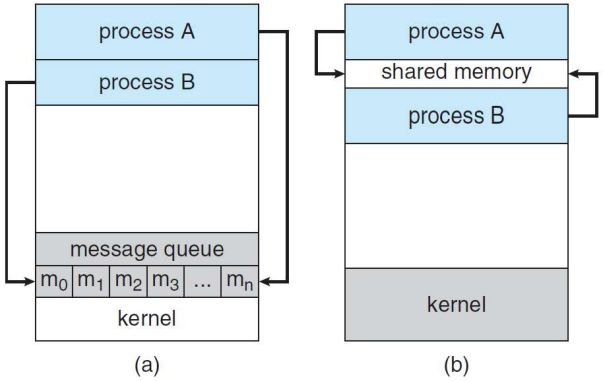
CIIC 4050 / ICOM 5007
Operating Systems

Lecture 4:
Interprocess Communication

Interprocess Communication (IPC)

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **IPC**
- Two models:
  - Shared memory
  - Message passing

Communications Models



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem (1)

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - ***unbounded-buffer*** places no practical limit on the size of the buffer
  - ***bounded-buffer*** assumes that there is a fixed buffer size

45

Producer-Consumer Problem (2)

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use `BUFFER_SIZE - 1` elements

46

Bounded Buffer - Producer

```
item next_produced;
while (true) {
  /* produce an item in next produced */
  while (((in + 1) % BUFFER_SIZE) == out);
  /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

NOTE: This solution might suffer from synchronization problems that we shall address later in the course … but for the moment …

47

Bounded Buffer - Consumer

```
item next_consumed;
while (true) {
   while (in == out)
      ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   /* consume the item in next consumed */
}
```


48

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

49

IPC – Message Passing (1)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

50

IPC – Message Passing (2)

- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

51

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

52

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

53

Indirect Communication

- Messages using mailboxes (or ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

54

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **send**(*A, message*) – send a message to mailbox A
- **receive**(*A, message*) – receive a message from mailbox A

55

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

56

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **send** sender blocks until the message is received
  - **receive** receiver blocks until a message is available
- **Non-blocking** is considered **asynchronous**
  - **send** sender sends message and continue
  - **receive** receive a valid message or null

57

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of *n* messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

58

Examples of IPC Systems - POSIX

- Process first creates shared memory segment
  - `segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);`
  - **`See also shm_open()`**
- Process wanting access to that shared memory must attach to it
  - `shared memory = (char *) shmat(id, NULL, 0);`
- Now the process could write to the shared memory
  - `sprintf(shared memory, "Writing to shared memory");`
- When done a process can detach the shared memory from its address space
  - `shmdt(shared memory);`

59

IPC Systems – Windows XP (1)

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels

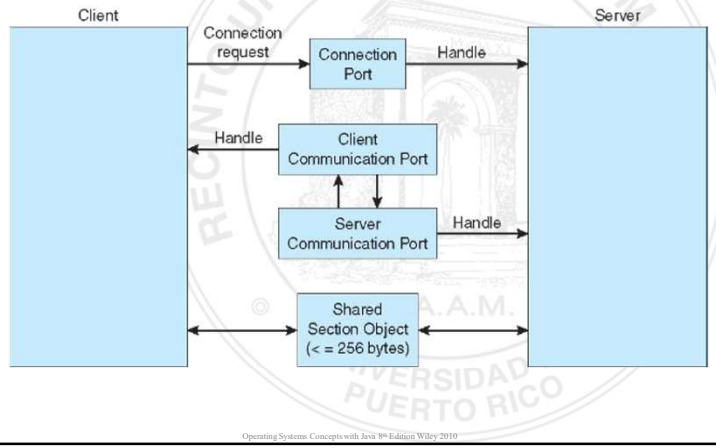
60

IPC Systems – Windows XP (2)

- Communication works as follows:
  - The client opens a handle to the subsystem's connection port object
  - The client sends a connection request
  - The server creates two private communication ports and returns the handle to one of them to the client
  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

61

LPCs in Windows XP



62

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

63

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8**:**1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a (unique) pair of sockets

64

Sockets (2)

- Two widely used address domains:
  - **unix domain**, 2 processes share a common file system
  - **internet domain**, 2 processes running on any two hosts on the internet communicate
- Two widely used socket types
  - **stream sockets**, treat communications as a continuous stream of characters
  - **datagram sockets** read entire messages at once

65

Socket Communication (Server)

http://www.linuxhowtos.org/C_C++/socket.htm

```
/* A simple server using TCP Sockets
   The port number used is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     char *z;

    FILE *file;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }

     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");

     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
```

66

Socket Communication (Server)

http://www.linuxhowtos.org/C_C++/socket.htm

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,
          sizeof(serv_addr)) < 0)
      error("ERROR on binding");

    listen(sockfd,5);
    clilen = sizeof(cli_addr);

    while((newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr,
              &clilen)) >= 0){
       if (newsockfd < 0)
          error("ERROR on accept");

       bzero(buffer,256);

       n = read(newsockfd,buffer,255);
       if (n < 0) error("ERROR reading from socket");

       printf("Here is the message: %s\n",buffer);
       z = inet_ntoa(*(struct in_addr *)&cli_addr); /* cast as a struct in_addr */

       printf("Client information: %s\n", z);

       file = fopen("clients.txt","a+");
       fprintf(file,"%s\t%s",z,buffer);
       fclose(file);

       n = write(newsockfd,"I got your message\nMessage: %",18);
       if (n < 0) error("ERROR writing to socket");
    }
    printf("Server Terminated\n");
    return 0; }
```

67

Socket Communication (Client)

http://www.linuxhowtos.org/C_C++/socket.htm

```
/* A simple client using TCP Sockets
   The server address and port number
   are passed as arguments          */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>


int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;


    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }

    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
```

68

Socket Communication (Client)

http://www.linuxhowtos.org/C_C++/socket.htm

```
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);

    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");

    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
         error("ERROR writing to socket");

    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```


Socket Communication in Java (1)

```
public class DateServer {
   public static void main (String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);


         // now listen for connections
         while (true) {
            Socket client = sock.accept();


            PrintWriter pout = new
            PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
           client.close();
         }
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
```

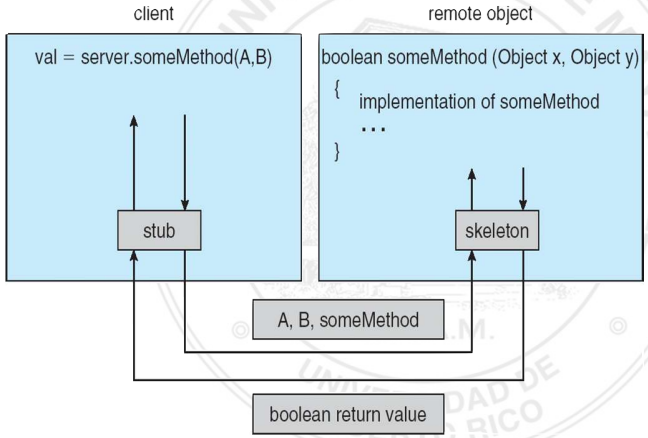

Socket Communication in Java (2)

```
public class DateClient {
   public static void main (String[] args) {
      try {
         // make connection to server socket
         Socket sock = new Socket("127.0.0.1", 6013);


         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));


         // read the Date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

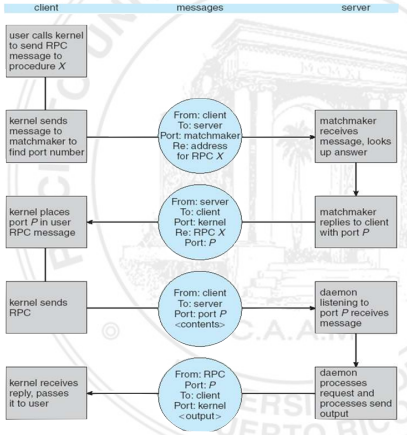

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems; built on top of local IPC mechanism
- **Stubs** – client-side proxy for the actual procedure on the server, typically one stub per each remote procedure
- The client-side stub locates the server and *marshalls* the parameters (packaging in a form that can be transmitted, endianess…)
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

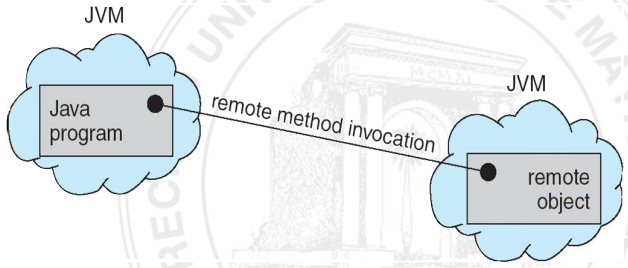
Marshaling Parameters



Execution of RPC



Remote Method Invocation



- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object

RMI Example

- Begin by declaring and interface that specifies the methods that can be invoked remotely…

```
public interface RemoteDate extends Remote {
   public abstract Date getDate() throws RemoteException;
}
```

RMI Example

```
public class RemoteDateImpl extends UnicastRemoteObject
                            implements RemoteDate {
   public RemoteDateImpl() throws Remote Exception {}

   public Date getDate() throws Remote Exception {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         Remote Date dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "DateServer"
         Naming.rebind("DateServer", dateSErver);
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```


RMI Example

```
public class RMIClient {
   public static void main(String[] args) {
      try {
         String host = "rmi://127.0.0.1/DateServer";


         RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
         System.out.println(dateServer.getDate());
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```