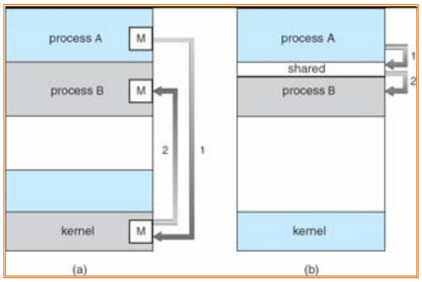
IPC and Intro to Networking

Interprocess Communication (IPC)

- 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*)
- 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)

Communications Models



Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a 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

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as 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

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

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.

Synchronization

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

Buffering

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

Networking . . .

- Packets
- LAN service
- Routers
- Internet service
- IP and NAT
- TCP and UDP service
- Port numbers

Networking . . .

- How TCP works
- Naming and DNS

TCP Java Client Code

```
import java.io.*;
import java.net.*;
class TCPClient {
   public static void main(String argv[]) throws Exception
   {
      Socket clientSocket = new Socket(“boo.cs.cornell.edu", 6789);
     DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
      BufferedReader inFromServer =
            new BufferedReader(new
                  InputStreamReader(clientSocket.getInputStream()));
      outToServer.writeBytes(stuff_to_write);
      stuff_to_read = inFromServer.readLine();
      clientSocket.close();
   }
}
```

TCP Java Server Code (listening thread)

```
import java.io.*;
import java.net.*;
class TCPServer {
  public static void main(String argv[]) throws Exception
    {
      ServerSocket listen_socket = new ServerSocket(6789);
      while(true) {
           Socket client_socket = listen_socket.accept();
           Connection c = new Connection(client_socket);
      }
    }
 }
```

TCP Java Server Code (spawned thread)

```
class Connection extends Thread {
        while(true) {
              BufferedReader inFromClient =
                   new BufferedReader(new
                   InputStreamReader(connectionSocket.getInputStream()));

             DataOutputStream  outToClient = new
                    DataOutputStream (connectionSocket.getOutputStream());

              inputString = inFromClient.readLine();
              …….
              outToClient.writeBytes(outputString);
        }
    }
```

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {
```

Create input stream

```
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));
```

Create client socket

```
      DatagramSocket clientSocket = new DatagramSocket();
```

Translate hostname to IP address using DNS

```
      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
```

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```
      DatagramPacket sendPacket =
        new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```
      clientSocket.send(sendPacket);

      DatagramPacket receivePacket =
        new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
      clientSocket.receive(receivePacket);

      String modifiedSentence =
          new String(receivePacket.getData());

      System.out.println("FROM SERVER:" + modifiedSentence);
      clientSocket.close();
      }
}
```

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {
```

Create datagram socket at port 9876

```
      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {
```

Create space for received datagram

```
          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
```

Receive datagram

```
           serverSocket.receive(receivePacket);
```

Example: Java server (UDP), cont

```
      String sentence = new String(receivePacket.getData());
```

Get IP addr port #, of sender

```
      InetAddress IPAddress = receivePacket.getAddress();

      int port = receivePacket.getPort();

                  String capitalizedSentence = sentence.toUpperCase();

      sendData = capitalizedSentence.getBytes();
```

Create datagram to send to client

```
      DatagramPacket sendPacket =
         new DatagramPacket(sendData, sendData.length, IPAddress,
                        port);
```

Write out datagram to socket

```
      serverSocket.send(sendPacket);
      }
    }
}
```

End of while loop, loop back and wait for another datagram