# Operating Systems (CSE531)
# Lecture # 08


Manish Shrivastava
LTRC, IIIT Hyderabad

# Outline

- Cooperating Processes
- Inter-process Communication
  - Shared Memory
  - Message Passing
- Communication between separate processes
- Methods to implement a link
- Client Server Communication
  - Sockets
  - Remote Procedure Call (RPC)
  - Pipes

# Cooperating Processes

- The processes can be independent or 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: Break into several subtasks and run in parallel
  - Modularity: Constructing the system in modular fashion.
  - Convenience: User will have many tasks to work simultaneously
    - Editing, compiling, printing

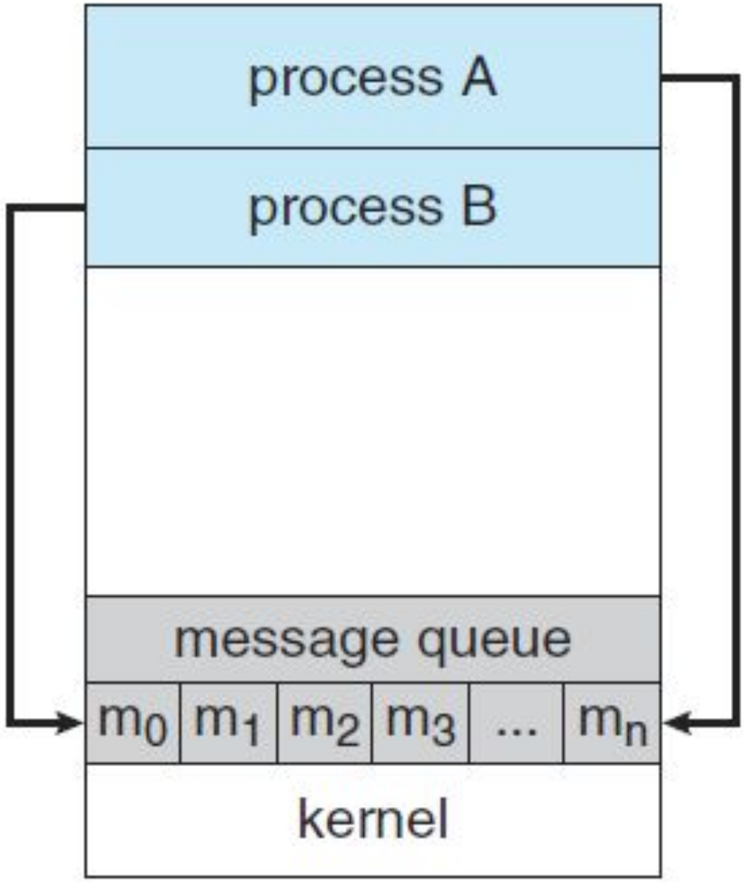
# Inter-process Communication (IPC)

- IPC facility provides a mechanism to allow processes to communicate and synchronize their actions.
- Processes can communicate through
  - **Shared Memory**
  - **Message Passing.**

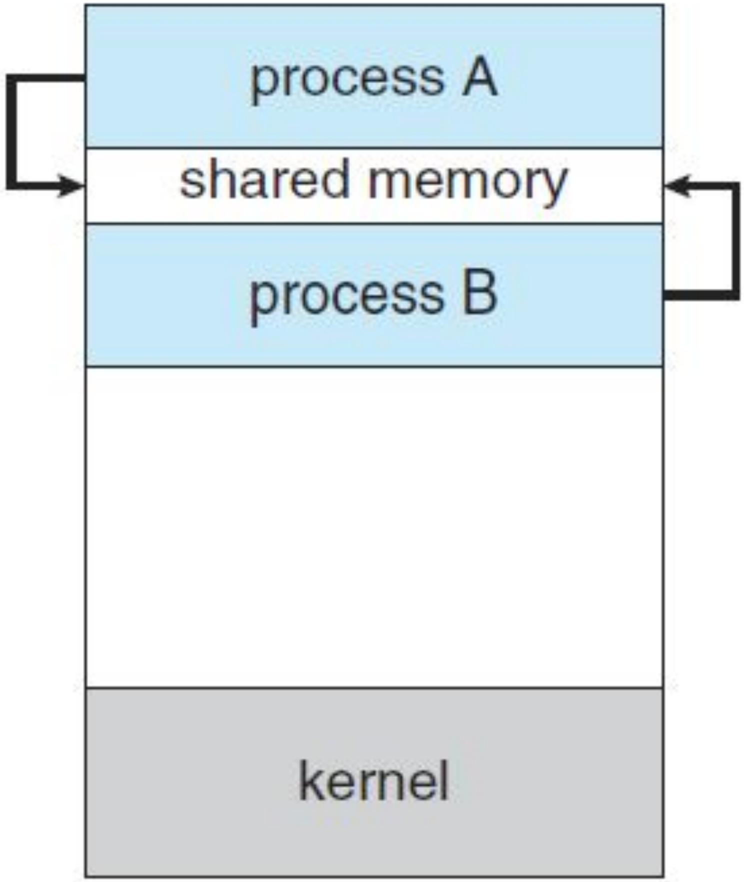
  Both schemes may exist in OS.

- The Shared-memory method requires communication processes to share some variables.
  - The responsibility for providing communication rests with the programmer.
  - The OS only provides shared memory.
- Example: producer-consumer problem.

# Communication Models




Message Passing




Shared Memory

# Producer-Consumer Problem: Shared memory

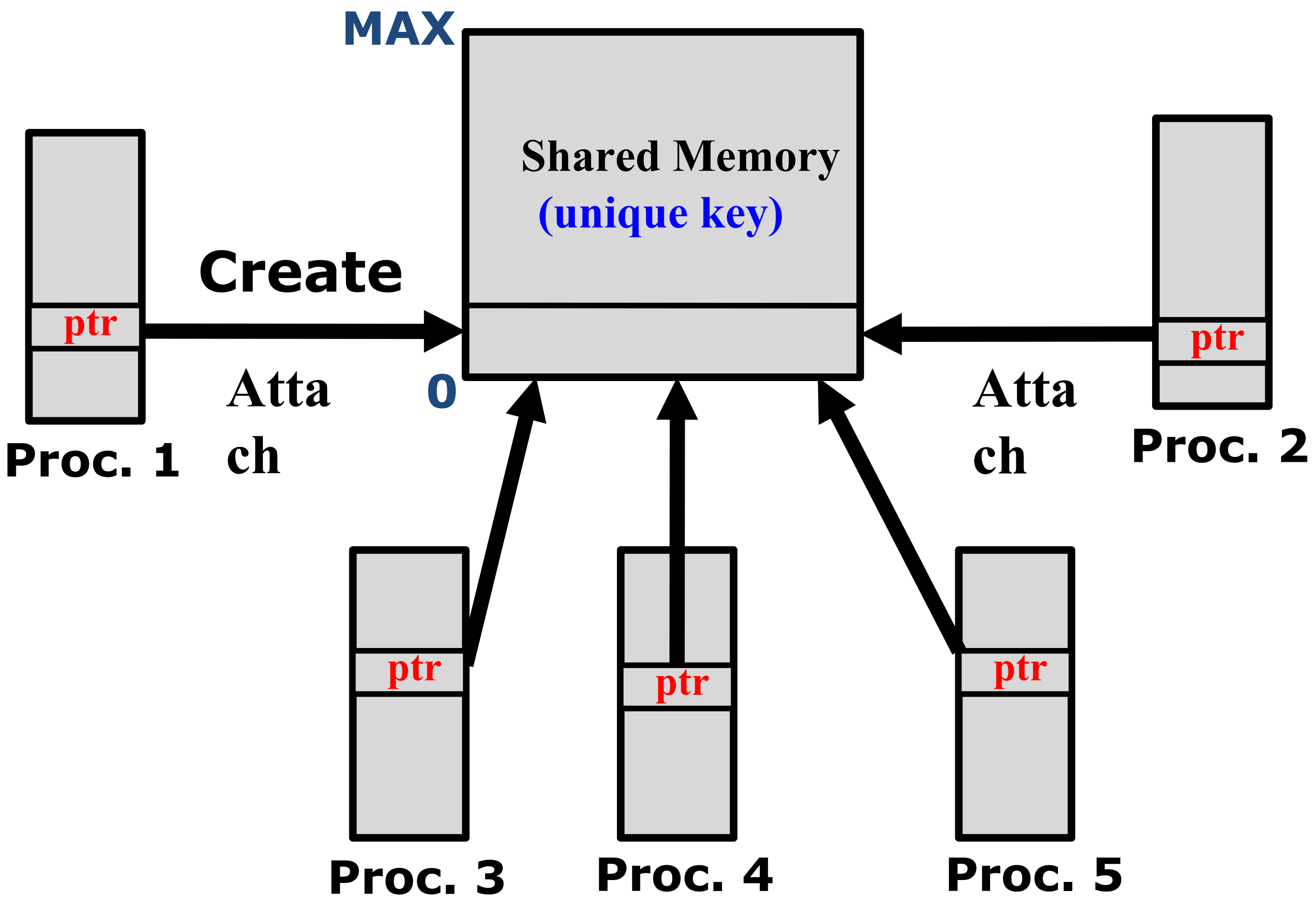
- *Producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
    - Producer can produce any number of items.
    - Consumer may have to wait
  - *bounded-buffer* assumes that there is a fixed buffer size.
    - Consumer or producer may have to wait

# Shared Memory

- Shared memory is memory that may be simultaneously accessed by multiple programs.

- Life cycle of Shared Memory
  - Create Memory Block
  - Attach to number of process simultaneously
  - Read & Write
  - Detach Memory block
  - Free Memory block

# Common chunk of read/write memory



# Example

***Shmget*** System Call

The shmget system call is used to create shared memory segment

***Shmat*** System Call

Attaches block to process

***int shmget (key_t key, int size, int shmflg);***

Arguments:

- ***key_t key***: key for creating or accessing shared memory
- ***int size***: size in bytes of shared memory segment to create. Use 0 for accessing an existing segment.
- ***int shmflg***: segment creation condition and access permission.

# Communicating Among Separate Processes

**Define the structure of a shared memory segment as follows:**

```
//Status of memory block
#define  NOT_READY  (-1)
#define  FILLED     (0)
#define  TAKEN      (1)

struct Memory {
   int  status;  // used for Synchronization
   int  data[4]; // data to share
};
```

# Communicating Among Separate Processes

*Prepare for a shared memory*

**The "Server"**

```
void main(int argc, char *argv[])
{
   key_t          ShmKEY;
   int            ShmID, i;
   struct Memory  *ShmPTR;

   ShmKEY = ftok(“./”, ‘x’);
   ShmID = shmget(ShmKEY, sizeof(struct Memory),
                  IPC_CREAT | 0666);
   ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
```

# Communicating Among Separate Processes

*shared memory not ready*

```
ShmPTR-->status = NOT_READY;
```

*filling in data*

```
for (i = 0; i < 4; i++)
    ShmPTR- -> data[i] = atoi(argv[i]);
```

```
ShmPTR-->status = FILLED;
while (ShmPTR-->status != TAKEN)
    sleep(1);   /* sleep for 1 second */
```

```
shmdt((void *) ShmPTR);
shmctl(ShmID, IPC_RMID, NULL);
```

```
exit(0);
}
```

*wait until the data is taken*

*detach and remove shared memory*

# Communicating Among Separate Processes

**The "Client"**

*prepare for shared memory*

```
void main(void)
{
    key_t          ShmKEY;
    int            ShmID;
    struct Memory  *ShmPTR;

    ShmKEY=ftok("./", 'x');
    ShmID = shmget(ShmKEY, sizeof(struct Memory), 0666);
    ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
    while (ShmPTR->status != FILLED)
        ;
    printf("%d %d %d %d\n", ShmPTR-->data[0],
        ShmPTR-->data[1], ShmPTR-->data[2], ShmPTR-->data[3]);
    ShmPTR->status = TAKEN;
    shmdt((void *) ShmPTR);
    exit(0);
}
```

# Communicating Among Separate Processes

- If you did not remove your shared memory segments (*e.g.*, program crashes before the execution of `shmctl()`), they will be in the system forever. This will degrade the system performance.
- Use the `Ipcs` command to check if you have shared memory segments left in the system.
- Use the `ipcrm` command to remove your shared memory segments.

# Inter-process Communication (IPC): Message Passing System

- **Message system** – processes communicate with each other without resorting to shared variables.

- If P and Q want to communicate, a communication link exists between them.

- OS provides this facility.

# IPC

- 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)
    - We are concerned with logical link.

# 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 a link unidirectional or bi-directional?
- Is the size of a message that the link can accommodate fixed or variable?

# Fixed and variable message size

- Fixed size message
  - Good for OS designer
  - Complex for programmer

- Variable size messages
  - Complex for the OS designer
  - Good for programmer

# Methods to implement a link

- Direct  or Indirect communication
- Synchronous or asynchronous  communication
- Automatic or explicit buffering

# Direct Communication

- **Symmetric:** Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q.
- **Asymmetric:** Only sender names the recipient, the recipient is not required to name the sender.
  - The send and receive primitives are as follows.
    - Send (P, message)– send a message to process P.
    - Receive(id, message)– receive a message from any process.
- 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
- **Disadvantages:** Changing the identifier of a process may necessitate examining all other process definitions. Any such ***hard-coding*** techniques, where identifiers must be explicitly stated, are less desirable.

# Indirect Communication

- The messages are sent and received from *mailboxes* (also referred to as *ports*).
- A mailbox is an object
  - Process can place messages
  - Process can remove messages.
- Two processes can communicate only if they have a shared mailbox.
- 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 a 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.
- A mailbox may be owned either by a process or by the operating system.
  - If the mailbox is part of the address space of the process,
    - We distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox).
    - Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox.
    - When a process that owns a mailbox terminates, the mailbox disappears.

# Indirect Communication

- Properties of a link:
  - A link is established if they have a shared mailbox
  - A link may be associated with more than two boxes.
  - Between a pair of processes there may be number of links
  - A link may be either unidirectional or bi-directional.
- OS owned mailboxes:
  - OS provides a facility
    - To create a mailbox
    - Send and receive messages  through mailbox
    - To destroy a mail box.
- The process that creates mailbox is a owner of that mailbox
- The ownership and send and receive privileges can be passed to other processes through system calls.

# Methods to implement a link

- Direct or Indirect communication
  - Direct link vs Mailbox
- Synchronous or asynchronous communication
- Automatic or explicit buffering

# Synchronous v/s asynchronous

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.
  - **Blocking send:** The sending process is blocked  until the message is received by the receiving process or by the mailbox.
  - **Non-blocking send:** The sending process sends the message and resumes operation.
  - **Blocking receive:** The receiver blocks until a message is available.
  - **Non-blocking receive:** The receiver receives either a valid message or a null.

# Methods to implement a link

- Direct or Indirect communication
  - Direct link vs Mailbox
- Synchronous or asynchronous communication
- Automatic or explicit buffering

# Automatic and explicit buffering

- A link has some capacity that determines the number of messages that can reside in it temporarily.

- Queue of messages is 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.

- In non-zero capacity cases a process does not know whether a message has arrived after the send operation.

# Automatic and explicit buffering

- The sender must communicate explicitly with receiver to find out whether the later received the message.
- Example: Suppose P sends a message to Q and executes only after the message has arrived.
- Process P:
  - send (Q. message)  : send message to process Q
  - receive(Q,message) : Receive message from process Q
- Process Q
  - Receive(P,message)
  - Send(P,"ack")

# Exception conditions

- When a  failure occurs error recovery (exception handling) must take place.
- Process termination
  - A sender or receiver process may terminate before a message is processed. (may be blocked forever)
  - A system will terminate the other process or notify it.
- Lost messages
  - Messages may be lost over a network
  - Timeouts; restarts.
- Scrambled messages
  - Message may be scrambled on the way due to noise
  - The OS will retransmit the message
  - Error-checking codes (parity check) are used.

# Client-Server Communication

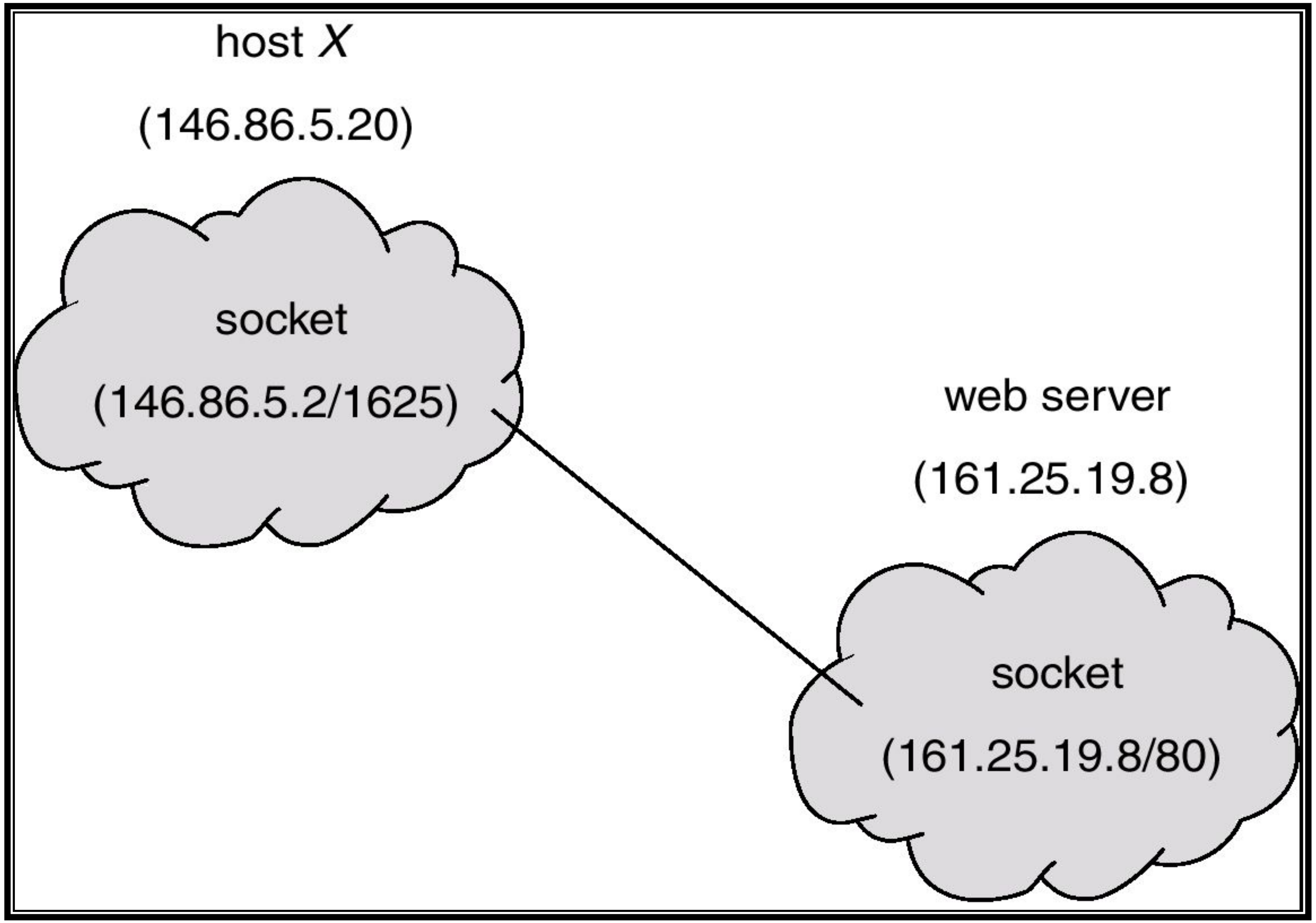
Strategies for communication in client–server systems

- Sockets
- Remote Procedure Calls
- Pipes

# Sockets

- A socket is defined as an endpoint for communication.
- A pair of processes communicating over a network employees **a pair of sockets**– one for each process.
- Socket: ***Concatenation of IP address and port***
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Servers implementing specific services listen to well-known ports
  - telnet server listens to port 23
  - ftp server listens to port 21
  - http server listens to port 80.
- The ports less than 1024 are used for standard services.
- The port for socket is an arbitrary number greater than 1024.
- Communication happens between a pair of sockets.

# Sockets


host X
(146.86.5.20)
socket
(146.86.5.2/1625)
web server
(161.25.19.8)
socket
(161.25.19.8/80)

# Socket: Example Code

- Server

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

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);
    }
  }
}
```

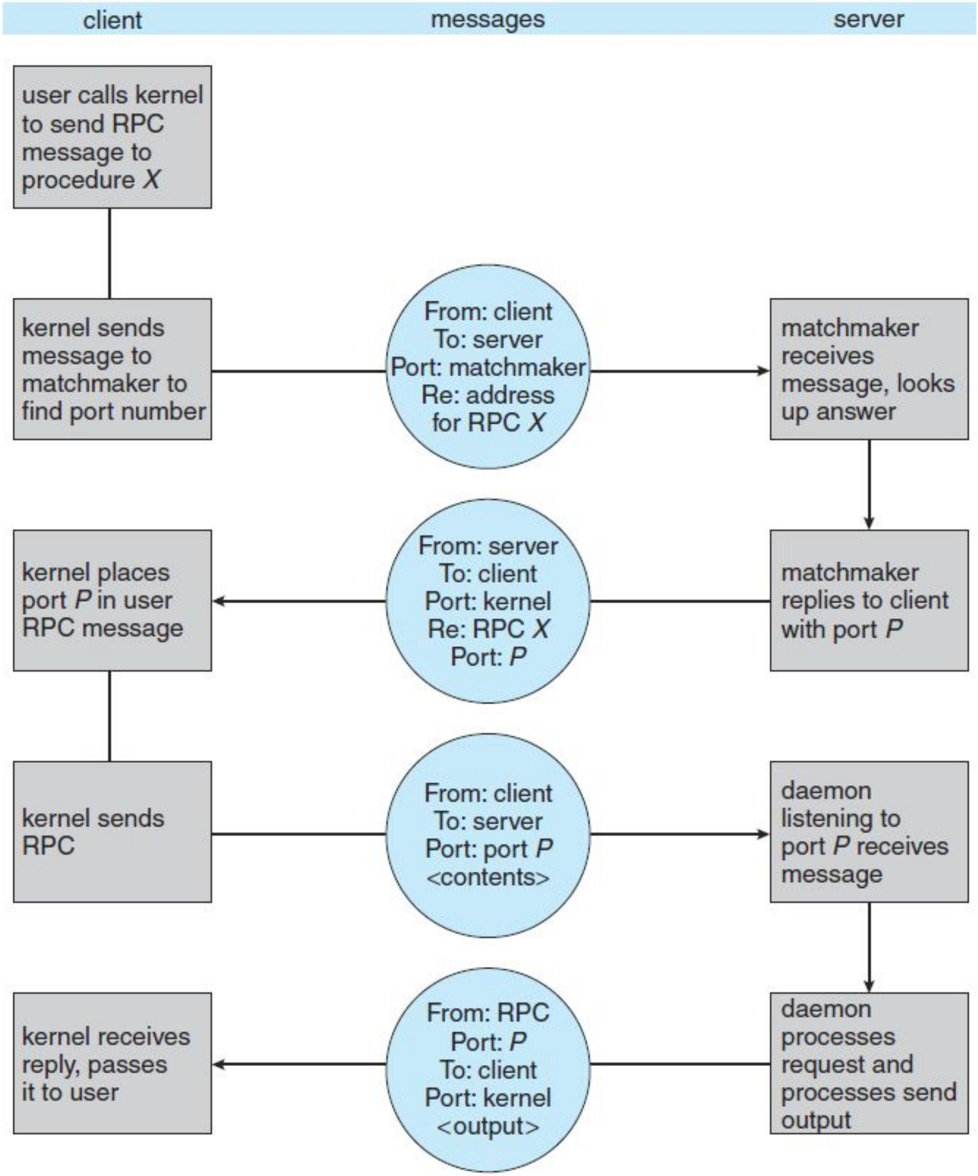
- Client

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

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.
- The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshals* the parameters.
    - Marshalling: Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
- The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.

# Execution of RPC



# Remote Procedure Calls

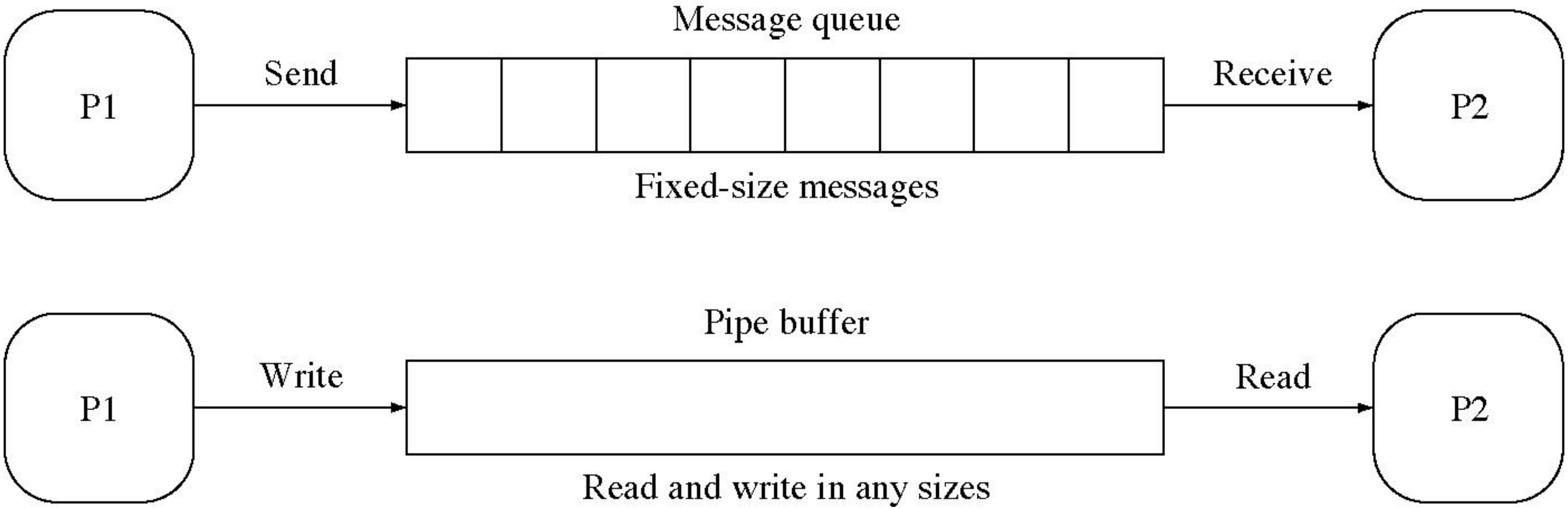
- There are several issues
  - Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors.
  - Solution
    - In RPC, the server must implement “at most once” semantics send the ack  to client to ensure no further communication.
      - The server detects the repeated messages.
      - Even client sends more messages but those will be ignored.
  - Binding issues:  linking, loading and execution
    - Fixed port addresses or rendezvous
- Applications: distributed file systems

# Pipes

- Pipe: one of the first IPC mechanisms in early UNIX systems
  - uses the familiar file interface
  - not a special interface (like messages)

- Connects an open file of one process to an open file of another process
  - Often used to connect the standard output of one process to the standard input of another process

# Messages v/s Pipes



# More about Pipes

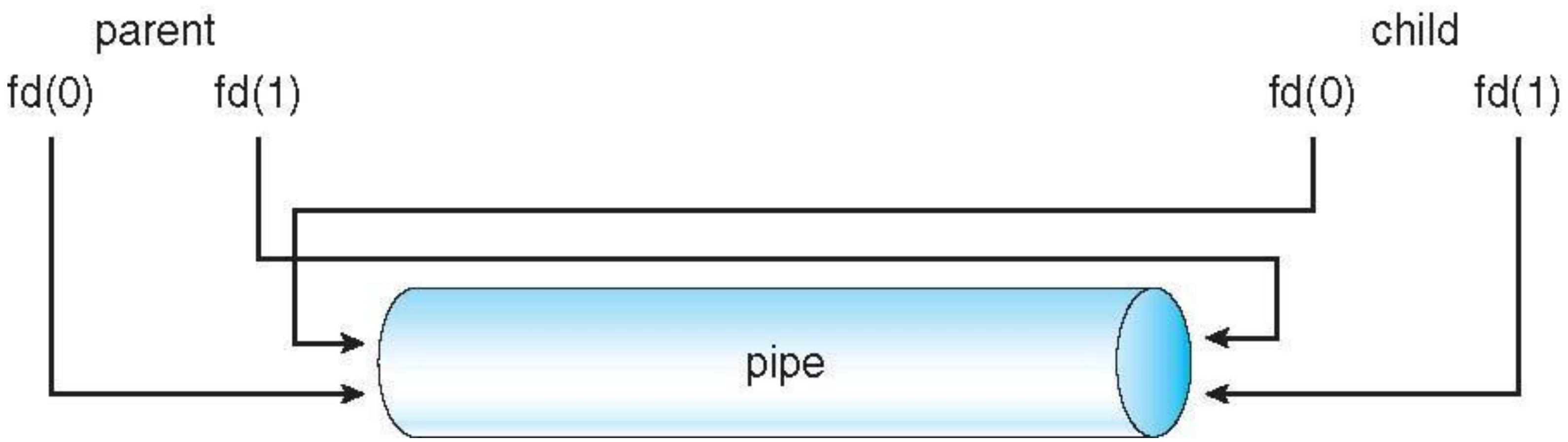
- Acts as a conduit allowing two processes to communicate

- **Issues**
    - Is communication unidirectional or bidirectional?
    - In the case of two-way communication, is it half-duplex (data can travel only one way at a time) or full-duplex?
    - Must there exist a relationship (i.e. parent-child) between the communicating processes?
    - Can the pipes be used over a network?

# Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
  - A parent process creates a pipe and uses it to communicate with a child process that it creates via fork().
- Ordinary pipe can not be accessed from outside the process that creates it.
- Once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

# Ordinary Pipes




- Unix function to create pipes: pipe(int  fd[])
- fd[0] is the read end. fd[1] is write end.

# Example Code

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
 /* create the pipe */
 if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
 }

 /* fork a child process */
 pid = fork();

 if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
 }
```

```
   if (pid > 0) { /* parent process */
     /* close the unused end of the pipe */
     close(fd[READ_END]);

     /* write to the pipe */
     write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

     /* close the write end of the pipe */
     close(fd[WRITE_END]);
   }
   else { /* child process */
     /* close the unused end of the pipe */
     close(fd[WRITE_END]);

     /* read from the pipe */
     read(fd[READ_END], read_msg, BUFFER_SIZE);
     printf("read %s",read_msg);

     /* close the write end of the pipe */
     close(fd[READ_END]);
   }

   return 0;
}
```

# Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bi-directional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
  - Typically, a named pipe has several writers
- Named pipes continue to exist after communicating processes have finished.
- Named pipes are referred to as FIFOs in UNIX systems
  - Only half-duplex transmission is permitted.
  - The communicating processes must reside on the same machine, else one should use sockets.

# Reference

- **Beej's Guide to Unix IPC:** http://beej.us/guide/bgipc/output/html/multipage/index.html

# THANK YOU