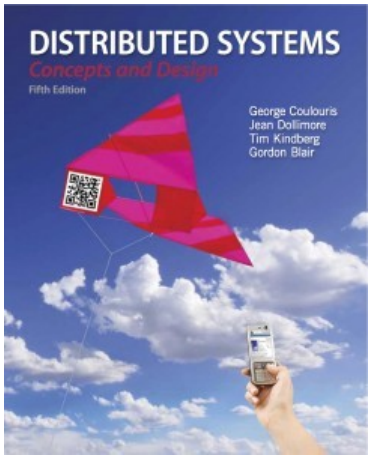
Week 4
Process and Thread Management – Operating System Support



Reference:

Chapter 7

Distributed Systems: Concepts and Design

Coulouris, Dollimore, Kindberg and Blair

Edition 5, © Addison Wesley 2011

Learning Objectives

- Explain what a modern operating system does in support of distributed applications and middleware.
  - Network operating systems
  - Distributed operating systems
  - Supporting distributed applications by the combination of middleware and NOSs
- Recognise OS abstractions for resource management.
  - Process execution environment
  - Multiple processes and threads mechanisms.

Learning Objectives

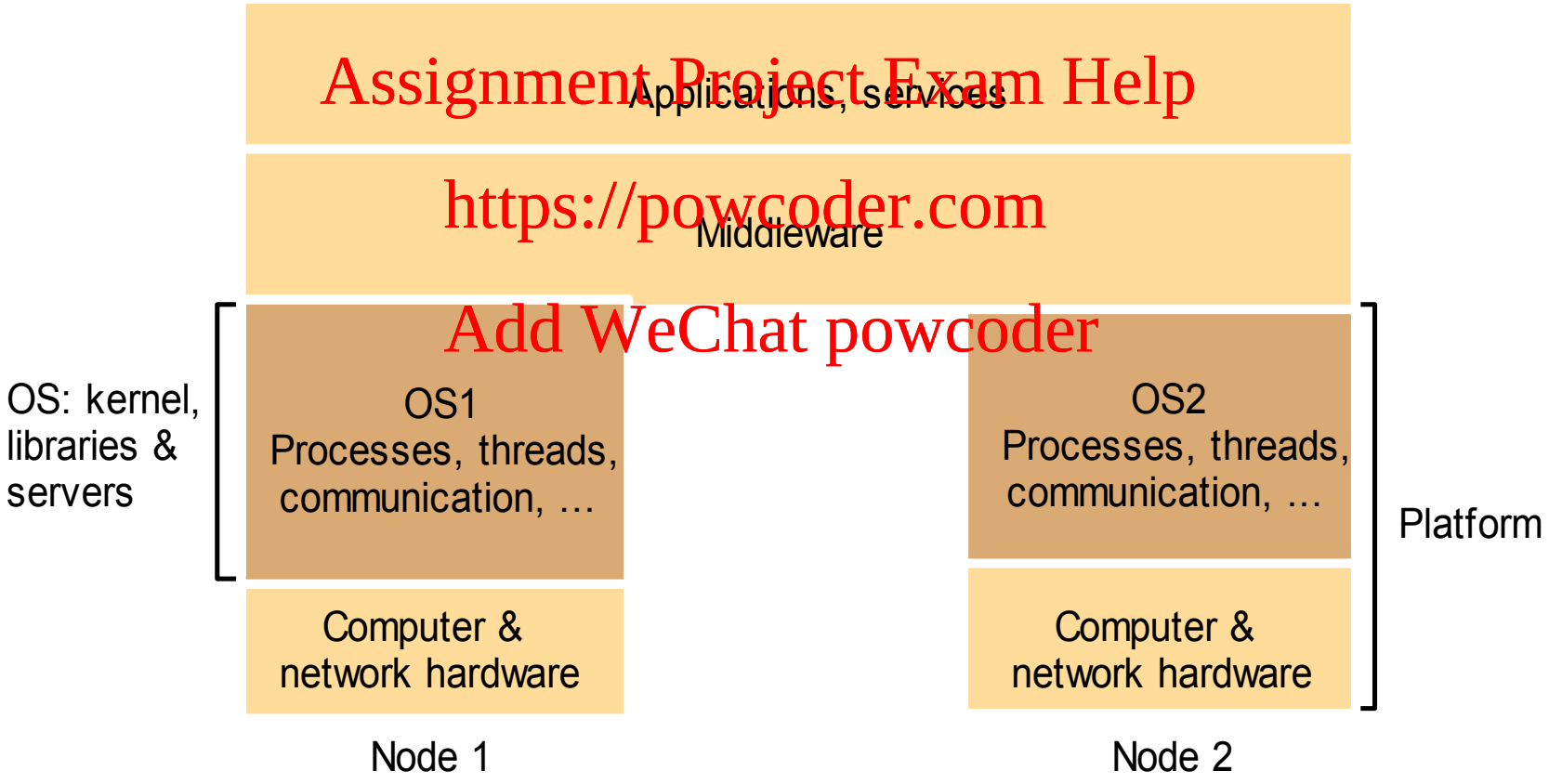
- Develop Java multi-thread programming for client/server applications.
- Appraise interprocess communication and invocation in terms of:
  - Performance issues and factors affecting performance
  - Potential OS or middleware supports for performance
  - Synchronous and asynchronous invocations

Middleware and NOSs

- The Middleware Layer
  - The client application invokes operations on another process, which are often on another node.
  - Middleware provides remote invocation between objects or processes at the nodes of a distributed system.
  - The requirements of middleware are to be met by the operating systems.
    - Efficient and robust access to physical resources.
    - Flexibility to implement a variety of resources-management policies.

Middleware and NOSs

- The System layers



Middleware and NOSs

- The operating systems
  - An operating system provides abstractions of the underlying physical resources (processors, memory, communication, and storage media).
  - It simplifies, protects and optimizes the use of resources.
  - The *network operating systems* have a networking capability built into them and so can be used to access remote resources.
  - Examples of NOSs include UNIX, Windows, and Linux.

Middleware and NOSs

- The operating systems
  - Access is network-transparent for some types of resources, such as files.
  - Nodes running a network operating system retain autonomy in managing their own processing resources.
  - With a network operating system, there are multiple system images/views, one per node.
  - An operating system that produces a single system image/view for all resources on all nodes is called a *distributed operating system*.

Middleware and NOSs

- The operating systems
  - A DOS presents users (and applications) with an integrated computing platform that hides the individual computers.
  - A DOS has control over all of the nodes (computers) in the network and allocates their resources to tasks without users' involvement.
  - With a DOS, the user doesn't know (or care) where his programs are running.
  - Examples include:
    - Cluster computer systems
    - V system, Sprite, Globe OS

Middleware and NOSs

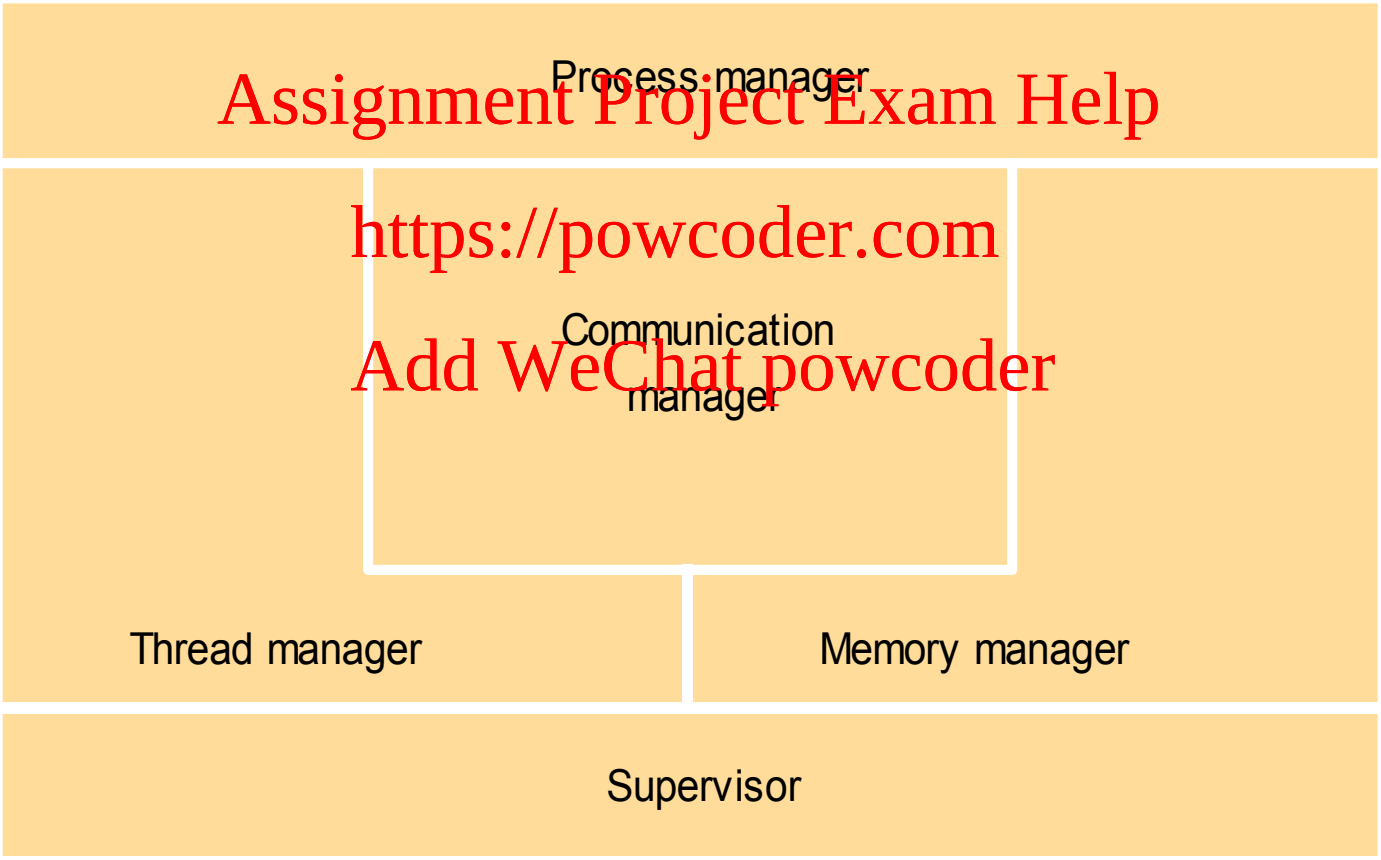
- The operating systems
  - Two reasons hinder the popularity of DOSs
    - Users are not willing to adopt a new operating system that will not run their legacy applications.
    - Users are not willing to lose autonomy by adopting DOSs.
  - The combination of middleware and NOSs is a solution of autonomy and network-transparency.
  - The Middleware
    - Runs on a variety of OS-hardware combinations at nodes of a distributed system.
    - Utilises local resources to implement its mechanism for remote invocations between objects or processes at the nodes.
  - Users will only be satisfied if their middleware-OS combination has good performance.

Middleware and NOSs

- The operating systems
  - A NOS provides the following interfaces:
    - Encapsulation of the basic resources
    - Protection of the resources used by applications
    - Concurrent processing to enable applications to complete their work concurrently
  - A NOS uses a combination of libraries, kernels and servers to perform
    - Communication
    - Scheduling
    - Process management
    - Thread management
    - Memory management

Middleware and NOSs

- The OS components



Resource Protection

- Resources require protection from illegitimate accesses.
  - Maliciously contrived codes (security system cares)
  - Benign codes with bugs (OS cares)
  - Examples like file read and write attributes
- To protect resources
  - The OS kernel runs in supervisor (privileged) mode and has complete privileges for physical resources.
  - The kernel arranges other process in user (unprivileged) mode for execution.

Resource Protection

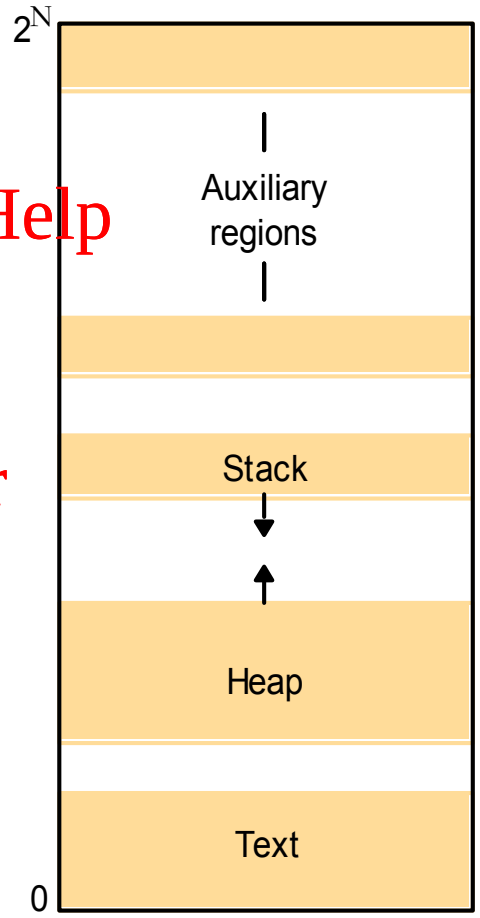
- To protect resources
  - The kernel sets up address spaces to protect itself and other process.
  - An address space is a collection of ranges of virtual memory locations.
  - Access rights are applied to address space.
  - A process can not access memory outsides its address space.
  - A process can safely transfer to the kernel address space via a system call trap to execute the kernel codes.

Process and Thread

- A process consists of an execution environment together with one or more threads.
  - An execution environment is a unit of resource management, consisting of
    - An address space
    - Thread synchronisation and communication resources (semaphores and sockets)
    - High-level resources (open files)

Process and Thread

- A address space consists of the following regions.
  - The text region contains the program codes.
  - A heap is initialised by values in the program binary file.
  - A stack contains dynamically created values.



Process and Thread

- In distributed systems, the creation of a new process can be on a local node or on a remote node to utilise remote resources.
- The transfer policy determines whether to situate a new process locally or remotely.
- The location policy determines which node should host the new process selected for transfer.
  - Static: based on expected long-term characteristics.
  - Adaptive: considering the current system state.

Process and Thread

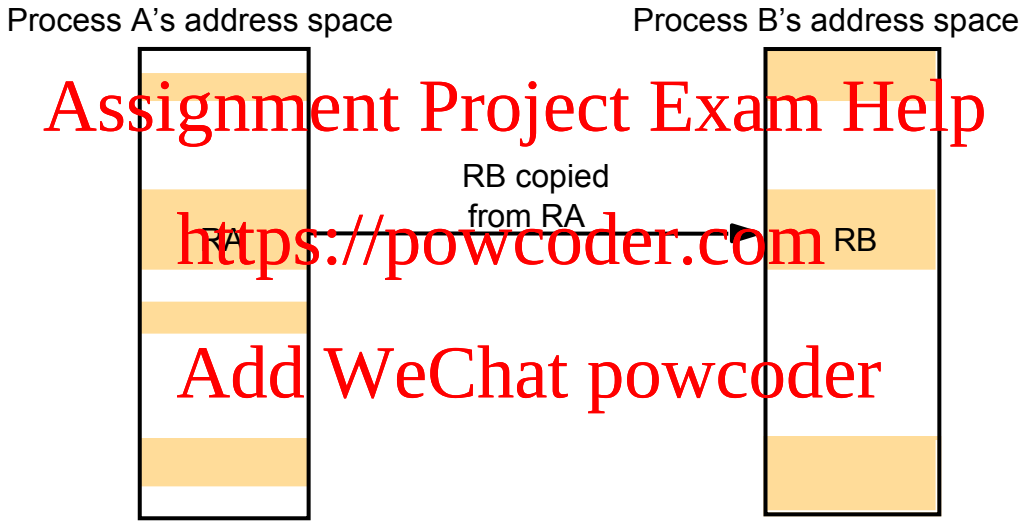
- Load-sharing systems can be:
  - Centralised
    - one load manager
  - Hierarchical
    - Managers make process allocation decisions as far down the tree as possible
    - Managers may transfer processes to one another via a common ancestor
  - Decentralised
    - Nodes exchange information with one another directly to make allocation decision.

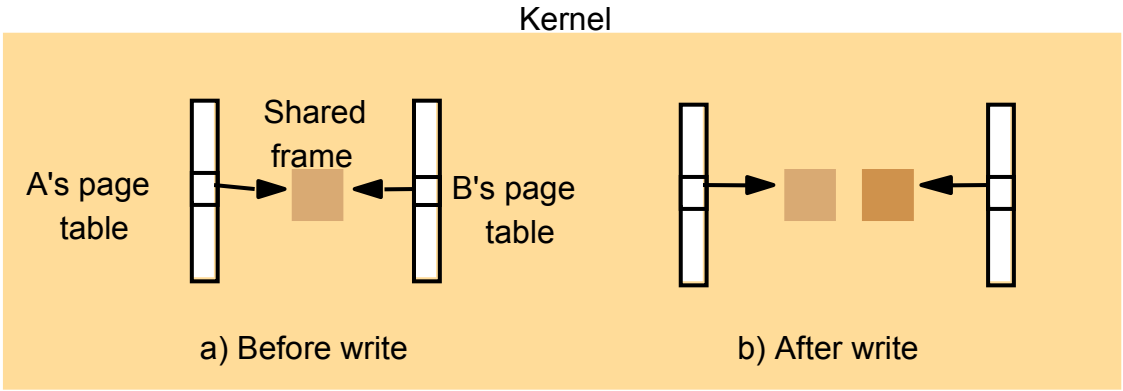
Process and Thread

- Load-sharing systems can be:
  - Sender-initiated
  - Receiver-initiated
- When a new process is crested, it requires a execution environment.
  - Initialised from an executable file
  - Copied from an existing environment, such as UNIX fork semantics
- Migratory load-sharing systems can shift load at any time, not just when a new process is created.

Process and Thread

- Copy-on-write is an optimization for copying execution environment.



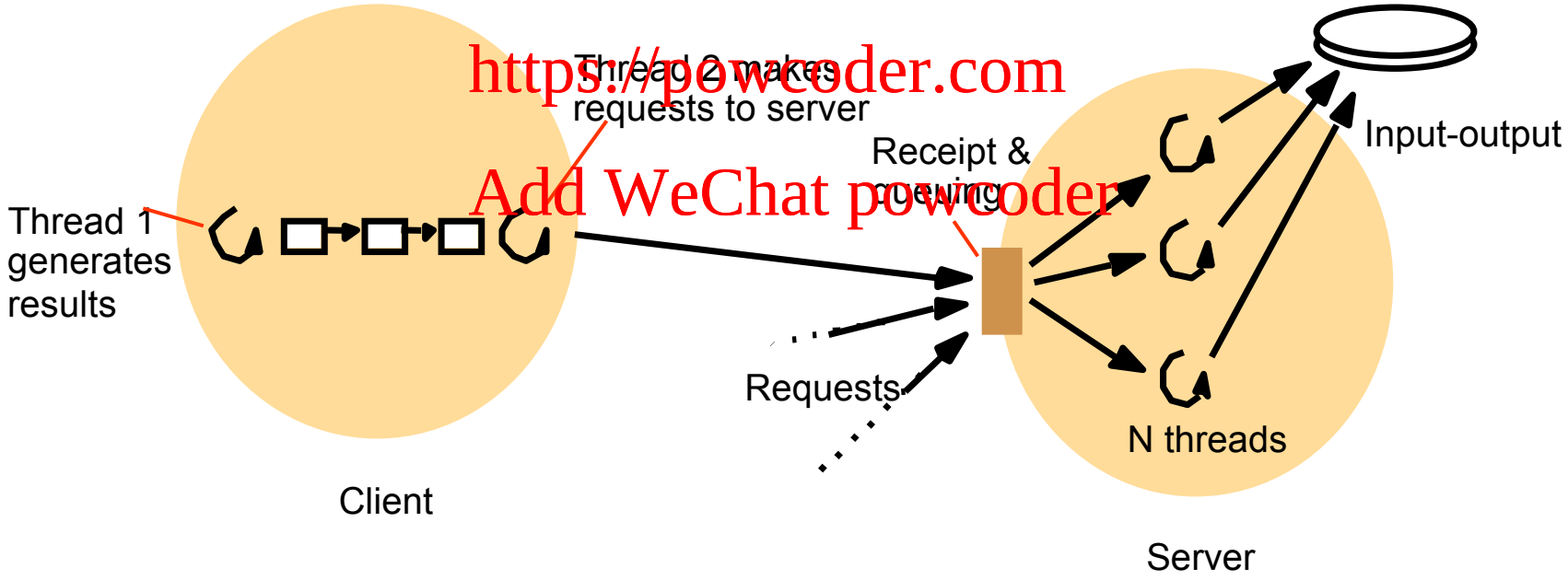


Process and Thread

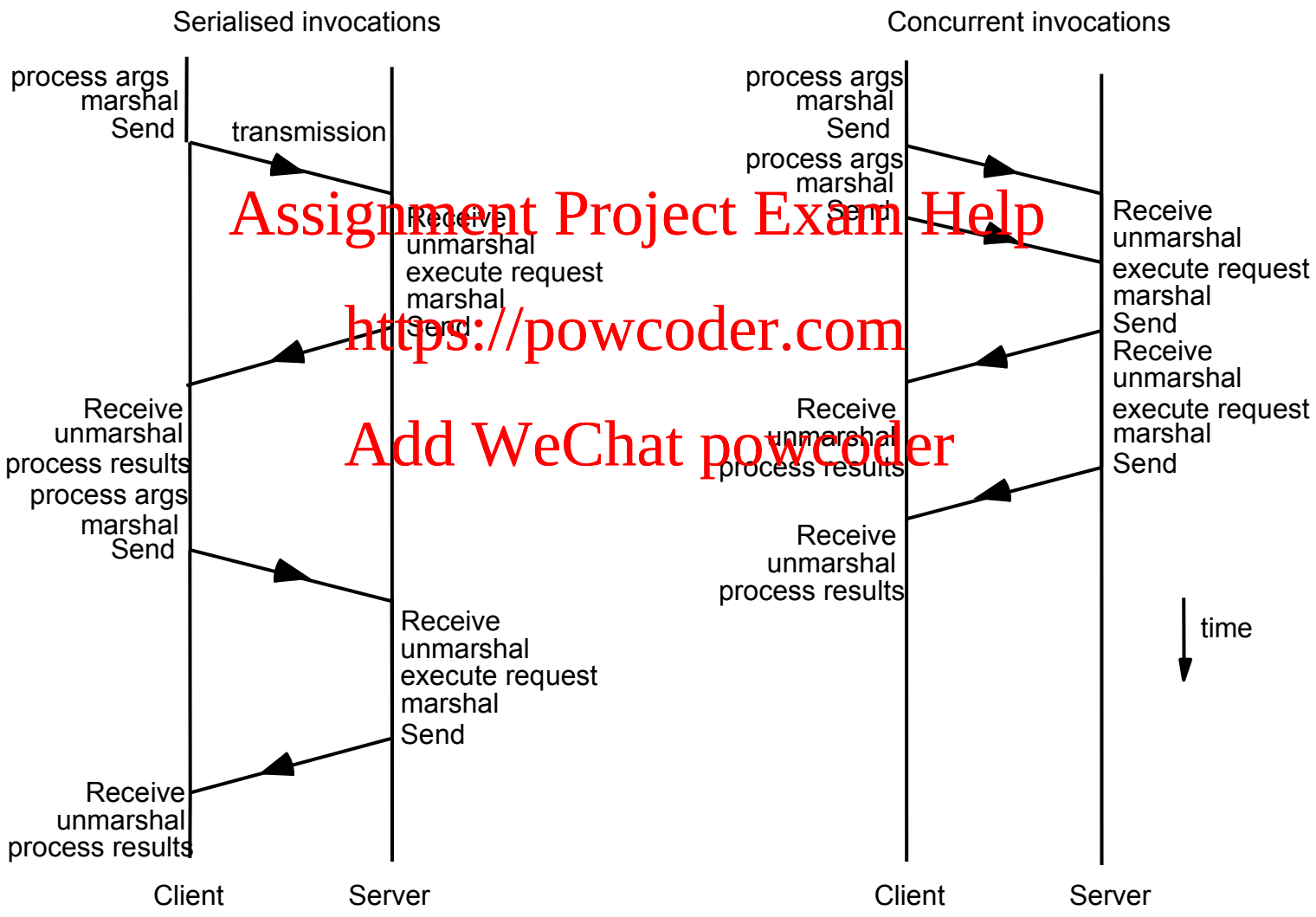
- Threads are sometimes called *lightweight processes* within a process.
- Threads share the creator process' resources, including memory and open files.
- Threads make use of concurrency to increase processing efficiency.
  - The client has two threads, one for preparing requests; the other for sending requests.
  - The server has a pool of threads, each of which removes a request from the queue and process it.

Process and Thread

- Client and server with multiple threads



Process and Thread

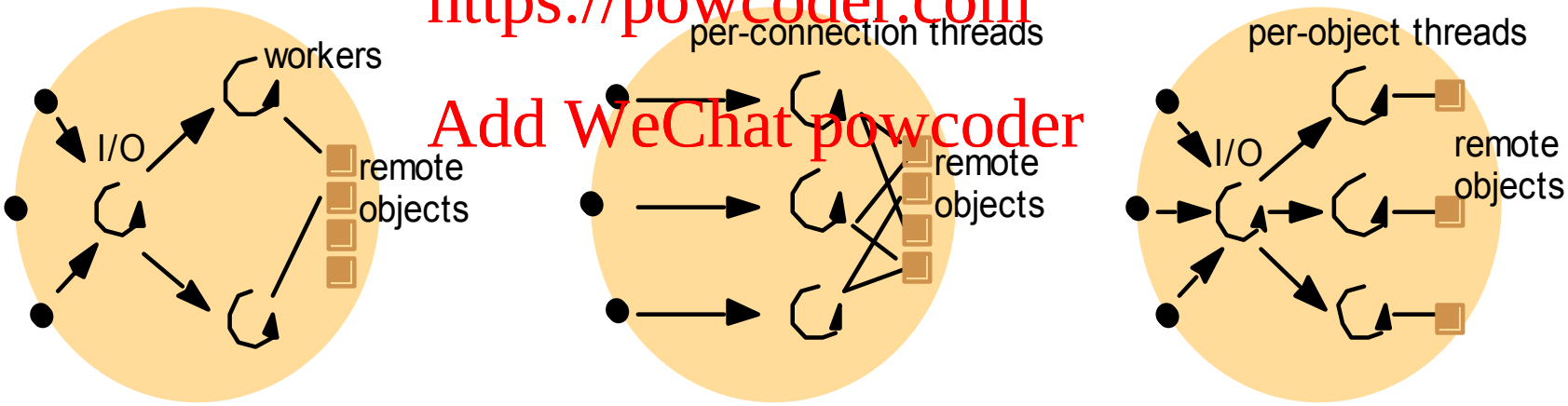


Process and Thread

- The architecture for multi-threaded servers concerns the various ways of mapping requests to threads within a server.
  - The ***worker pool*** architecture uses a fixed size pool of worker threads.
  - The ***thread-per-request*** architecture creates a worker thread for each request.
  - The ***thread-per-connection*** architecture associates a worker thread with each connection.
  - The ***thread-per-object*** architecture associates a worker thread with each remote object.

Process and Thread

- The architecture for multi-threaded servers



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Process and Thread

- Threads versus multiple processes
  - Creating a thread is (much) cheaper than creating a process (10-20 times).
  - Switching to a different thread in same process is (much) cheaper (5-50 times)
  - Threads within same process can share data and other resources more conveniently and efficiently (without copying or messaging)
  - Threads within a process are not protected from each other.

Process and Thread

- Threads within client
  - The web browser is a good example of using threads within client.
  - A web pages typically contains several images.
  - The browser has to fetch each of these images in a separate HTTP GET request.
  - The browser does not need to obtain the images in a particular sequence.
  - The browser can make concurrent request by using multiple threads.
  - At the same time the main thread can continue its own tasks such as image rendering.

Process and Thread

- Threads versus multiple processes

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Process and Thread

- Threads programming
  - Threads programming is concurrent programming.
  - Thread synchronization is necessary to maintain data consistency.
  - A thread has its own lifetime from creation (such as using ***new*** operation in Java) to end (such as its ***destroy()*** method is called in Java).
  - Threads can be assigned with priorities, with higher priority thread has more chance to be scheduled for execution than a lower priority one.

Process and Thread

- Threads programming
  - Threads scheduling can be
    - Preemptive: a thread may be suspended at any time to make a way to another thread.
    - Non-preemptive: a thread may call the threading system to yield processor.
- Java Threads programming
  - Java provides methods for creating, destroying and synchronizing threads.
  - A Java thread is defined by extending the class ***Thread*** or implementing interface: ***Runnable.***
  - A thread is run by calling its ***start()*** methods, which calls its ***run()*** method.
  - The functional methods are defined in the ***run()*** method.

Process and Thread

- Java threads constructor and management methods.

`Thread(ThreadGroup group, Runnable target, String name)`
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

`setPriority(int newPriority), getPriority()`
Set and return the thread's priority.

`run()`
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

`start()`
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

`sleep(int millisecs)`
Cause the thread to enter the *SUSPENDED* state for the specified time.

`yield()`
Enter the *READY* state and invoke the scheduler.

`destroy()`
Destroy the thread.

Process and Thread

- Java threads synchronization calls

`thread.join(int millisecs)`
Blocks the calling thread for up to the specified time until *thread* has terminated.

`thread.interrupt()`
Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

`object.wait(long millisecs, int nanosecs)`
Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

`object.notify(), object.notifyAll()`
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Process and Thread

- Java threads example
  - This example consists of a client program and a server program, which are modified versions of the **`TCPClient`** and **`TCPServer`** on page 156-157.
  - The modification is to demonstrate the *thread-per-connection* architecture.
  - In this example, the server dynamically creates one thread for each client connection, which accepts multiple requests and replies responses.
  - In this demonstration, we use two clients, each of which sends 5 requests and receives 5 responses in a single connection.

Process and Thread

The TCP Client

```
import java.net.*;
import java.io.*;
public class TCPClient {
  public static void main (String args[]) {
     //arguments supply message, hostname of destination and client ID
    Socket s=null;
      try{
        int serverPort=7896;
        s=new Socket(args[1], serverPort);
        DataInputStream in = new DataInputStream(
                                    s.getInputStream());
        DataOutputStream out =new DataOutputStream(
                                    s.getOutputStream());
        for (int i=1; i<=5 ;i++ ){
          String request="Client "+args[2]+": "+args[0]+" "+i;
          out.writeUTF(request);
          String data=in.readUTF();
         System.out.println("Received: "+ data) ;
       }
```


☞ This program continues on the next slide

Process and Thread

- The TCP Client

```
        }catch (UnknownHostException e){
           System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){
           System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){
           System.out.println("IO:"+e.getMessage());
        }finally {
           if(s!=null)
             try {s.close();}
             catch (IOException e){
               System.out.println("close:"+e.getMessage());}}
        }
}
```


The end of this program

Process and Thread

- The TCP Server

```
import java.net.*;
import java.io.*;
public class TCPServer {
  public static void main (String args[]) {
    try{
      int serverPort=7896;
      ServerSocket listenSocket=new
                        ServerSocket(serverPort);
      int i=0;
      while(true) {
        Socket clientSocket=listenSocket.accept();
        Connection c = new Connection(clientSocket, i++);
        System.out.println("Thread " +i+ " is created");
      }
    } catch(IOException e) {
       System.out.println("Listen :"+e.getMessage());}
    }
}
```


☞ This program continues on the next slide

Process and Thread

- The TCP Server

```
class Connection extends Thread {
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;
  int thrdn;
  public Connection (Socket aClientSocket, int tn){
    try {
      thrdn=tn;
      clientSocket = aClientSocket;
      in=new DataInputStream(
              clientSocket.getInputStream());
      out=new DataOutputStream(
              clientSocket.getOutputStream());
      this.start();
    } catch(IOException e){
       System.out.println("Connection:“
                          +e.getMessage());}
}
```


☞ This program continues on the next slide

Process and Thread

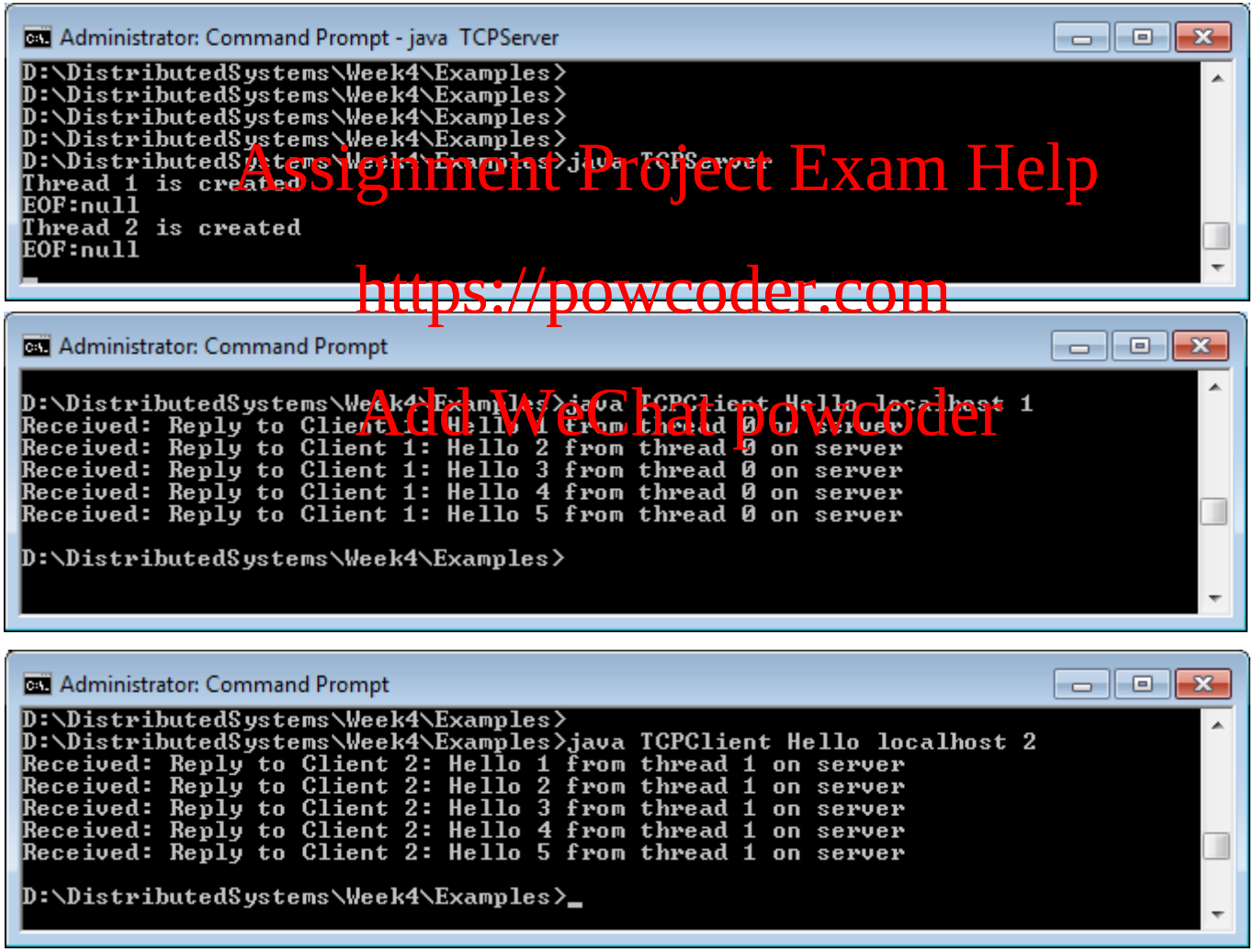
- The TCP Server

```
        public void run(){
          try {
            String data;
            while ((data=in.readUTF())!=null){
            out.writeUTF("Reply to "+data+ " from thread
                                  "+thrdn+" on server");
            }
          } catch(EOFException e) {
              System.out.println("EOF:"+e.getMessage());
          } catch(IOException e){
              System.out.println("IO:"+e.getMessage());
          } finally {
            try {clientSocket.close();}
            catch(IOException e){/*close failed*/}}
        }
}
```


The end of this program

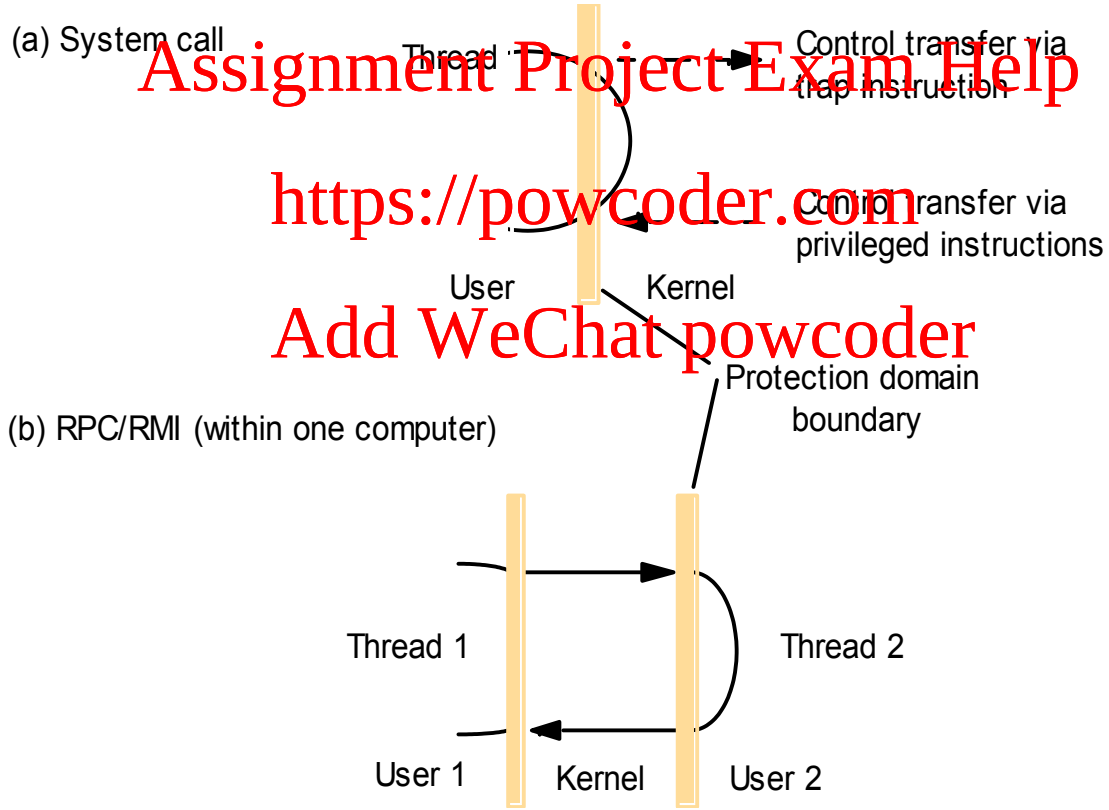
Process and Thread

- Output from the ***TCPClient*** and the ***TCPServer***



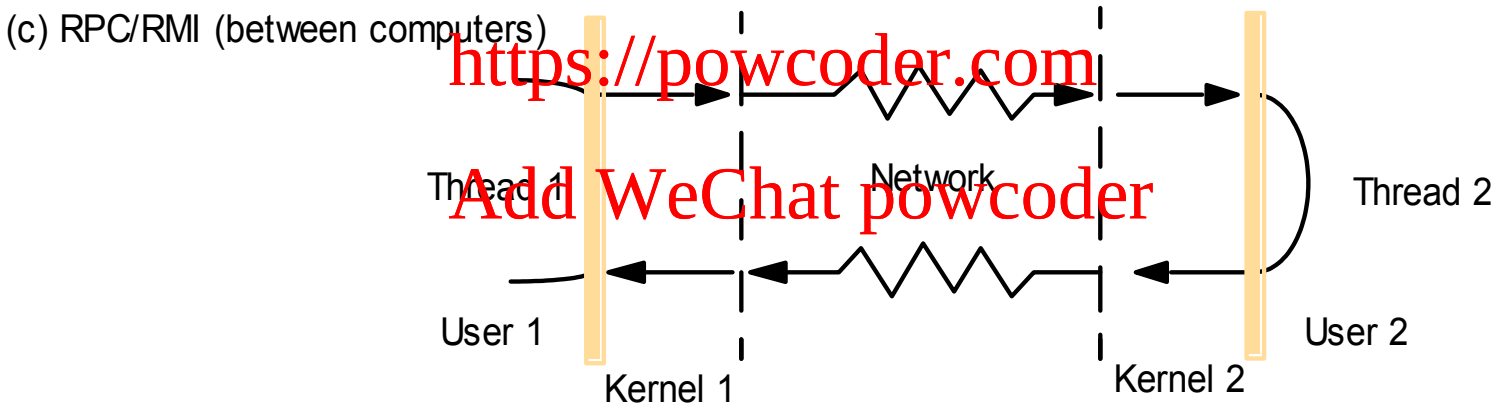
Communication and Invocation

- Invocations between address spaces



Communication and Invocation

- Invocations between address spaces



Communication and Invocation

- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
  - A local *null procedure call* is less than 1 microseconds.
  - A Remote *null procedure call* is about 10 milliseconds.
    - Network time (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only 0.01 millisecond.
    - The remaining delays must be in OS and middleware latency.

Communication and Invocation

- Factors affecting RPC/RMI performance
  - Marshalling, unmarshalling, operations and despatch at the server
  - Data copying from application to kernel space to communication buffers.
  - Thread scheduling, context switching, and kernel entry.
  - Protocol processing for each protocol layer
  - Network access delays of connection setup, and network latency
- Concurrent and asynchronous invocations
  - Middleware or applications are made asynchronous if they don't block, waiting for reply to each invocation.

Communication and Invocation

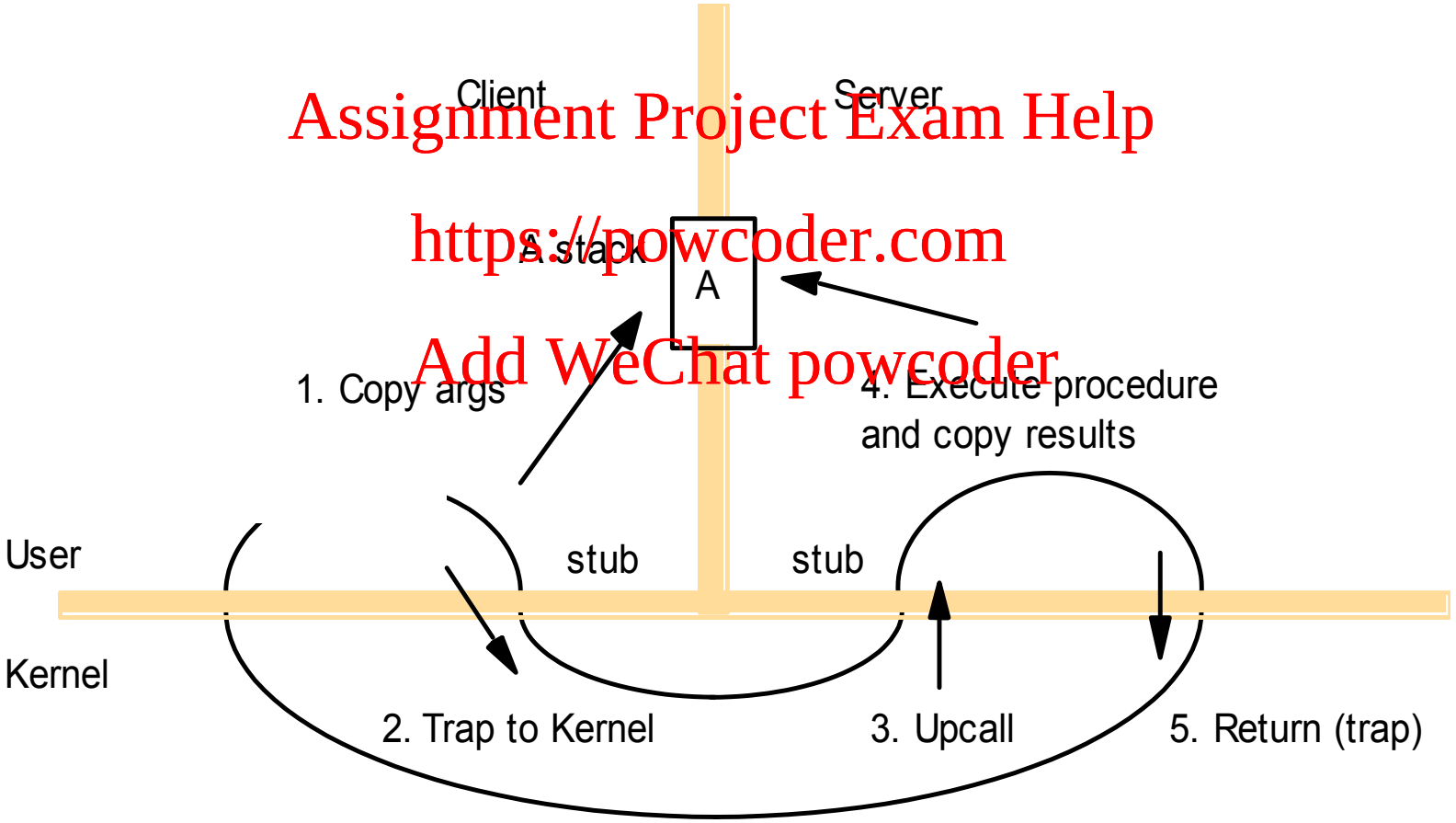
- Most middleware such as CORBA, Java RMI, HTTP, is implemented over TCP.
  - Supporting universal availability, unlimited message size and reliable transfer.
  - Sun RPC (used in NFS) is implemented over both UDP and TCP and generally works faster over UDP.
- Research-based systems have implemented much more efficient invocation.
  - Firefly RPC (www.cdk3.net/oss)
  - Amoeba's doOperation, *getRequest*, *sendReply* primitives (www.cdk3.net/oss)
  - LRPC [Bershad et. al. 1990]

Communication and Invocation

- LRPC (Lightweight RPC)
  - Uses shared memory for interprocess communication, while maintaining protection of the two processes.
  - Arguments copied only once rather than four times for conventional RPC.
    - Client stub stack to a message
    - The message to a kernel buffer
    - The kernel buffer to a server message
    - The message to the server stub stack
  - Client threads can execute server code via protected entry points only.
  - Up to 3 x faster for local invocations.

Communication and Invocation

- LRPC (Lightweight RPC)



Communication and Invocation

- Asynchronous invocations
  - An asynchronous invocation is the one that is performed asynchronously with respect to the caller.
  - An asynchronous invocation returns as soon as the invocation request message has been created and is ready for dispatch.
  - Asynchronous invocation is applicable
    - When the client does not require response.
    - When the client uses a separate call to collect results.
  - Middleware or applications are made asynchronous if they don't block, waiting for reply to each invocation.

Summary

- The combination of middleware and NOSs is a solution of autonomy and network-transparency.
- Modern OSs provide fundamental services for resource management, such as memory, process and thread, and communication management, on which middleware is built in support of distributed applications.
- Multiple threading is a cheaper way to implement concurrency than multiple processes, where Java is powerful tool for concurrent programming.
- Performance issues should be taken into account when designing/using middleware.