3. Process Concept

[ECE321/ITP302] Operating Systems

Agenda

- Overview
- Process scheduling
- Operations on processes
- <u>**Inter-process communication**</u>
- Example of IPC system
- Communication in client-server systems

Inter-process Communication (IPC)

- Goal of IPC: cooperation
  - Information sharing
    - Shared file, …
  - Computation speedup
    - Multiple CPU or I/O
  - Modularity
    - Dividing system functions
  - Convenience
    - Editing, printing, compiling in parallel

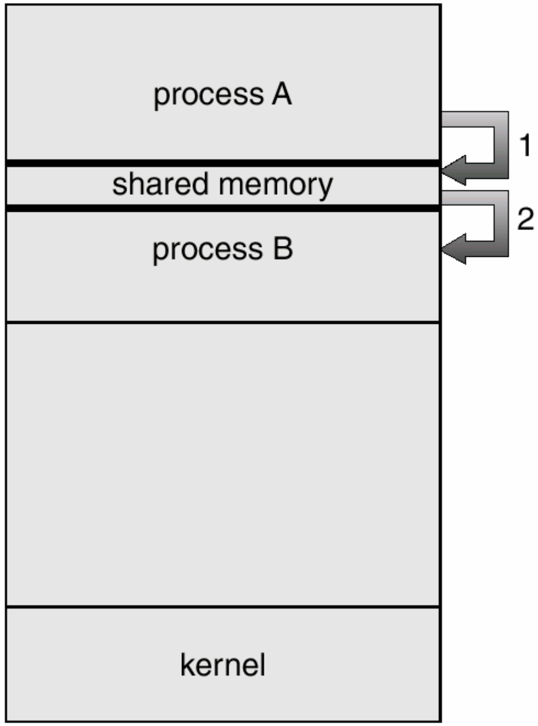
- IPC Models
  - Shared-memory model
  - Message-passing model

Shared-Memory Systems

- Shared-memory segment
  - Special memory space that can be shared by two or more processes
  - Form of data and location is not determined by OS, but those processes
    - Processes should avoid simultaneous writing by themselves
- Advantage
  - Fast

  -> Suitable for large amount of data

Example) producer-consumer problem



Producer–Consumer Problem

- Producer and consumer communicate information (item) through shared memory

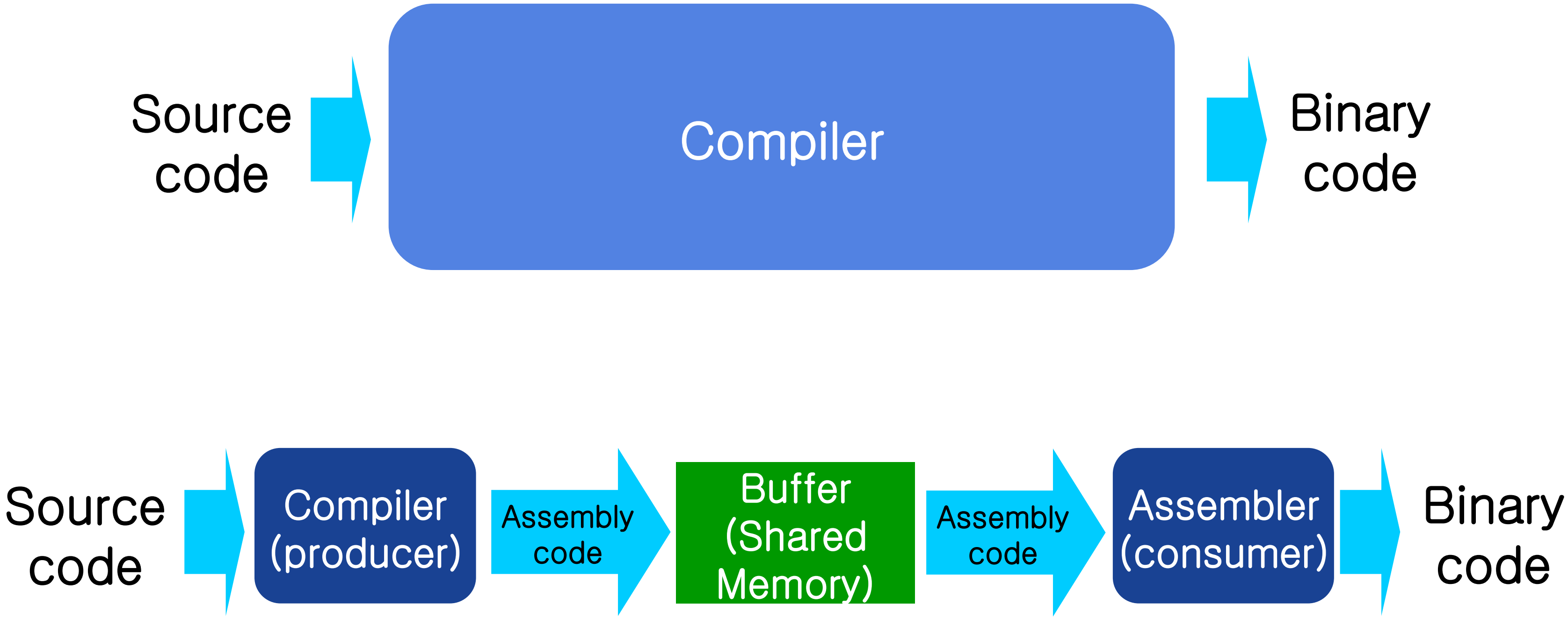
  - Producer: produce information for consumer
  - Consumer: consume information written by producer

  Ex) compiler – assembler, server – client

  Note! Producer and consumer should be synchronized.

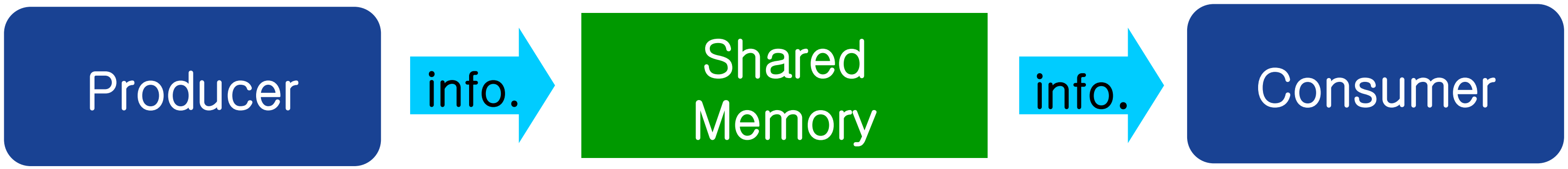
  → Discussed in chapter 6

Producer–Consumer Problem



Producer–Consumer Problem

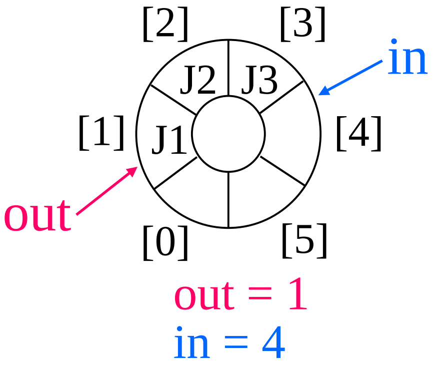
- Two types of buffer
  - Unbounded buffer
    - No practical limit on buffer size
    - Producer can always produce
  - Bounded buffer
    - Producer must wait if buffer is full.



Producer-Consumer Problem using Bounded Buffer

- Representation of buffer
  - Buffer is represented by circular queue

```
#define BUFFER_SIZE 6
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;             // tail or rear
int out = 0;            // head or front
```



  - Empty/full condition
    - in == out: buffer is empty
    - (in+1)%BUFFER_SIZE == out: buffer is full

    Cf. Buffer can store at most BUFFER_SIZE – 1 items

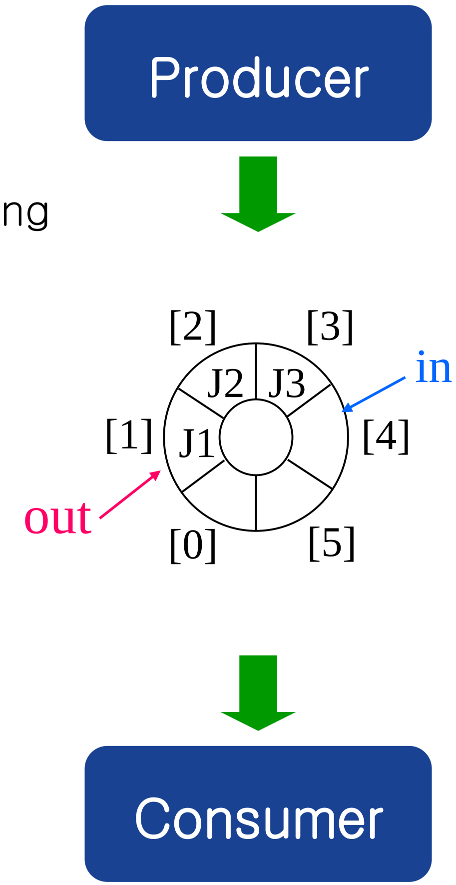
Producer-Consumer Problem using Bounded Buffer

- Producer

```
item nextProduced;

while (1) {
   // produce an item in nextProduced
   while (((in + 1) % BUFFER_SIZE) == out); // do nothing
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer

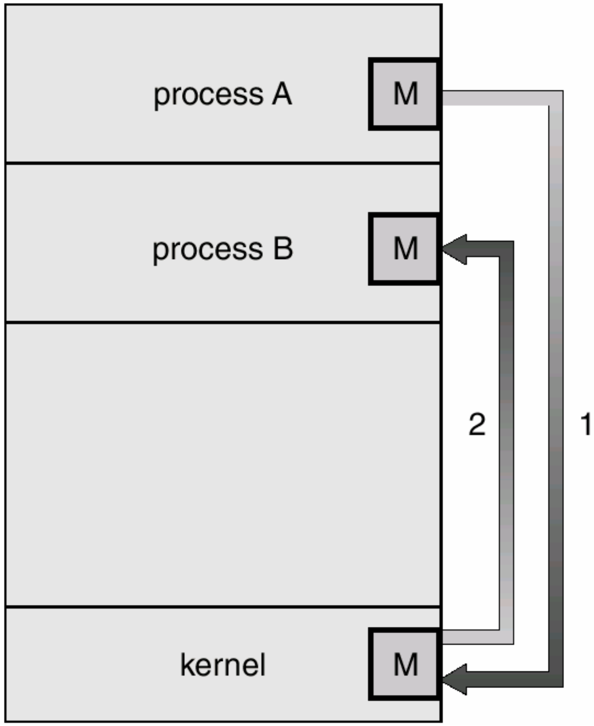
```
item nextConsumed;

while (1) {
   while (in == out); /* do nothing */
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   // consume the item in nextConsumed
}
```



Message-Passing Systems

- Process communication via passage-passing facility provided by OS
- Advantage
  - No conflict

  -> Suitable for smaller amounts of data
  - <u>Communication between processes on different computer</u>



Message-Passing Systems

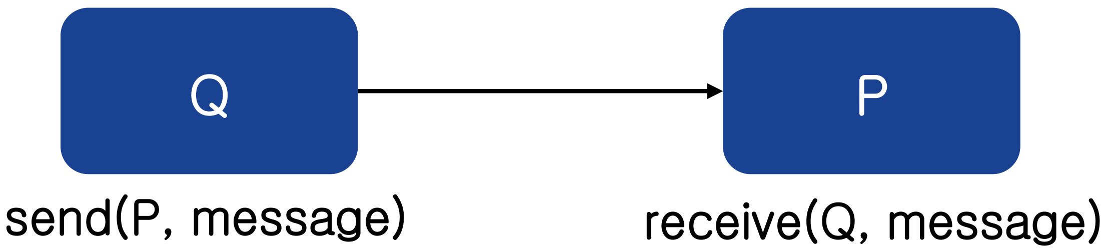
- For message passing, communication link should be exist between the processes

- Essential operations
  - send(message)
  - receive(message)

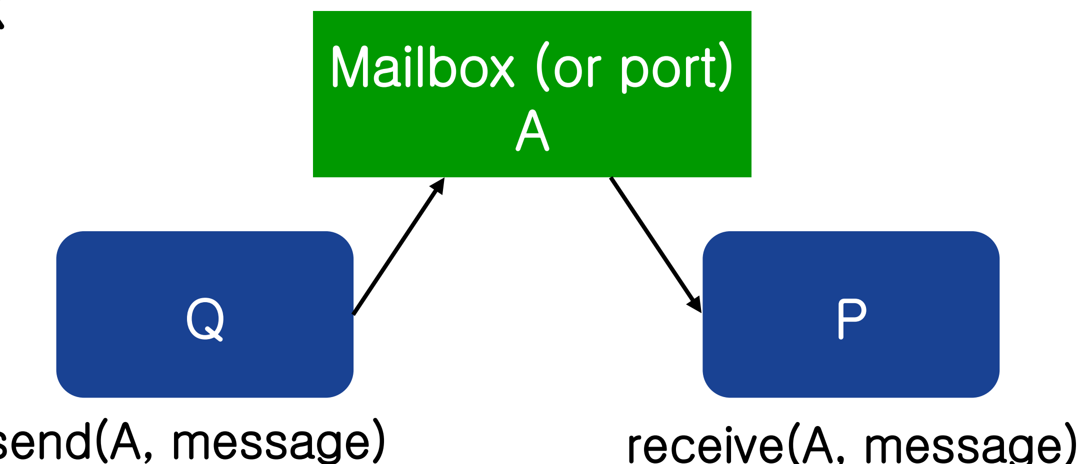
- (Logical) Implementation methods
  - Direct/indirect
  - Synchronous/asynchronous
  - Buffering
    - Zero/bounded/unbounded capacity

Direct/Indirect Communication

- Direct communication: connection link directly connects processes



- Indirect communication: processes are connected via mailbox



Direct Communication

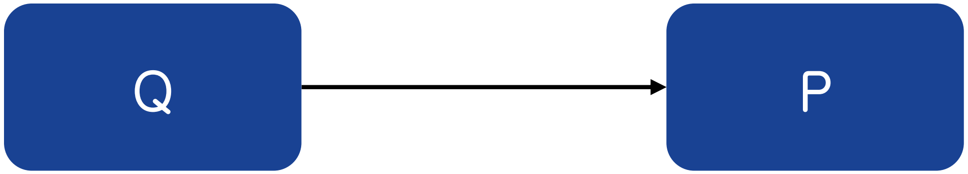
- Processes are connected directly

- Symmetry vs. asymmetry in addressing
  - Symmetric addressing: both sender and receiver know each other
    - Sender: send(P, message)
    - Receiver: receive(Q, message)

  - Asymmetric addressing: only sender knows receiver
    - Sender: send(P, message)
    - Receiver: receive(id, message)
      - id is set to name of sender (output argument)

Direct Communication

- Properties
  - The processes know each other.
  - A link is associated with exactly two processes
  - Between a pair of processes, only one link exists

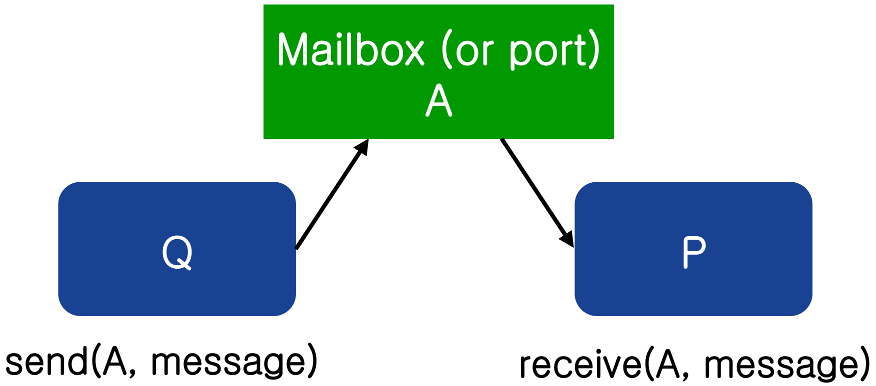


send(P, message)   receive(Q, message)

- Disadvantage: limited modularity
  - Changing identifier of a process requires examining all other process definitions

Indirect Communication

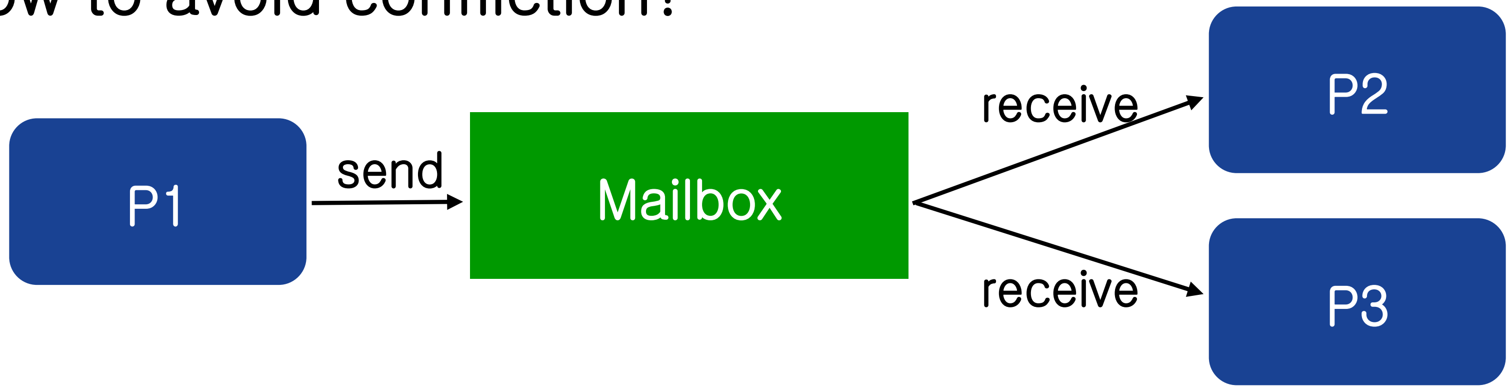
- Processes are connected indirectly through a mailbox or port
  - Sender: send(A, message)
  - Receive: receive(A, message)
- Who is the owner of mailbox?
  - Owned by a process
    - No confusion on receiver
  - Owned by OS
    - Ownership can be transferred



- Properties
  - Processes are connected if they share a common mailbox
  - A link may be associated with more than two processes
  - Between a pair of processes, there may be a number of different links

Indirect Communication

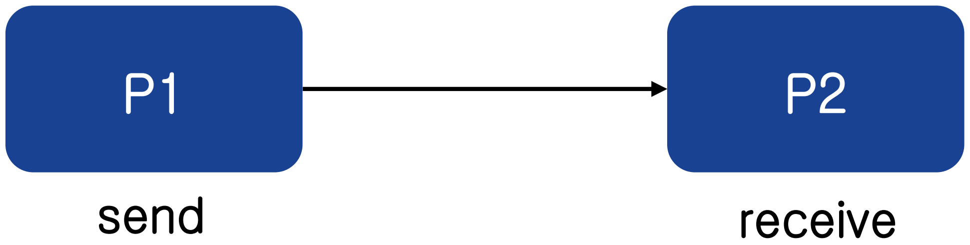
- How to avoid confliction?



- Solutions
  - Allow a link to be associated with two processes at most
  - Allow at most one process at a time to execute receive()
  - Allow the system to select a process to receive arbitrary

Synchronization

- Sender
  - Blocking send: sender is blocked until receiver takes message
  - Non-blocking send: sender just send message and resumes operation
- Receiver
  - Blocking receive: if message is not available, receiver waits
  - Non-blocking receive: if message is not available, receiver is not blocked but receives a null message



Synchronization

- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous
- Producer-consumer becomes trivial

```
message next produced;
while (true) {
    /* produce an item in next produced */
   send(next produced);
}
```

```
message next consumed;
while (true) {
   receive(next consumed);

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

Buffering

- During communication, messages are stored in temporary queue (buffer)

- Three kinds of buffer capacity
  - Zero capacity: only blocking send is possible
  - Bounded capacity: buffer has finite length n
    - If buffer is full, sender must be blocked
    - Otherwise, sender can resume
  - Unbounded capacity: buffer has infinite capacity
    - Sender never blocks

Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- <u>Example of IPC system</u>
- Communication in client-server systems

Examples of IPC Systems

- Shared-memory (POSIX)
- Message-passing (MACH)
- Local Procedure Call (Windows XP)
  - Undocumented internal API (Skip)

Examples of IPC Systems – POSIX

POSIX Shared Memory

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDRW, 0666);
```

- Also used to open an existing segment to share it

- Set the size of the object

```
ftruncate(shm_fd, 4096);
```

- Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

IPC POSIX Producer

```c
#include <stdio.h>
#include <stlib.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_RDRW, 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;
}
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.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;
}
```

[Mach] Message Passing

- Mach: an OS kernel developed at CMU
  - One of the earliest examples of a microkernel
  - Even system calls are made by message
  - Especially designed for multiprocessing, distributed system
  - Execution unit: task (similar to process, but has multiple thread)

- If a task is created, two mailboxes (ports) are also created
  - Kernel mailbox -> communication with task
  - Notify mailbox -> notification of event occurrences

- Message passing system calls
  - msg_send(), msg_receive()
  - msg_rpc() – Remote Procedure Call that can work between systems.

[Mach] Message Passing

- Creating mailbox

  port_allocate(): create new mailbox

- Mailbox of Mach
  - New mailbox is owned by the process that creates it.
  - At a time, only one process can own and receive from a mailbox
  - Basically, FIFO order
    - But order of messages from other processes are not guaranteed
- Mailbox set
  - A set of mailboxes which is treated as single mailbox
  - Each mailbox is assigned to a thread in a task
- Major problem: poor performance
  - Double copying of messages (sender -> mailbox -> receiver)
    - For local communication, a remedy is provided

[Mach] Message Passing

- 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

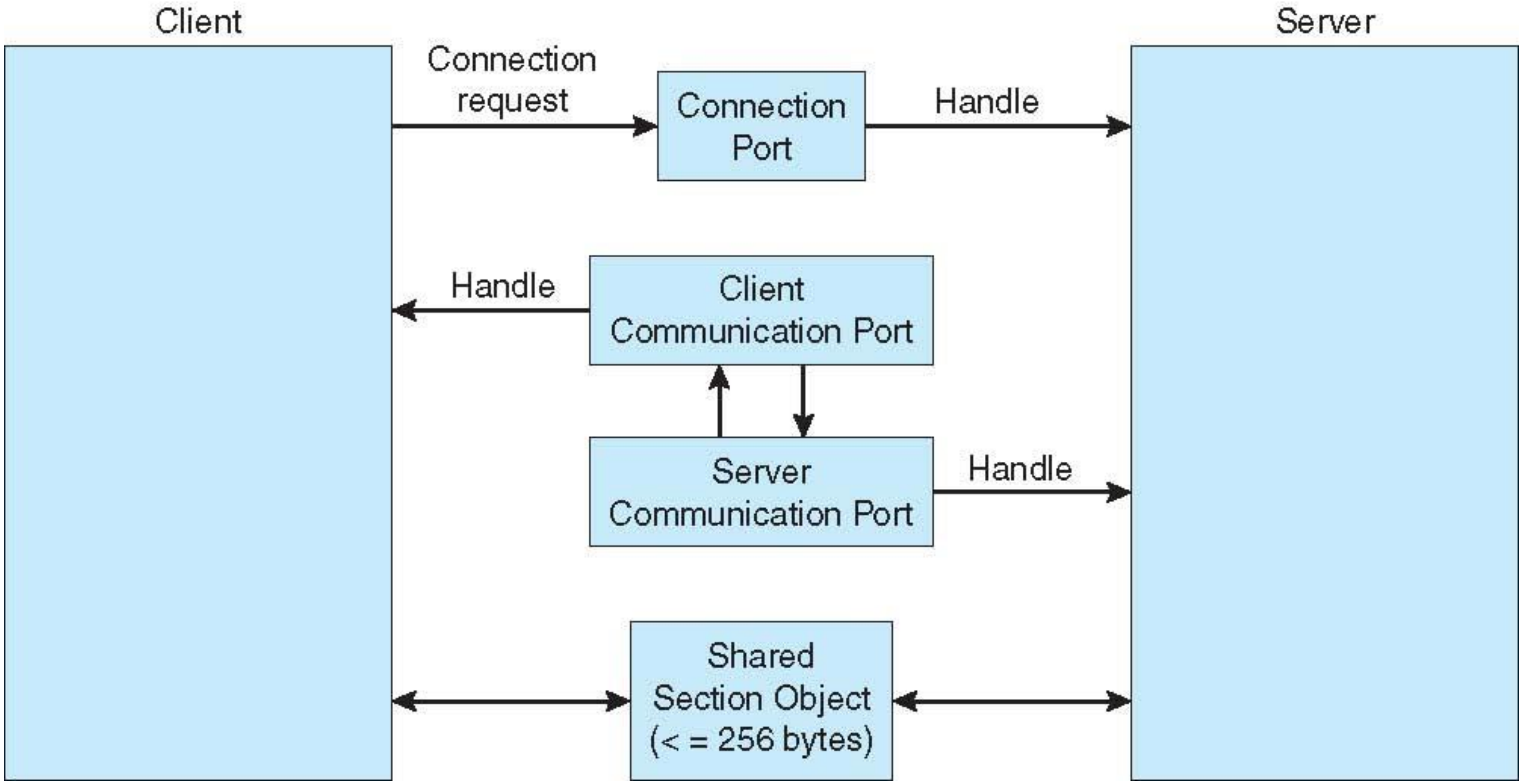
[Windows] Message Passing

- Message-passing centric via (Advanced) local procedure call (LPC) facility
  - Only works between processes on the same system (machine)
  - Similar to RPC, but optimized for WindowsXP
  - 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

[Windows] Message Passing

- Three possible message-passing techniques
  - 1) If message is small (<= 256 bytes), the port's message queue is used as the intermediate storage
  - 2) Other wise, the message is passed through section object which is a region of shared memory.
    (Avoids data copying)
  - 3) Callback mechanism can be used, when client or server cannot respond immediately.
- LPC is not a part of win32
  - LPC facility is not visible to the application programmer
    - Applications invoke standard RPC.
  - If RPC is being invoked on a process on the same system, RPC is indirectly handled through LPC

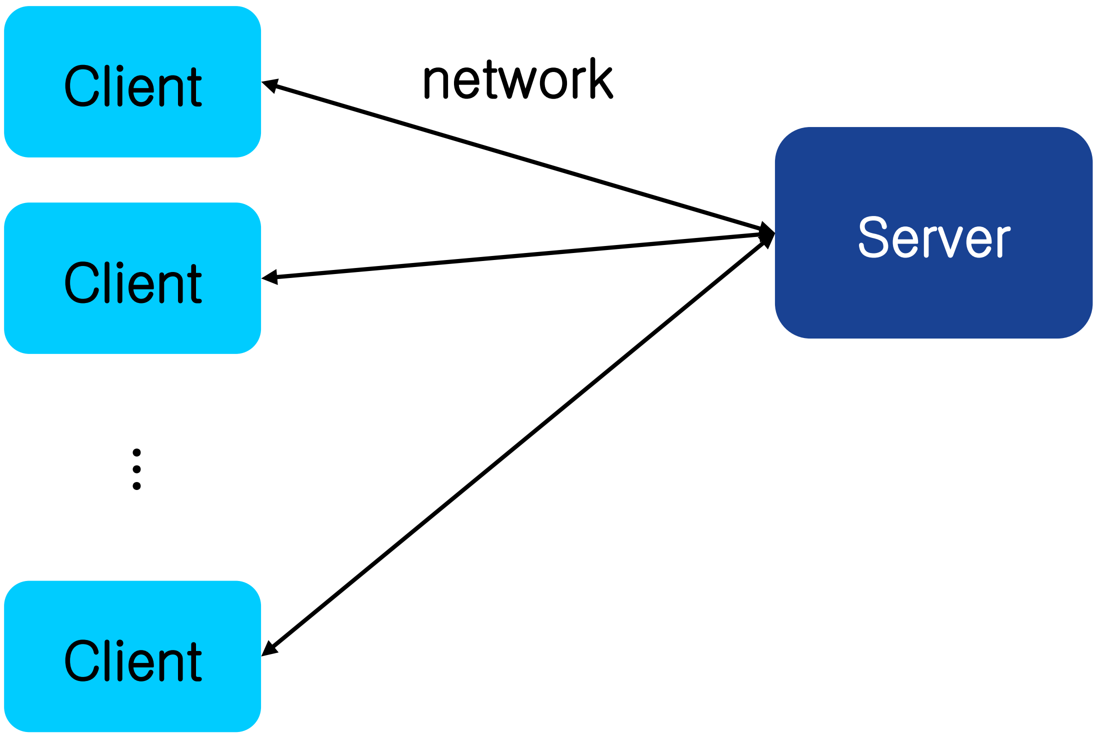
ALPC in windows



Agenda

- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- <u>Communication in client-server systems</u>

Client–Server



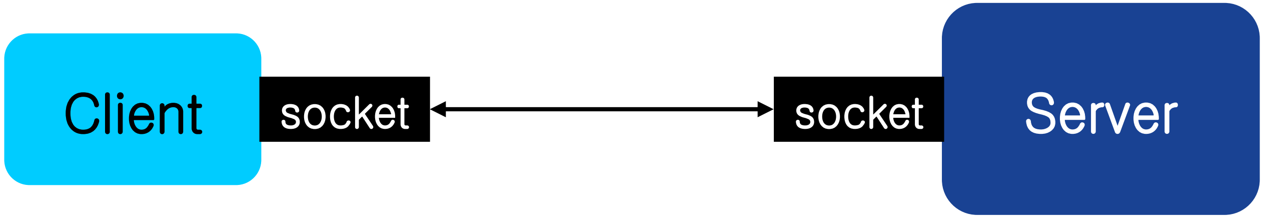
Communications in Client-Server Systems

- Socket
  - Data communication
- RPC (Remote Procedure Call)
  - Procedure call between systems (machines)
    - Procedural programming
- RMI (Remote Method Invocation) of JAVA
  - Invocating method of <u>object</u> in other system
    - Object oriented programming

Socket

- Socket: logical endpoint for communication

  - Identified by <ip address>:<port #>



  - Each connection is identified by a pair of sockets

Socket

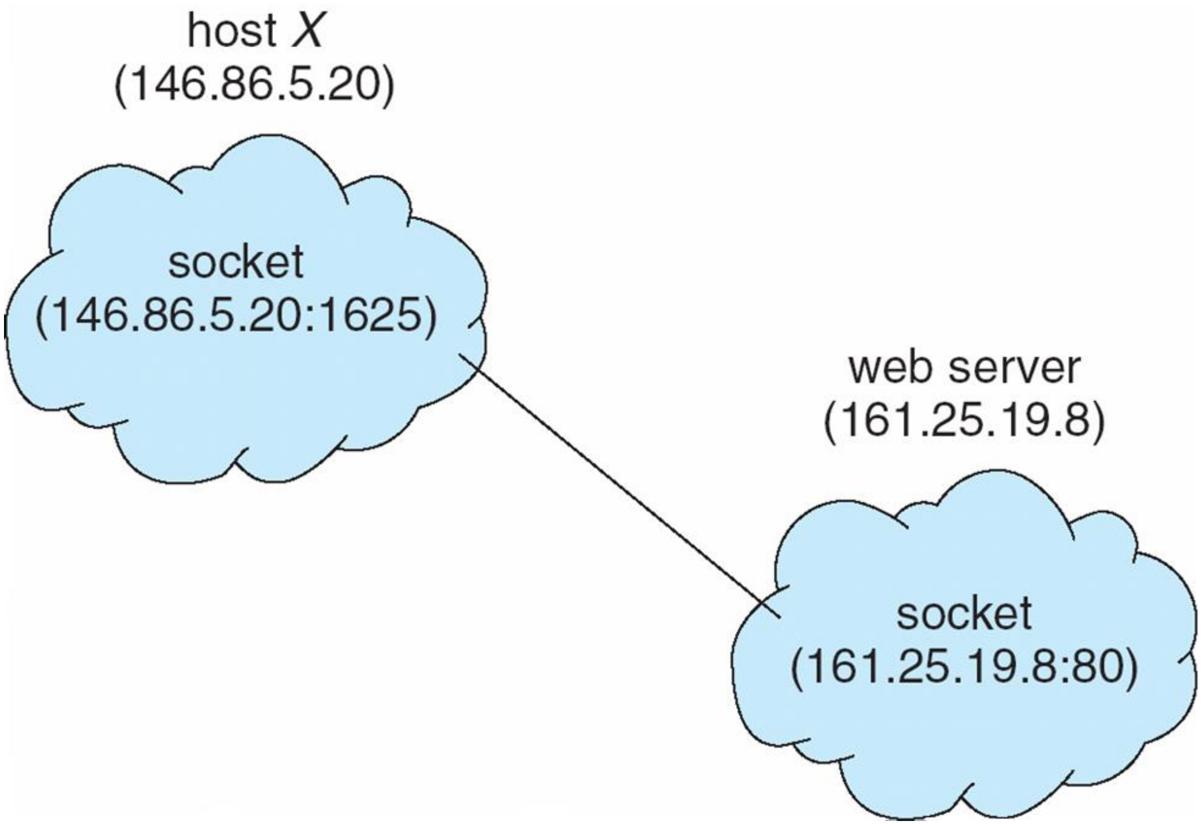
- Port: logical contact point to a computer recognized by TCP and UDP protocols
  - A computer may have multiple ports (0 ~ 65535)

  - Well-known services have their own ports below 1024

    Ex) telnet: 23, ftp: 21, http: 80

    - Server always listens corresponding port
  - Ports above 1024 can be arbitrary assigned for network communication

Socket


host *X*
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

Socket

- Server opens a port to accept connection request
- Initiating connection
  - Client arbitrary assigns a port above 1024

    Ex) a client 146.86.5.20 assigned a port 1625
  - Client request a connection to server

    Ex) a web server 161.25.19.8 (port # of web service: 80)
  - If server accepts request, connect is established

    Ex) <146.86.5.20:1625> – <161.25.19.8:80>

Java Socket

- Socket classes
  - ServerSocket: accepts request for connection
  - Socket: in charge of actual communication

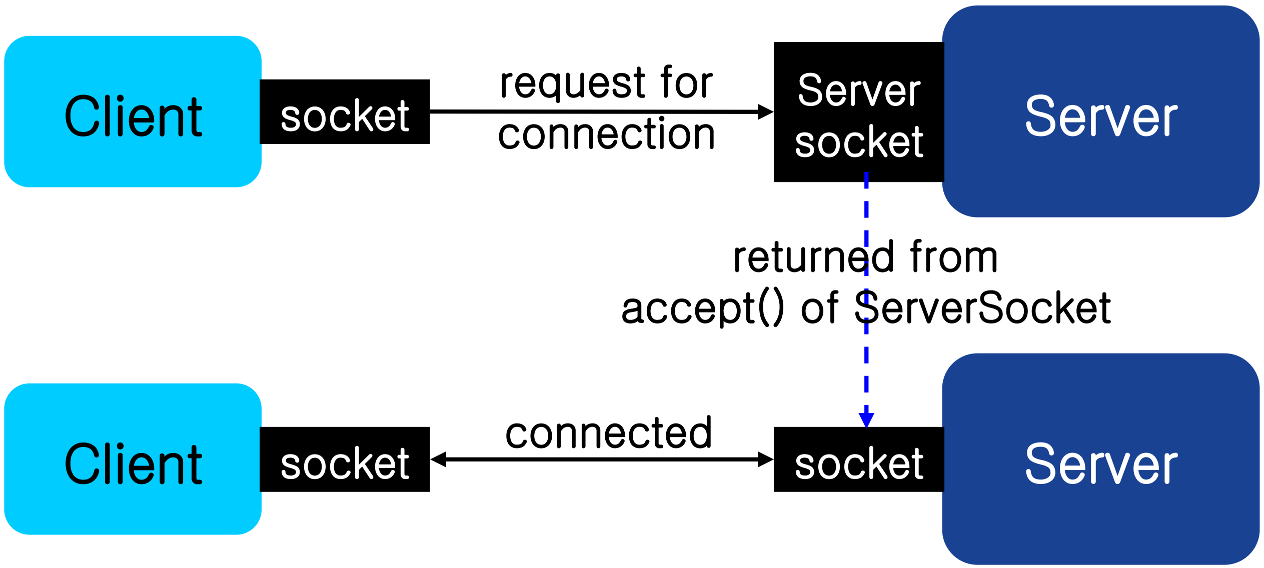


Figure3.21 / 22

```
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();
            // we have a connection

            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 more connections
            client.close();
            }
      }
      catch (IOException ioe) {
            System.err.println(ioe);
      }
}}
```

```
import java.net.*;
import java.io.*;
public class DateClient{
public static void main(String[] args)  {
      try {
            // this could be changed to an IP name or address other than the localhost
            Socket sock = new Socket("127.0.0.1",6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new BufferedReader(new InputStreamReader(in));

            String line;
            while( (line = bin.readLine()) != null)
                        System.out.println(line);
            sock.close();
      }
      catch (IOException ioe) {
            System.err.println(ioe);
      }
}}
```

Java Socket

- Server
  1. Create a ServerSocket

     ServerSocket socket = new ServerSocket(6013);

  2. Wait for a client

     Socket *client* = *socket*.accept();

  4a. If a client is accepted, communicate with client via *client*

- Client
  3. Create a socket to server

     Socket sock = new Socket("127.0.0.1", 6013);

  4b. If connection was established, communicate with server via *sock*

Java Socket

- Server (given *client*)

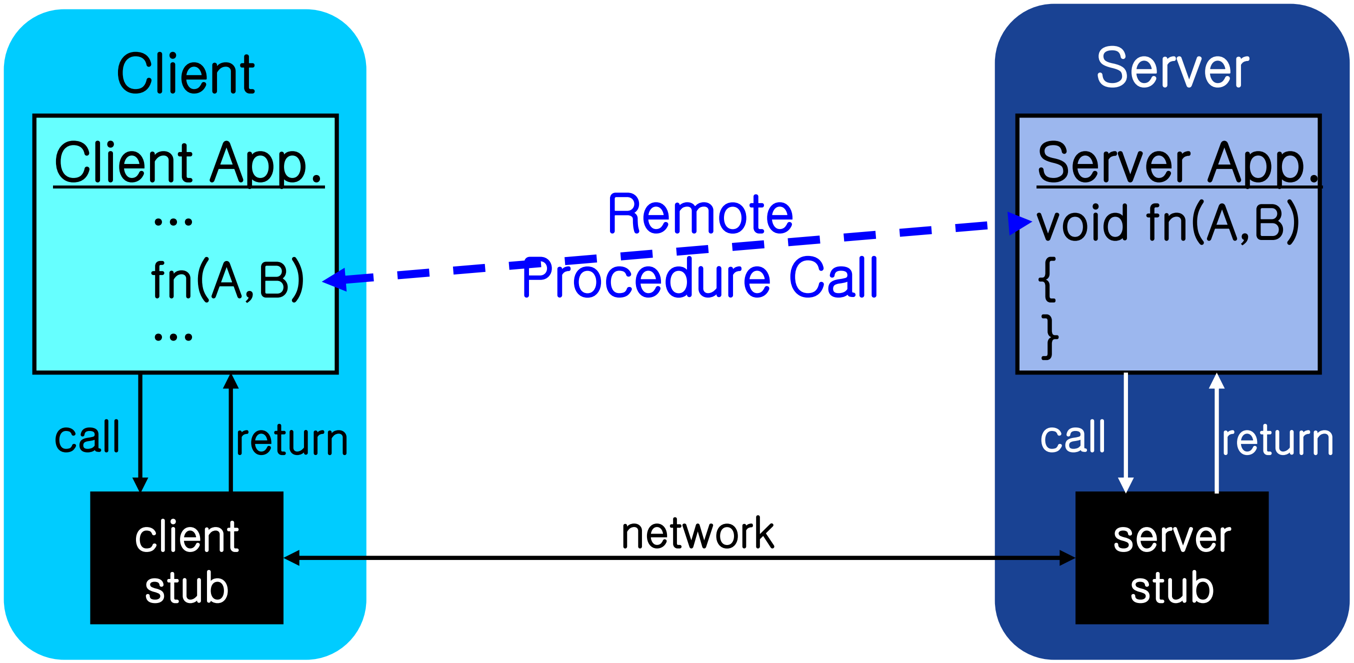
```
PrintWriter pout = new
    PrintWriter(client.getOutputStream(), true);

pout.println(new
    java.util.Date().toString());

client.close();
```

- Client (given *sock*)

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

String line;
while((line = bin.readLine()) !=
    null)
    System.out.println(line);

sock.close();
```

Remote Procedure Calls (RPC)

- RPC: procedure call mechanism between systems
- On server, RPC daemon listens to a port
- Client sends a message containing identifier of function and parameters



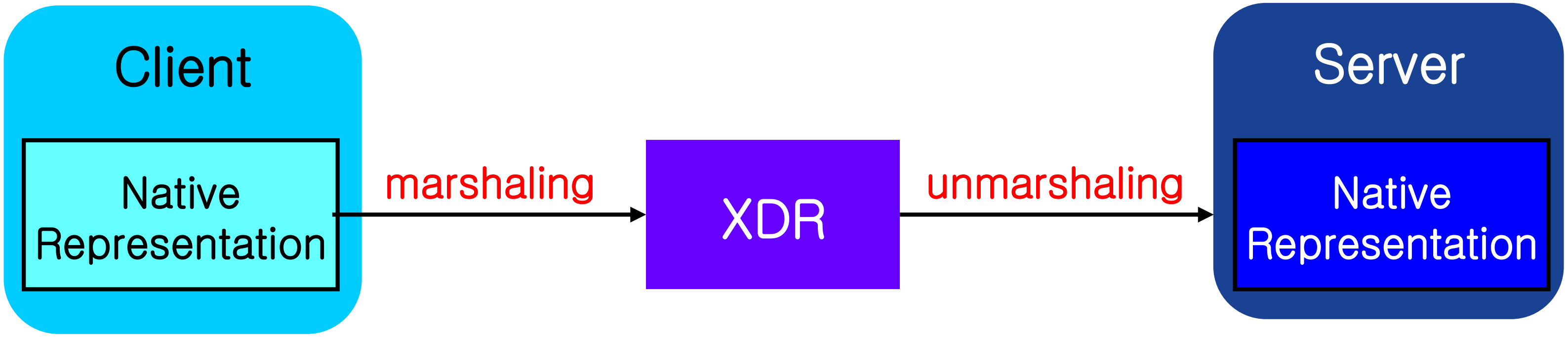
Remote Procedure Calls

- RPC is served through stubs
  - Client invoke remote procedure as it would invoke a local procedure call

- Stub: a small program providing interface to a larger program or service on remote side
  - Client stub / server stub
  - Locate port on server
  - Marshal / unmarshal parameters

Remote Procedure Calls

- Parameter marshaling
  - Motivation: each system has its own data format
    - Ex) representation of integer on a system may different from that on other system
  - -> parameter should be transferred in **standard format**
    - XDR: eXternal Data Representation
  - Marshalling: native representation -> XDR
  - Unmarshalling: XDR -> native representation



Remote Procedure Calls

- Every RPC request should serviced "<u>exactly once</u>"
  - Attach a timestamp to each message
  - Server keeps history of message it has served
    - At every request, it checks the history.
  - Server send ACK message to client
  - Client resend RPC call periodically, until it receives ACK.

Remote Procedure Calls

- Issue: how to bind the client and the server port with no information at start?
- Two approaches for assigning RPC port
  1) Fixed address (hard-coding)

  2) Transferred through rendezvous daemon (matchmaker) illustrated in the figure3.23

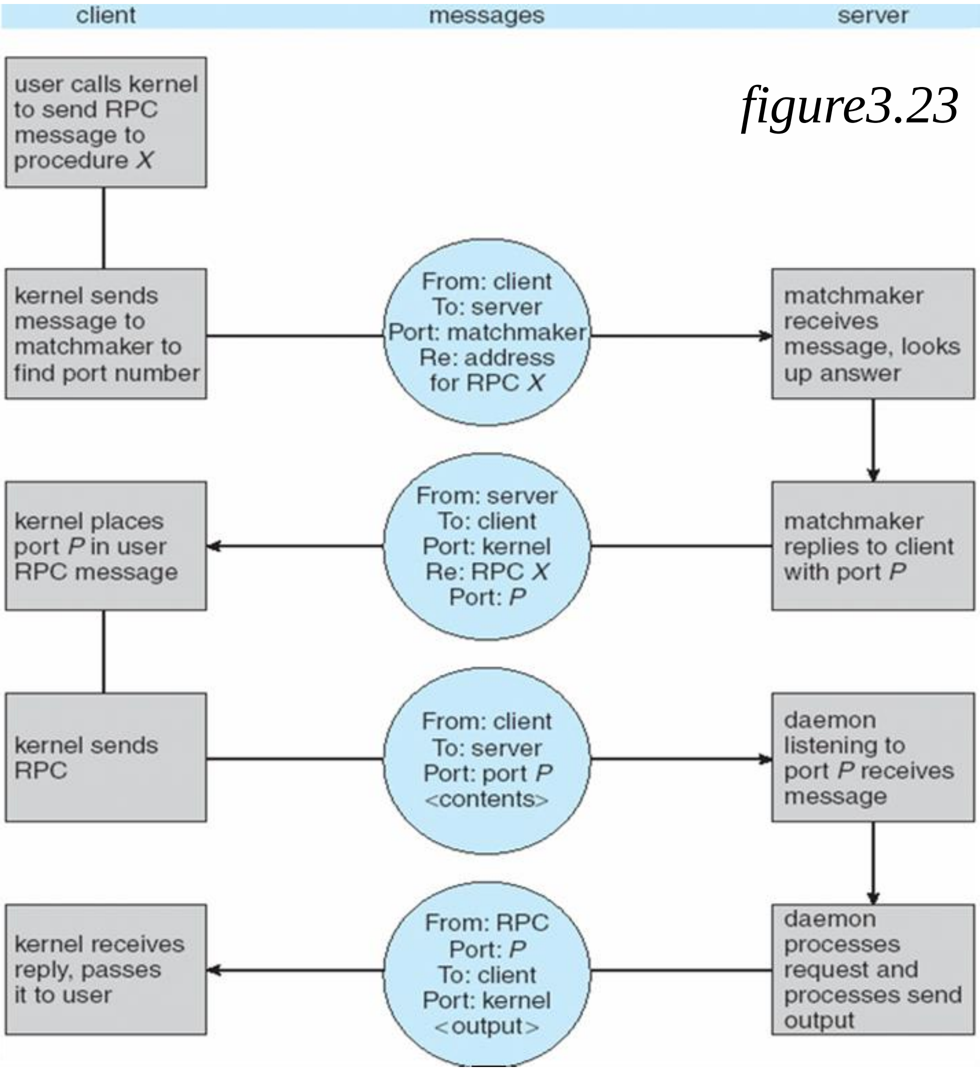


figure3.23

RPC Reference Sites

- Windows
  - MSDN RPC page: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/rpcank.asp
- Unix
  - Document about *rpcgen*.

Others

- 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 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: Unidirectional, parent-child
- Named Pipes: Bidirectional, no parent-child