OPERATING SYSTEMS

WESTERN
WASHINGTON UNIVERSITY

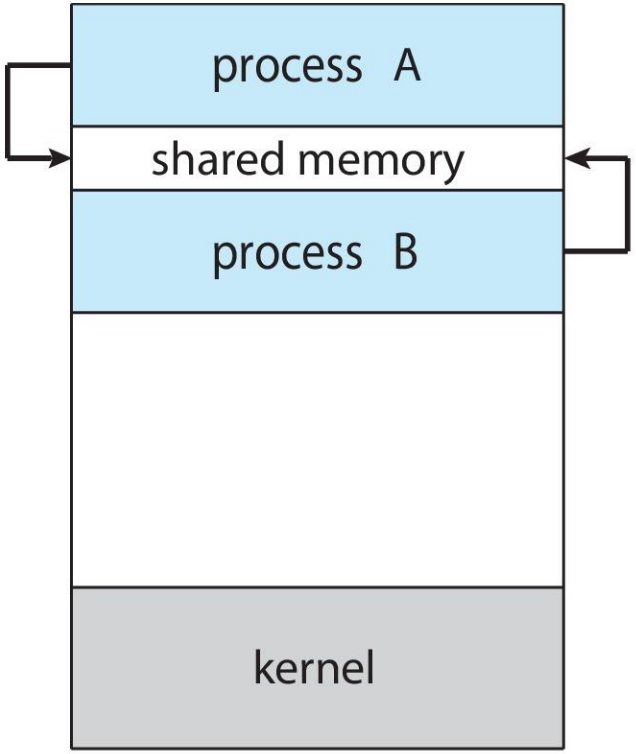
IPC: INTER-PROCESS COMMUNICATION

- Two primary methods:
  - Shared Memory
  - Messaging

SHARED MEMORY VS MESSAGE PASSING

(a) 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.
- Advantage: very fast and efficient

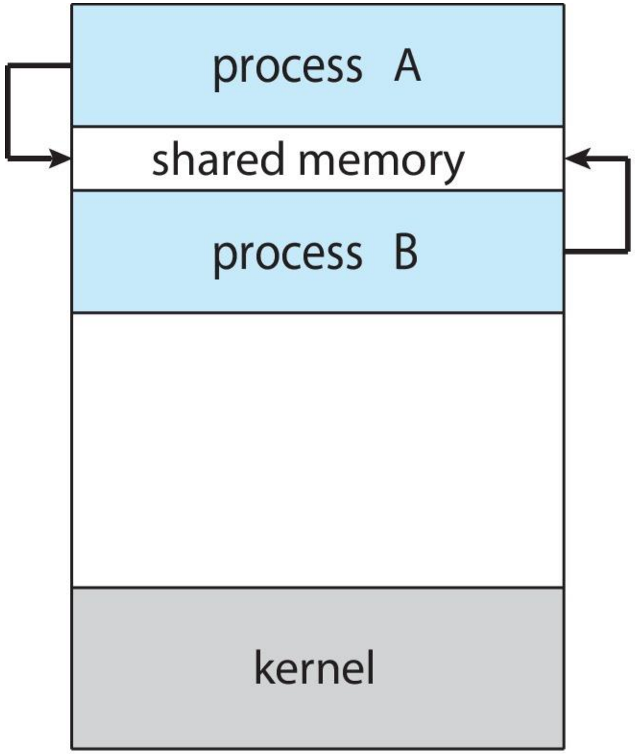


(a)

SHARED MEMORY VS MESSAGE PASSING

(a) 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.
- Advantage: very fast and efficient
- Disadvantage?

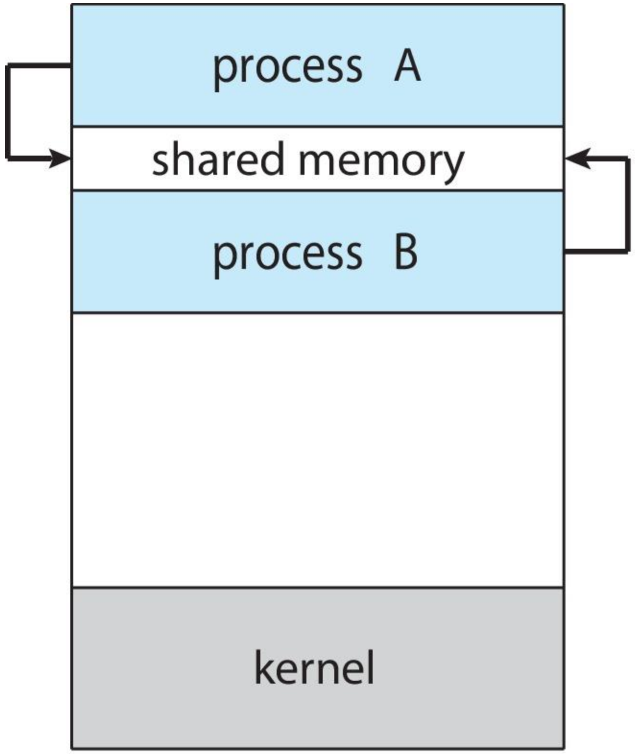


(a)

SHARED MEMORY VS MESSAGE PASSING

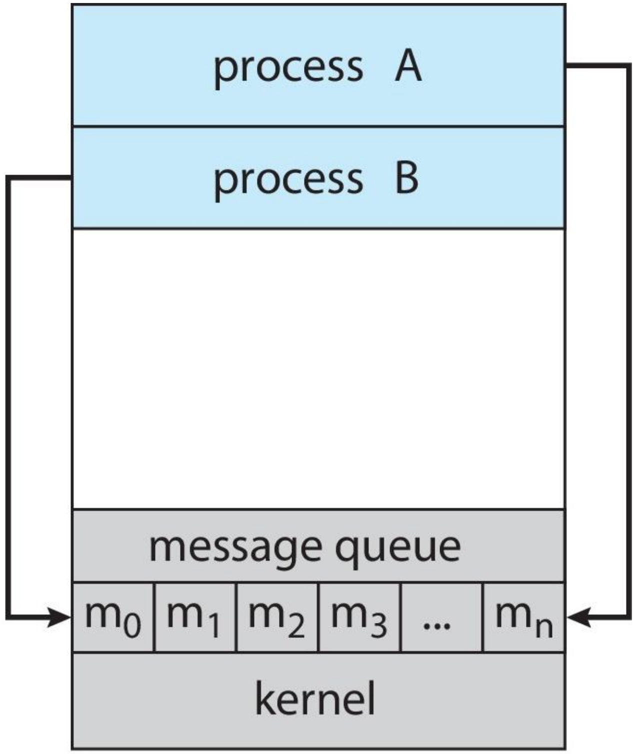
(a) Shared memory

- 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.
- Advantage: very fast and efficient
- Disadvantage?
- OS needs 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 later chapters.



(a)

SHARED MEMORY VS MESSAGE PASSING

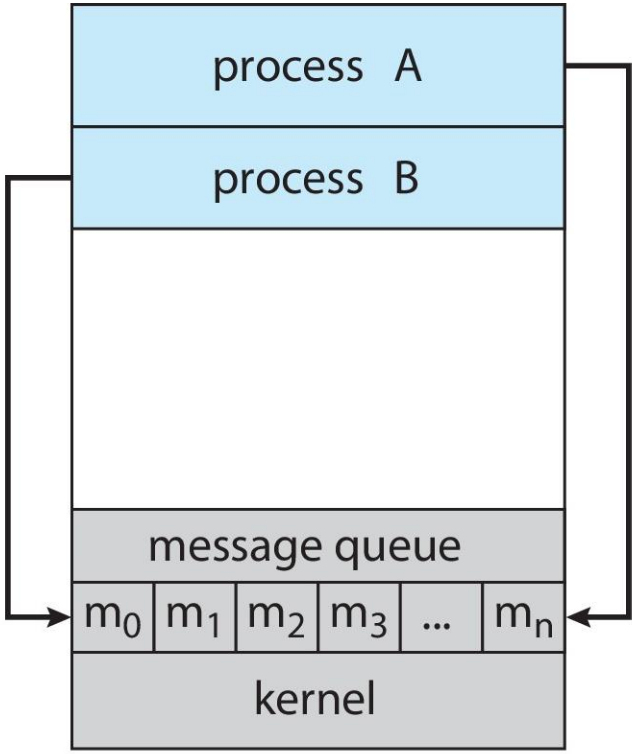


(b)

(b) 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*)

SHARED MEMORY VS MESSAGE PASSING

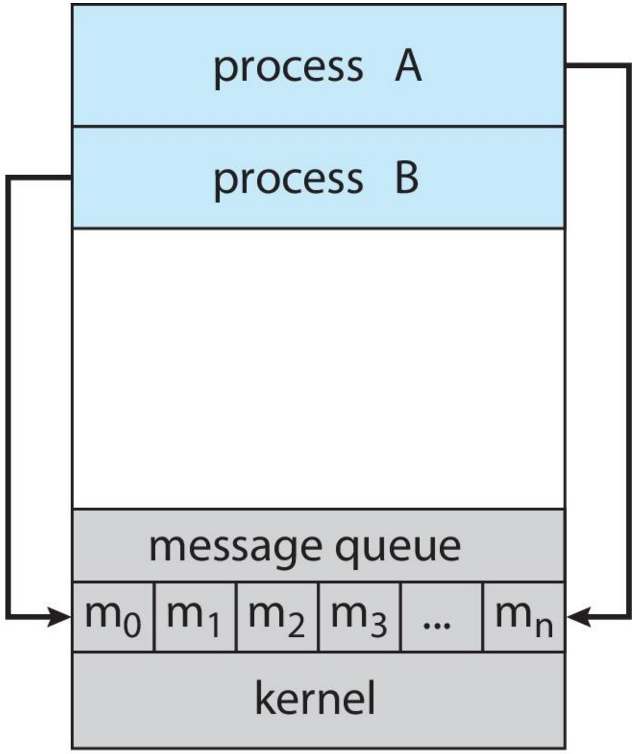


(b)

(b) 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

SHARED MEMORY VS MESSAGE PASSING



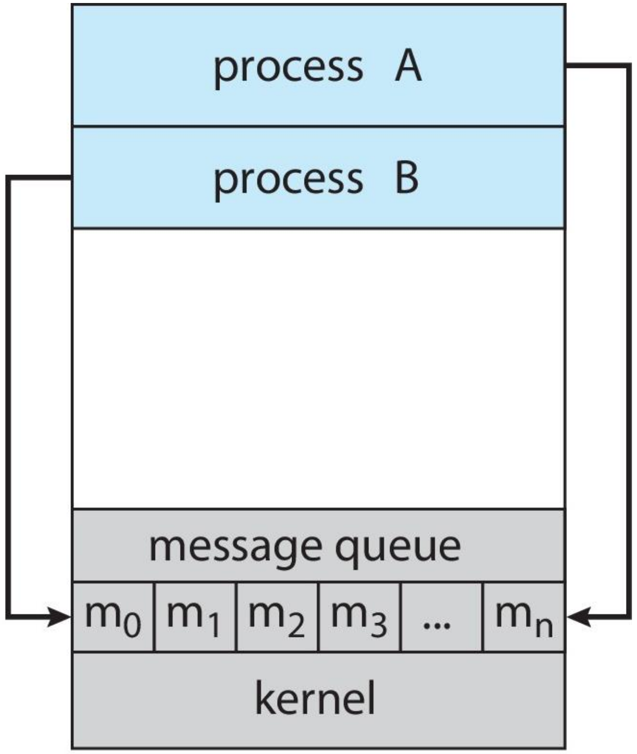
(b)

(b) 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
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.

SHARED MEMORY VS MESSAGE PASSING

(b) Message passing

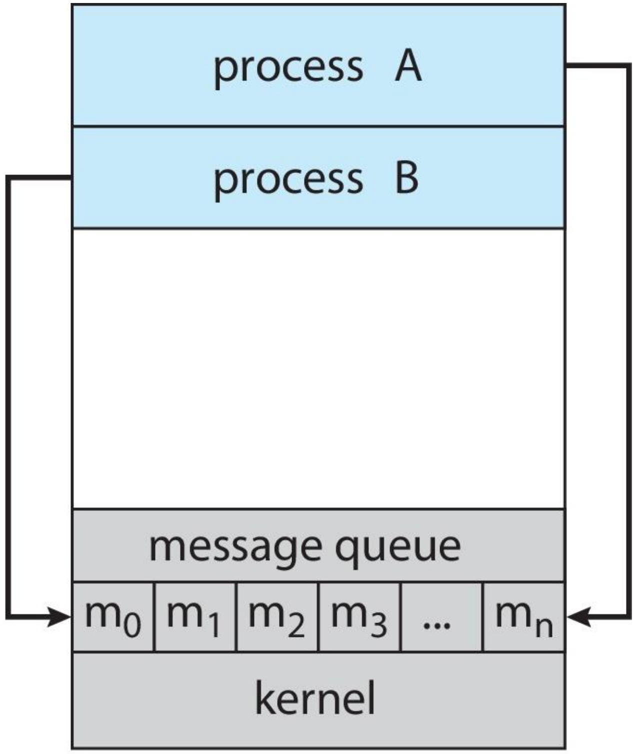


(b)

- 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
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.
- Disadvantage?

SHARED MEMORY VS MESSAGE PASSING

(b) Message passing

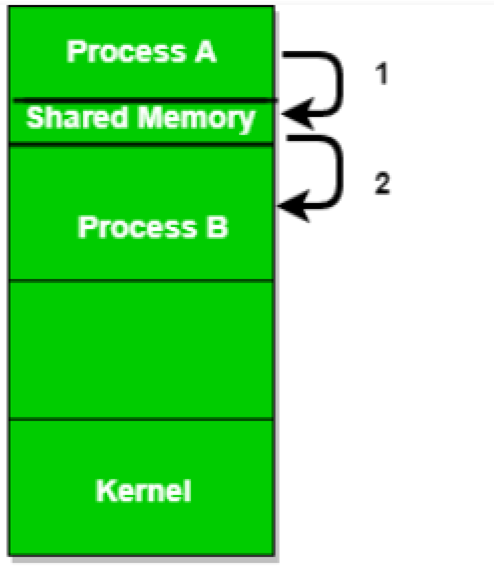


(b)

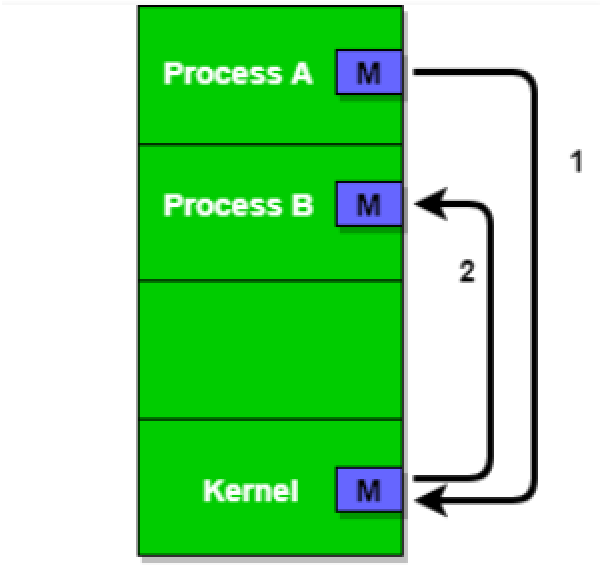
- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.
- Disadvantage: Requires more operations and more read/writes than shared memory.

SHARED MEMORY VS MESSAGE PASSING

(a) Shared memory



(b) Message passing



POSIX SHARED MEMORY

SHARED MEMORY EXAMPLES: POSIX SHARED MEMORY

- POSIX: Portable Operating System Interface
- Standards specified by the IEEE Computer Society
- Objective: OS have unified API allowing portability of software.

POSIX SHARED MEMORY

- POSIX: Portable Operating System Interface
- Standards specified by the IEEE Computer Society
- Objective: OS have unified API allowing portability of software.

Most Linux systems are partially or fully compliant to the POSIX standards making it fairly easy to port code.

POSIX SHARED MEMORY

- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O_CREAT);
    ```

  - Also used to open an existing segment.
  - Set the size of the object

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

  - **Map shared memory object to the process’s address space**

    ```
    void* addr = mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd , 0);
    ```

IPC - MESSAGE IMPLEMENTATIONS

- Pipes
- Sockets
- Local/Remote Procedure Calls

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)

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
- Windows calls these **anonymous pipes**

BLOCKING VS UNBLOCKING

- **Blocking** is considered **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 **asynchronous**
  - **Non-blocking send --** the sender sends the message and continue
  - **Non-blocking receive --** the receiver receives:
    - A valid message, or
    - Null message

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.
- Interrupt blocked read/write thread and wake it up.

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

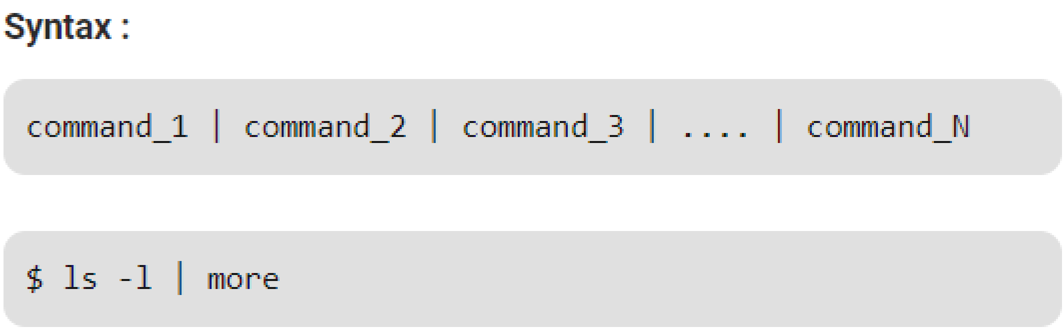
Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.
- Interrupt blocked read/write thread and wake it up.
- Create a worker thread just for the read/write. The program can continue running in main thread.

PIPES IN UNIX

- The vertical bar | is the pipe operator in unix shell.
- The transferred data is never saved in a file, it is simply communicated to the other process.
- Unnamed or “ordinary” pipes are destroyed after the process completes execution.



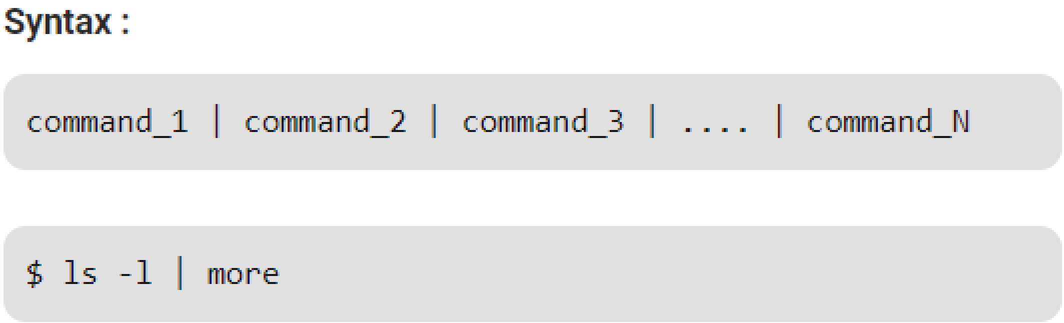
Syntax :

```
command_1 | command_2 | command_3 | .... | command_N
```

```
$ ls -l | more
```

PIPES IN UNIX

- The vertical bar | is the pipe operator in unix shell.
- The transferred data is never saved in a file, it is simply communicated to the other process.
- Unnamed or "ordinary" pipes are destroyed after the process completes execution.
- Named pipes can be created by the mknod() system call with the 'FIFIO' option:
  `mknod("mypipe",SIFIFO,0)`
- You can also use mkfifo("name",0666)
- Named pipes allow communication between any two processes



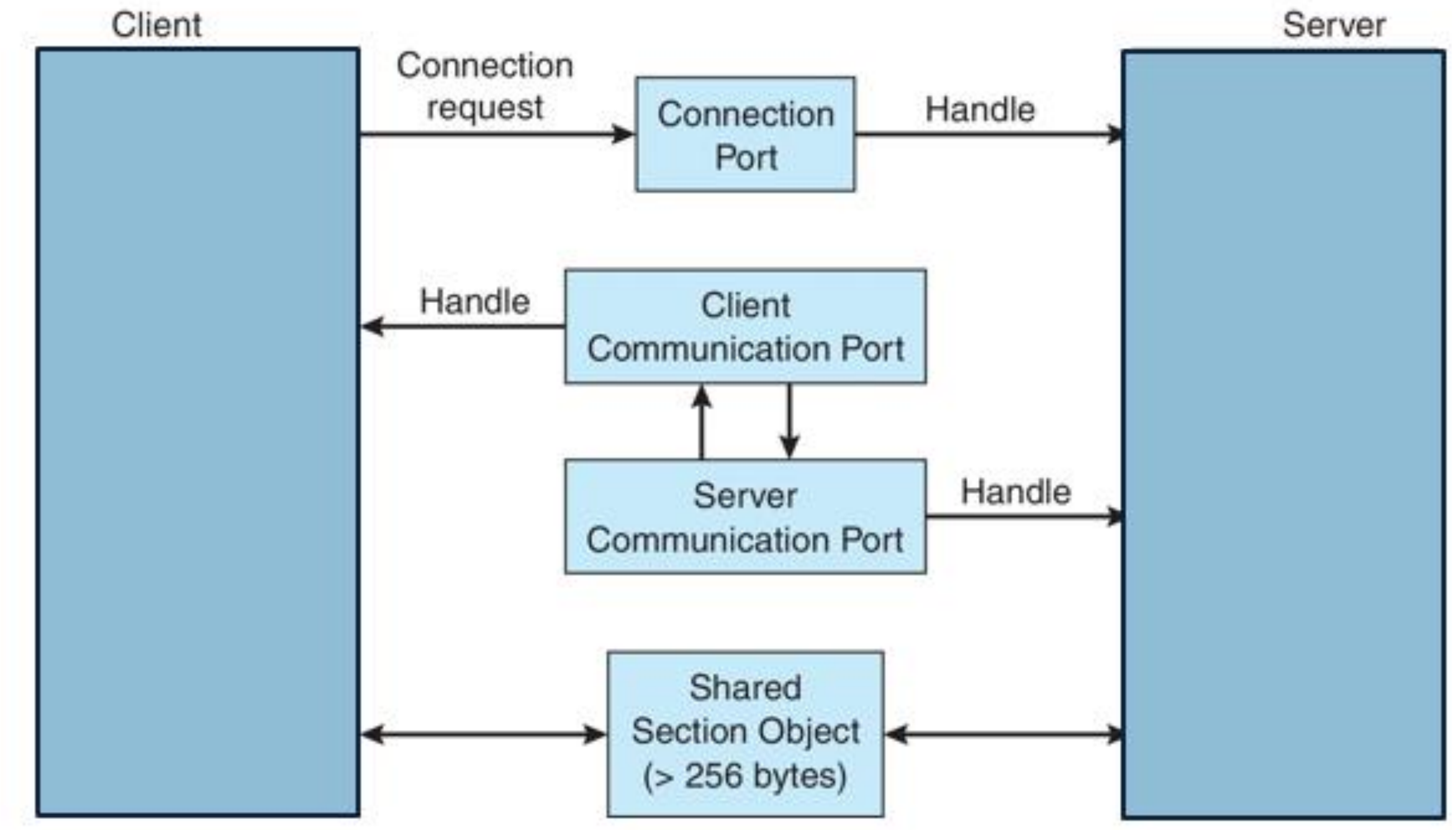
Syntax :

```
command_1 | command_2 | command_3 | .... | command_N
```

```
$ ls -l | more
```

LOCAL PROCEDURE CALLS IN WINDOWS

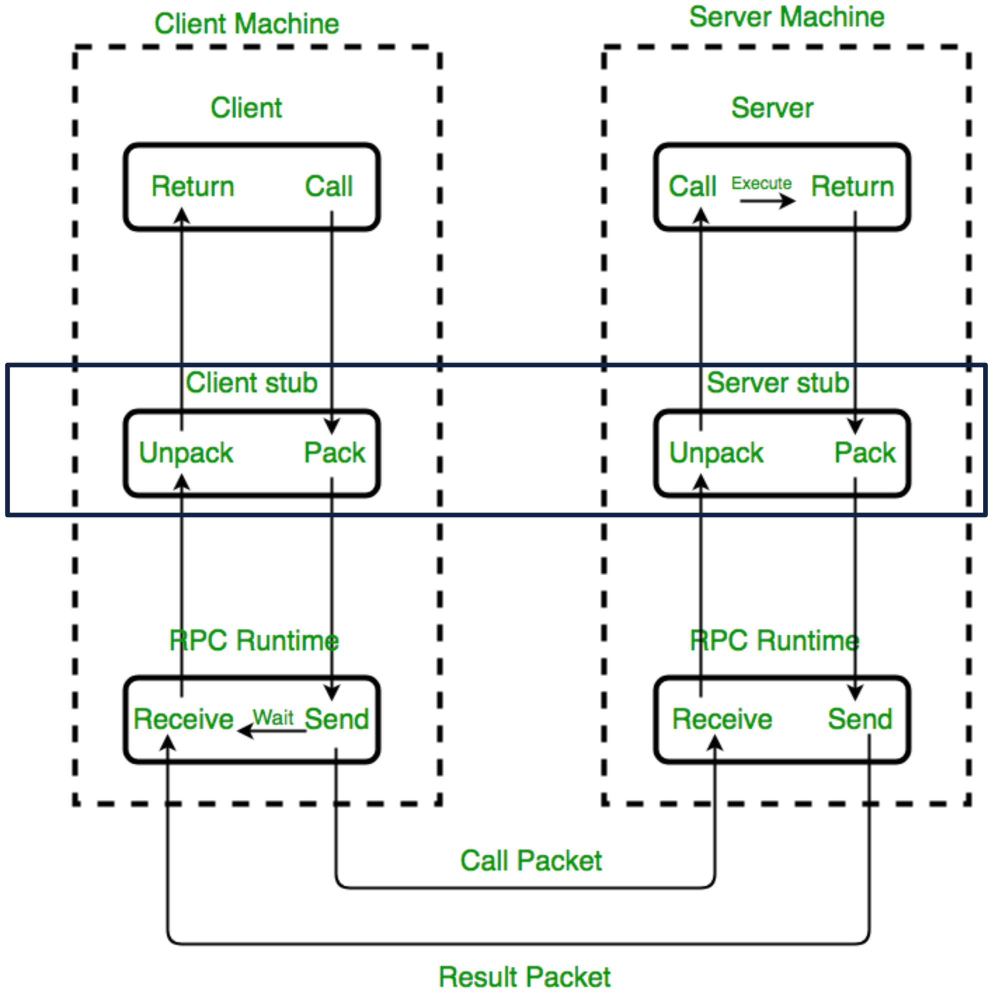
- Message-passing centric via **advanced local procedure call** **(LPC)** facility
  - Only works between processes on the same system
  - Uses ports to establish and maintain communication channels



Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (> 256 bytes)

REMOTE PROCEDURE CALLS

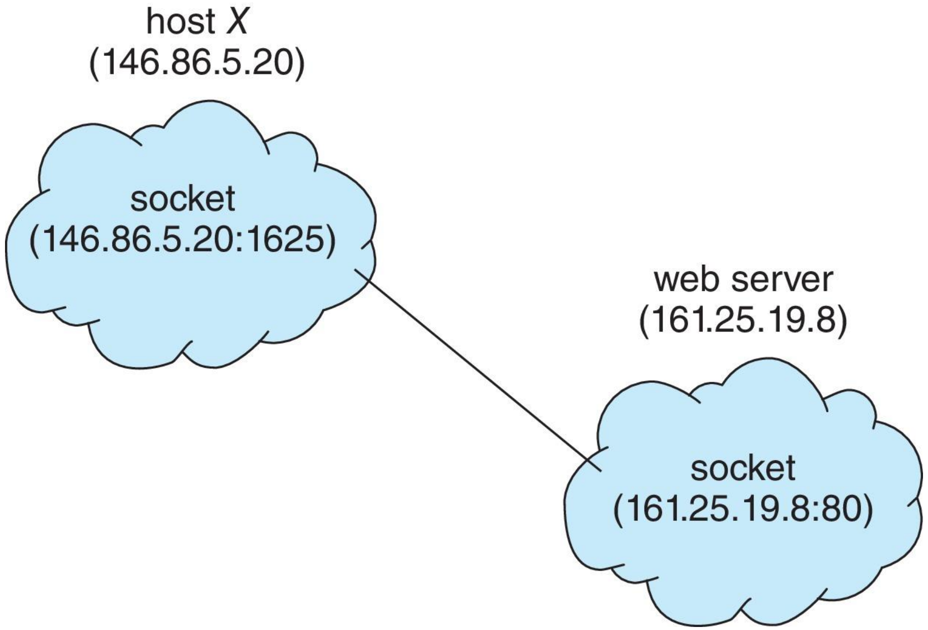
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** (packs) the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



Implementation of RPC mechanism

SOCKETS

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between 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 system on which process is running


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