non access modifier
- Different context final are used
  - Variable : To create constant variable
  - Method : to prevent method overriding
  - Class: to Prevent inheritance

Vector Sum Example

```
import java.util.concurrent.*;
 public class VectorSum  {
private static final int MAX = 16; // Size of array
private static final int MAX_THREAD = 4;
 private static int[] a = { 1, 5, 7, 10, 12, 14, 15, 18, 20, 22, 25, 27, 30, 64, 110, 220 };
 private static int[] sum = new int[MAX_THREAD];
 private static int part = 0;
  static class SumArray implements Runnable {
        @Override
       public void run() {
           // Each thread computes sum of 1/4th of array
          int thread_part = part++;
           for (int i = thread_part * (MAX / 4); i < (thread_part + 1) * (MAX / 4); i++) {
              sum[thread_part] += a[i];
          }
       }
    }
```

Vector Sum Example

```
// Driver Code
public static void main(String[] args) throws InterruptedException {
    Thread[] threads = new Thread[MAX_THREAD];
    // Creating 4 threads
    for (int i = 0; i < MAX_THREAD; i++) {
        threads[i] = new Thread(new SumArray());
        threads[i].start();
    }
    // Joining 4 threads i.e. waiting for all 4 threads to complete
    for (int i = 0; i < MAX_THREAD; i++) {   threads[i].join();    }
    // Adding sum of all 4 parts
    int total_sum = 0;
    for (int i = 0; i < MAX_THREAD; i++) {   total_sum += sum[i];   }
    System.out.println("sum is " + total_sum);
}
}
```

Basic thread: Command line Args

```
import java.util.concurrent.*;
 public class BasicThread  {
   private static int part = 0;
   static class PrintThread implements Runnable {
      @Override
      public void run() {
         int thread_part = part++;
             char c= (char) (65+thread_part);
         for (int i = 0; i< 10; i++) { System.out.print(c); }
      }
   }
```

Basic thread: Command line Args

```
// Driver Code
  public static void main(String[] args) throws InterruptedException {
    for(int i=0;i<args.length;i++)  System.out.println(args[i]);  //parse
     MAX_THREAD=Integer.parseInt(args[0]); //First Argument : for running
with four thread Use command : java BasicThread 4
     Thread[] threads = new Thread[MAX_THREAD];
          System.out.println("Number of thread created..="+MAX_THREAD);
     for (int i = 0; i < MAX_THREAD; i++) {
        threads[i] = new Thread(new PrintThread());
        threads[i].start();
     }
    for (int i = 0; i < MAX_THREAD; i++) {    threads[i].join();     }
  }
```

Mutlithreaded Counter

```
 // Java Program to demonstrate synchronization in Java
class Counter {
    private int c = 0; // Shared variable
    // Synchronized method to increment counter
    public synchronized void inc() {
        c++;
    }
    // Synchronized method to get counter value
    public synchronized int get() {
        return c;
    }
}
```

Mutlithreaded Counter

- Contains a private integer count as the shared resource.
- The increment **method is synchronized,**
  - **Ensuring that only one thread can execute it at a time, preventing concurrent modifications.**

Mutlithreaded Counter

```
public class CounterSyncMethod {
  public static void main(String[] args) {
    Counter cnt = new Counter(); // Shared resource
     Thread t1 = new Thread(() -> {
       for (int i = 0; i < 1000; i++) {  cnt.inc();}  });
    Thread t2 = new Thread(() -> {
      for (int i = 0; i < 1000; i++) {   cnt.inc(); }  });

    t1.start();    t2.start(); // Start both threads
    try {    t1.join();    t2.join();
    } catch (InterruptedException e) {  e.printStackTrace();  }

     System.out.println("Counter: " + cnt.get());
  }
}
```

Mutlithreaded Counter: Sync Block

```
// Java Program to demonstrate synchronization block in Java
class Counter {
    private int c = 0; // Shared variable
    // Method with synchronization block
    public void inc() {
        synchronized(this) { // Synchronize only this block
            c++;
        }
    }
    public int get() {    return c;    }
}
```

Another Example: Ticket Booking

```
// thread synchronization for Ticket Booking System
class TicketBooking {
    private int availableTickets = 10; // Shared resource (available tickets)
    // Synchronized method for booking tickets
    public synchronized void bookTicket(int tickets) {
        if (availableTickets >= tickets) {
            availableTickets -= tickets;
            System.out.println("Booked " + tickets + " tickets, Remaining tickets: " + 
availableTickets);
        } else {
            System.out.println("Not enough tickets available to book " + tickets);     }
    }
    public int getAvailableTickets() {    return availableTickets;    }
}
```

Another Example: Bank Balance

```
class BankAccount {// Java Program to demonstrate Process Synchronization
    private int balance     = 1000; // Shared resource (bank balance)
    // Synchronized method for deposit operation
    public synchronized void deposit(int amount)   {
        balance += amount;
        System.out.println("Deposited: " + amount     + ", Balance: " + balance);
    }
    // Synchronized method for withdrawal operation
    public synchronized void withdraw(int amount)    {
        if (balance >= amount) {   balance -= amount;
            System.out.println("Withdrawn: " + amount   + ", Balance: " + balance);
        }
        else {    System.out.println(  "Insufficient bal to withdraw: “    + amount);    }
    }
    public int getBalance() { return balance; }
}
```

Synchronized Method using Anonymous Class

```
import java.io.*;
class Test {
    synchronized void test_func(int n)    {
        // synchronized method
        for (int i = 1; i <= 3; i++) {
            System.out.println(n + i);
            try {   Thread.sleep(100);   }
            catch (Exception e) {  System.out.println(e);  }
        }
    }
}
```

Driver Code

```
public class SyncPrintAsync {
 public static void main(String args[])    {
      // only one object
      final Test O = new Test();
      Thread a = new Thread() {
                      public void run() { O.test_func(15); }};
      Thread b = new Thread() {
           public void run() { O.test_func(30); } };
      a.start();
      b.start();
   }
}
```

Another Sync Example

```java
import java.io.*;
// A Class used to send a message
class Sender {
    public void send(String msg)    {
        System.out.println("Sending " + msg);
        try {   Thread.sleep(100);    }
        catch (Exception e) {
            System.out.println("Thread  interrupted.");
        }
        System.out.println(msg + "Sent");
    }
}
```

Another Sync Example: Contd

```
// Class for sending a message using Threads
class ThreadedSend extends Thread {
    private String msg;
    Sender sender;
    // Receives a message object and a string message to be sent
    ThreadedSend(String m, Sender obj)    {
        msg = m;        sender = obj;
    }
    public void run() { // Only one thread can send a msg at a time.
        synchronized (sender) {  // Synchronizing the send object
            sender.send(msg);
        }
    }
}
```

Another Sync Example: Contd

```
// Driver class
class SyncMessg {
    public static void main(String args[])  {
        Sender send = new Sender();
        ThreadedSend S1 = new ThreadedSend("Hi ", send);
        ThreadedSend S2 = new ThreadedSend("Bye ", send);
        // Start two threads of ThreadedSend type
        S1.start();    S2.start();
        // Wait for threads to end
        try {    S1.join();      S2.join();    }
        catch (Exception e) {   System.out.println("Interrupted");   }
    }
}
```