LINUX PROGRAMMING

Inter-Process Communication (IPC)

www.cce.hcmut.edu.vn

14/10/2022

TS. Nguyễn Lê Duy Lai

Outline

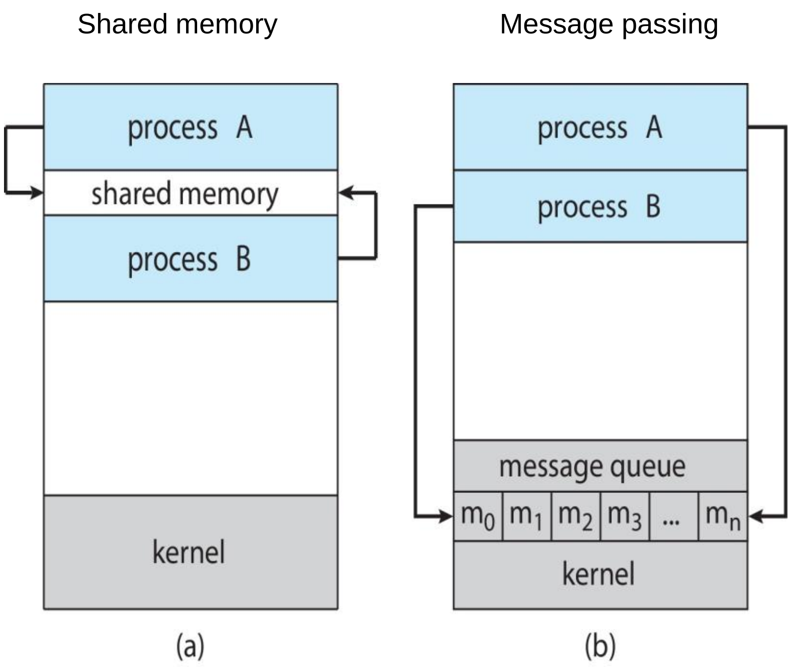
- Introduction to Inter-Process Communication
  - Inter-Process Communication (IPC) models
    - IPC in Shared-Memory Systems
      - Mapped Memory
      - Pipes
    - IPC in Message-Passing Systems
  - Examples of IPC Systems
  - Communication in Client-Server Systems
    - Sockets
- Labs

Inter-Process Communication (IPC)

- Processes within a system may be *independent* or *cooperating*
  - *Independent process* does not share data with any other processes executing in the system
  - *Cooperating process* can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Cooperating processes need *Inter-Process Communication* (**IPC**)

Communication Models

- Types of IPC differ by the following criteria
  - Related/unrelated processes
  - Read/write operations
  - Number of processes permitted
  - Synchronization
- Two models of IPC
  - *Shared Memory*
  - *Message Passing*



Shared Memory

Inter-Process Communication – Shared Memory

- Shared memory: an *area of memory* shared among the processes that wish to communicate
  - allowing two or more processes to access the same memory as if they all called `malloc()` and were returned *pointers* to the same actual memory.
  - When one process changes the memory, all the other processes see the modification
- The communication is *under the control of the user processes*, not the operating system (kernel).
- Major issue is to provide a mechanism that will allow the user processes to *synchronize* their actions when they access shared memory (e.g., avoid race conditions).

Synchronization: Producer-Consumer Problem

- Paradigm for cooperating processes: Producer-Consumer problem
  - *Producer process* produces information that will be consumed by a *consumer process*
- Buffer
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

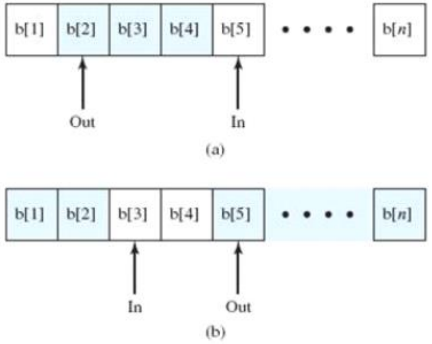
Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

Bounded Buffer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



- Solution is correct, but can only use `BUFFER_SIZE-1` elements

Producer Process – Shared Memory

```
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;
}
```

Consumer Process – Shared Memory

```
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 */
}
```

Inter-Process Communication - Mapped Memory

- Mapped memory permits different processes to communicate *via a shared file* (or as an easy way to access the contents of a file).
- Mapped memory forms an *association* between a file and a process's memory.
  - Linux splits the file into *page-sized chunks* and then copies them into *virtual memory pages* available in a process's address space.
- Mapped memory
  - allocating a buffer to hold a file's entire contents,
  - reading the file into the buffer,
  - writing the buffer back out to the file.

Mapped Memory - Mapping an Ordinary File

- To map an ordinary file to a process's memory, use the `mmap` call.
    - first argument is the address,
    - second argument is the length of the map in bytes,
    - third argument specifies the protection on the mapped address range,
    - fourth argument is a flag value,
    - fifth argument is a file descriptor,
    - last argument is the offset from the beginning of the file (where starting the map).
  - If the call succeeds, it returns a pointer to the beginning of the memory.
- To finish with a memory mapping, release it by using `munmap` call.

Mapped Memory - Shared Access to a File

- Different processes can communicate *using memory-mapped regions* associated with the same file.
- Specify the `MAP_SHARED` flag so that any writes to these regions are immediately transferred to the underlying file and made visible to other processes.
- Alternatively, you can force Linux to *incorporate buffered writes* into the disk file by calling `msync`.
- Users of memory-mapped regions must establish and follow a protocol to *avoid race conditions*.

Message Passing

Inter-Process Communication – Message Passing

- 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)
  - receive(message)
- The message size is either *fixed* or *variable*

Message Passing (Cont.)

- If processes **P** and **Q** wish to communicate, they need to:
  - establish a *communication link* between them
  - *exchange messages* via send/receive primitives
- Implementation issues:
  - 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?

Message Passing (Cont.)

- Implementation of the communication link
  - *Physical*
    - Shared memory
    - Hardware bus
    - Network equipment
  - *Logical*
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering
    - Network protocols

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 the 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 the 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 (Cont.)

- Operations
  - *create* a new mailbox (or port)
  - *send* and *receive* messages through the 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 (Cont.)

- *Mailbox sharing*
  - Example
    - **P1**, **P2**, and **P3** share mailbox **A**,
    - **P1** sends; **P2** and **P3** 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

Message Passing – Synchronization

- Message passing may be either *blocking* or *non-blocking*
- **Blocking** is considered as synchronous
  - *Blocking send* – the sender is blocked until the message is received
  - *Blocking receive* – the receiver is blocked until a message is available
- **Non-blocking** is considered as asynchronous
  - *Non-blocking send* – the sender sends the message and continue
  - *Non-blocking receive* – the receiver receives: a valid message, or Null message
- Different combinations possible
  - If both send and receive are blocking, we have a *rendezvous mechanism*

Producer – Message Passing

```
message next_produced;

while (true) {
		/* produce an item in next_produced */

	send(next_produced);
}
```

Consumer – Message Passing

```
message next_consumed;

while (true) {
        receive(next_consumed)

        /* consume the item in next_consumed */
}
```

Buffering

- *Queue of messages* attached to the link.
- Implemented in one of three ways
  - *Zero capacity* – no messages are queued on a link
    - Sender must wait for the receiver (rendezvous)
  - *Bounded capacity* – finite length of n messages
    - Sender must wait if link is full
  - *Unbounded capacity* – infinite length
    - Sender never waits

Examples of IPC Systems - POSIX

- *POSIX Shared Memory*
  - Process first creates shared memory segment
    - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object
    - `ftruncate(shm_fd, 4096);`
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory are done by using the pointer returned by `mmap()`.

IPC POSIX Producer – Consumer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Examples of IPC Systems - Mach

- Mach communication is *message-based*
  - Even system calls are *messages*
  - Each task gets two *ports* at creation – *Task Self port* and *Notify port*
  - Messages are sent and received using the `mach_msg()` function
  - Ports needed for communication, created via `mach_port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Mach Messages

```
#include<mach/mach.h>

struct message {
        mach_msg_header_t header;
        int data;
};

mach port t client;
        mach port t server;
```

Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
   MACH_SEND_MSG, // sending a message
   sizeof(message), // size of message sent
   0, // maximum size of received message - unnecessary
   MACH_PORT_NULL, // name of receive port - unnecessary
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

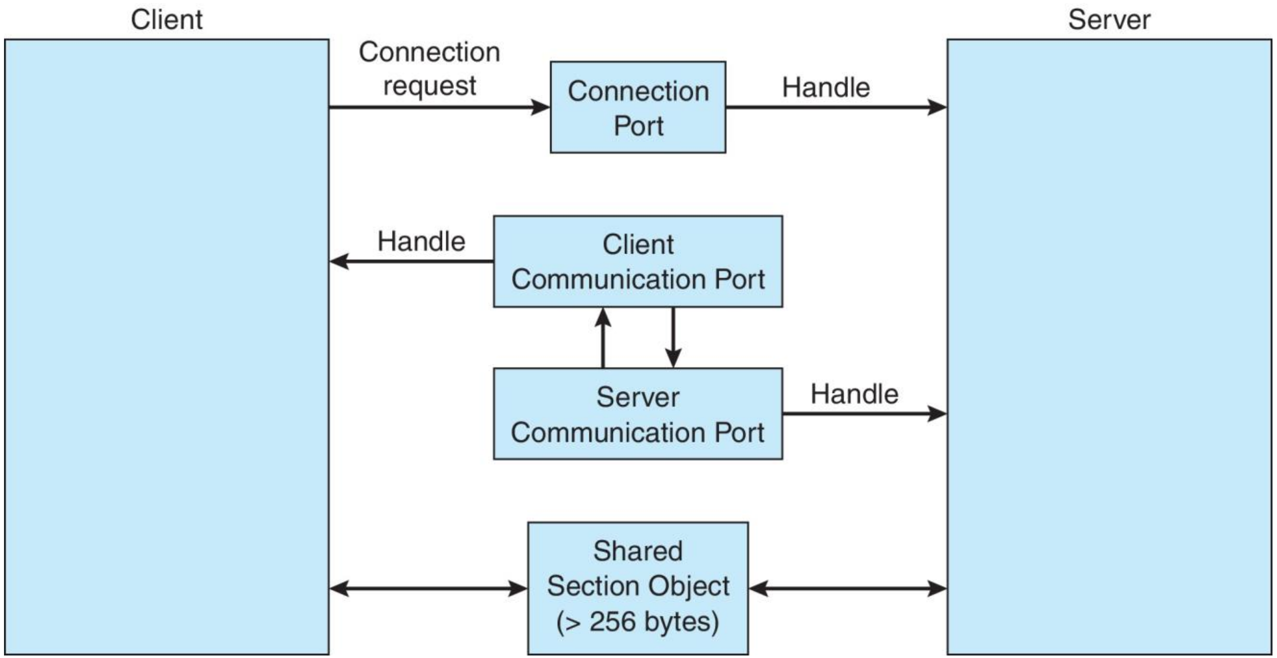
Mach Message Passing - Server

```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
   MACH_RCV_MSG, // sending a message
   0, // size of message sent
   sizeof(message), // maximum size of received message
   server, // name of receive port
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

Examples of IPC Systems – Windows

- Message-passing centric via advanced *Local Procedure Call* (**LPC**) facility
  - Only works between *processes on the same system*
  - Uses *ports* (like mailboxes) to establish and maintain communication channels
  - 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*

Local Procedure Calls in Windows



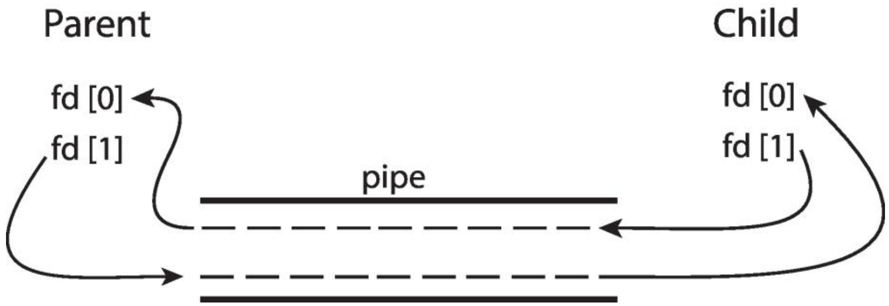
Pipes

Pipes

- Acts as a conduit allowing *2 processes* (or *threads*) to communicate
- Issues:
  - Is communication *unidirectional* or *bidirectional*?
  - In the case of two-way communication, is it *half* or *full-duplex*?
  - Must there exist a *relationship* (e.g., parent-child) between communicating processes?
  - Can the *pipes* be used over a network?
- *Ordinary pipes* – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- *Named pipes* – can be accessed without a parent-child relationship.

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



Named Pipes

- *Named pipes* are more powerful than ordinary pipes
- Communication is *bidirectional*
- *No parent-child relationship* is necessary between the communicating processes
- *Several processes* can use the named pipe for communication
- Provided on both **UNIX** and **Windows** systems

Creating Pipes

- In a shell, the symbol `|` creates a pipe.
- To create a pipe, invoke the `pipe` command.
    - Supply an integer array of size 2.
    - The call to `pipe` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.
    - Data written to the file descriptor `read_fd` can be read back from `write_fd`.
- Communication Between Parent and Child Processes
    - A call to `pipe` creates file descriptors, which are valid only within that process and its children.
    - When the process calls `fork`, file descriptors are copied to the new child process.
    - Pipes can connect only related processes.

Ordinary Pipes – Redirecting the Standard Input, Output, and Error Streams

- Frequently, the need is to create a child process and set up one end of a pipe as its *standard input* or *standard output*.
  - Using the `dup2` call, you can equate one file descriptor with another.
- A common use of pipes is to send data to or receive data from a *program* being run in a *subprocess*.
  - The `popen` and `pclose` functions ease this paradigm by eliminating the need to invoke `pipe`, `fork`, `dup2`, `exec`, and `fdopen`.

Named Pipe – First-In, First-Out (FIFO)

- A FIFO file is a pipe that has a name in the filesystem.
  - FIFOs are also called *named pipes*.
  - Processes on either end of the pipe need not be related to each other.
- *Creating a FIFO*: programmatically, using the `mkfifo` command.
- *Accessing a FIFO*: like an ordinary file.
  - To communicate through a FIFO, one program must open it for *writing*, and another program must open it for *reading*.
- Differences from *Windows Named Pipes*
  - Main differences: named pipes for Win32 function more like *sockets*.
  - *Win32 named pipes* can connect processes on separate computers connected *via a network*.

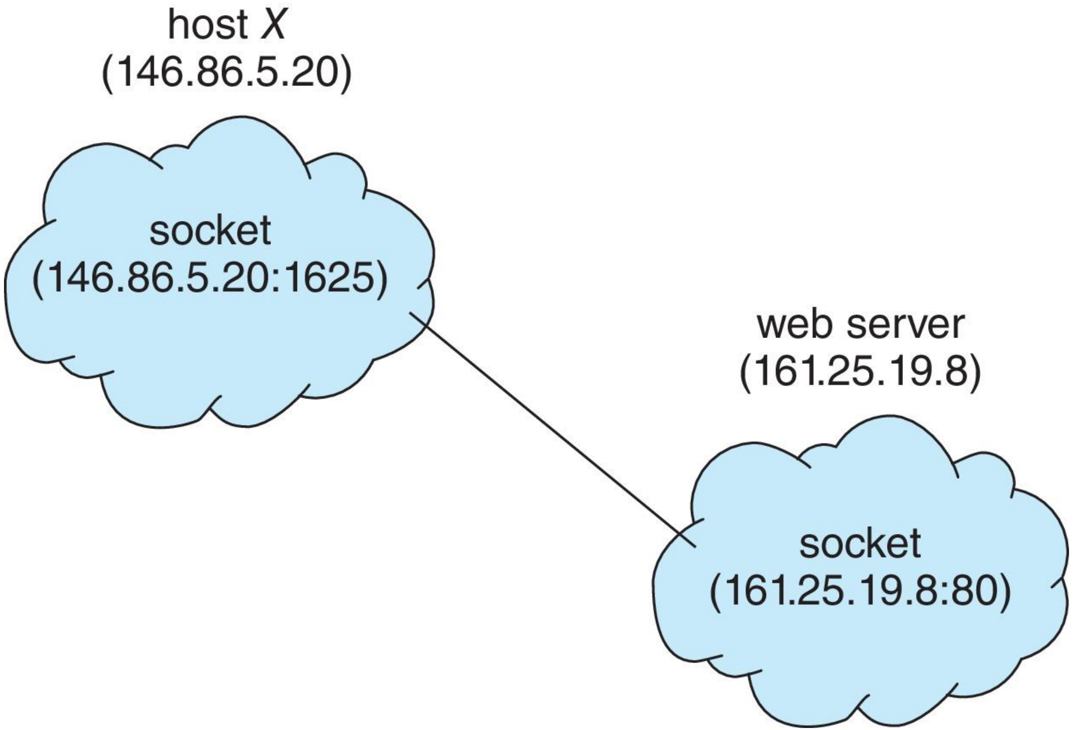
Communications in Client-Server Systems

- *Sockets*
  - A *socket* is defined as an *endpoint* for communication
  - It is a concatenation of *IP address* and *port* – a number included at start of message packet to differentiate network services on a host
    - E.g., The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
  - *Communication* consists of *a pair of sockets*
  - All ports below 1024 are *well-known*, used for *standard services*
  - Special IP address `127.0.0.1` (*loopback*) to refer to the system on which process is running
- *Remote Procedure Calls* (**RPC**)

Socket Concepts

- Creating a socket must specify three parameters:
  - *Communication style*: how the socket treats transmitted data and specifies the number of communication partners? connection-oriented, connectionless (or datagram),
  - *Namespace*: how socket addresses are written? in the "local namespace" or in the "Internet namespace ",
  - *Protocol*: how data is transmitted? **TCP** or **UDP**.
- System Calls
  - `Socket` — Creates a socket
  - `Closes` — Destroys a socket
  - `Connect` — Creates a connection between two sockets
  - `Bind` — Labels a server socket with an address
  - `Listen` — Configures a socket to accept conditions
  - `Accept` — Accepts a connection and creates a new socket for the connection

Socket Communication



Example app: TCP Server in Python

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                            encode())
    connectionSocket.close()
```

Sockets in Java – Server

- Three types of sockets
  - Connection-oriented (TCP)
  - Connectionless (UDP)
  - MulticastSocket class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

```
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);
      }
   }
}
```

Example app: TCP Client in Python

Python TCPClient

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input(‘Input lowercase sentence:’)
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print (‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000 → `clientSocket = socket(AF_INET, SOCK_STREAM)`

No need to attach server name, port → `modifiedSentence = clientSocket.recv(1024)`

Sockets in Java – Client

- The equivalent "Date" 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 (RPC)

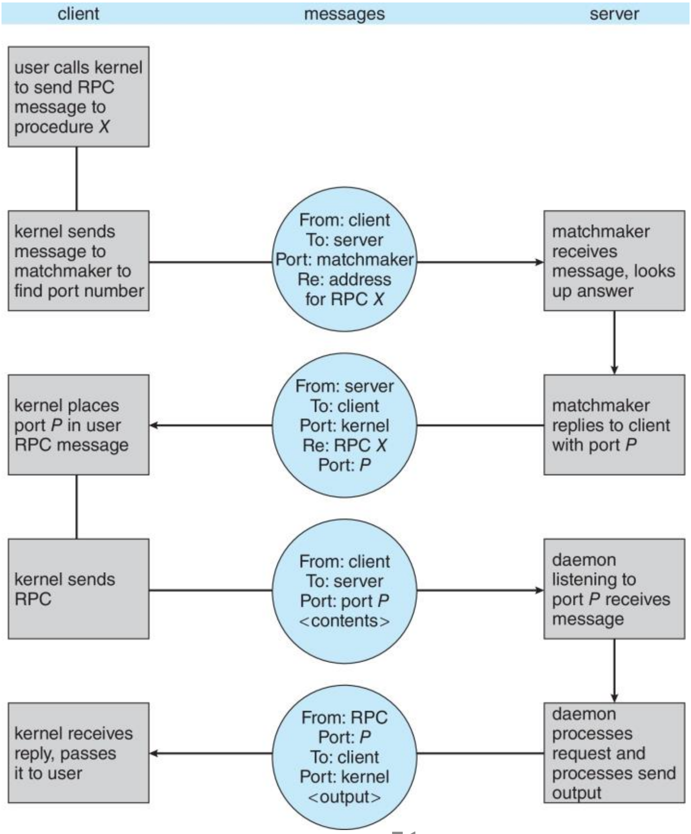
Remote Procedure Calls

- *Remote Procedure Call* (**RPC**) abstracts procedure calls between processes *on networked systems*
  - Again, uses *ports* for service differentiation
- *Stubs* – proxies for the actual procedure on the server and client sides
  - The *client-side stub* locates the server and marshals the parameters
  - The *server-side stub* receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On **Windows**, stub code compiled from specifications written in *Microsoft Interface Definition Language* (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via *External Data Representation* (**XDR**) format to account for different architectures
  - E.g., Big-endian (Motorola) and little-endian (Intel x86)
- *Remote communication* has more failure scenarios than local
  - Messages can be delivered exactly once rather than at most once
- OS typically provides a *rendezvous* (or *matchmaker*) service to connect Client and Server

Execution of RPC



Summary (Cont.)

- When *shared memory* is used for communication between processes, two (or more) processes share the same region of memory. **POSIX** provides an API for shared memory.
- Two processes may communicate by *exchanging messages* with one another using message passing. The **Mach** operating system uses message passing as its primary form of inter-process communication. **Windows** provides a form of message passing as well.

Summary (Cont.)

- A *pipe* provides a conduit for two processes to communicate. There are two forms of pipes, *ordinary* and *named*. Ordinary pipes are designed for communication between processes that have a parent-child relationship. Named pipes are more general and allow several processes to communicate.
- **UNIX** systems provide ordinary pipes through the `pipe()` system call. *Ordinary pipes* have *a read end* and *a write end*. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. *Named pipes* in UNIX are termed FIFOs.

Summary (Cont.)

- **Windows** systems also provide two forms of pipes—anonymous and named pipes. *Anonymous pipes* are similar to UNIX ordinary pipes. They are unidirectional and employ parent-child relationships between the communicating processes. *Named pipes* offer a richer form of inter-process communication than the UNIX counterpart, FIFOs.
- Two common forms of client-server communication are *sockets* and *remote procedure calls* (**RPCs**). Sockets allow two processes on different machines to communicate *over a network*. RPCs (used in Android) abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.

THANK YOU !

Center Of Computer Engineering

www.cce.hcmut.edu.vn