

Operating Systems ICS 431

Weeks 4-5

Chapter 3: Process Management

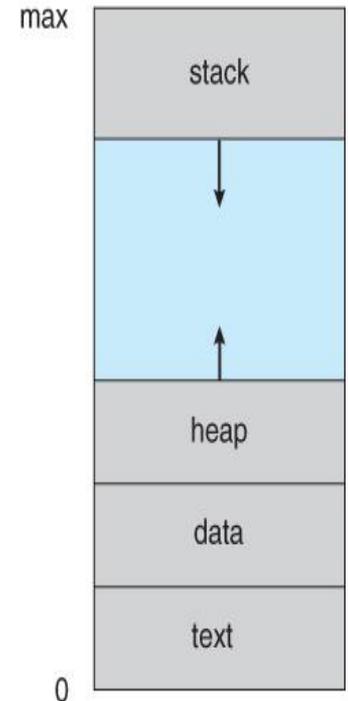
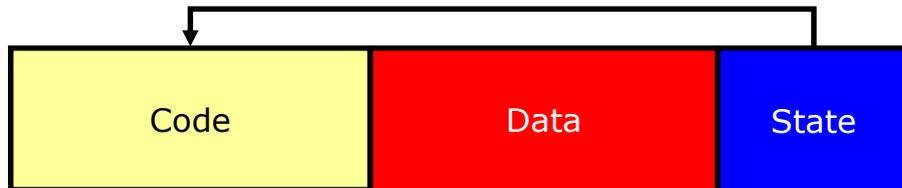
Dr. Tarek Helmy El-Basuny

Process Management

- **In this chapter, we will discuss:**
 - **The Process Concept:** Definition, Components, Sharing, ...
 - **Process states:** New, Ready, Running, Waiting, Terminated
 - **Process Control Block (PCB):** Its contents and dependency of HW and SW
 - **Context switching among processes and its overhead**
 - **Reasons for the Context Switching:** Multitasking, Interrupts, Mode switch, ..
 - **Process Scheduling Queues:** Job, Ready and Device queues
 - **Process Scheduling:** Scheduling evaluation Criteria: **CPU utilization, fairness, responsiveness**
 - **Different Types of Schedulers:** Short, Medium, and Long-Term Schedulers
 - **Operations on Processes:** **Creation, Interpretation, Execution, and Termination,**
 - **Parent and Children Processes:** Resources, execution, and address space sharing
 - **Reasons for child Process Termination:** Voluntary and Involuntary termination
 - **Cooperating/Dependent & Independent Processes,** advantages of cooperating Processes
 - **IPC through Message Passing:** Blocking or non-blocking message sending and receiving
 - **IPC through Buffering (Shared Memory):** Buffer size constrains on the communicating processes
 - **Producer-Consumer processes as an example of cooperating processes & their synchronization**
 - **Inter-Process Communication (IPC):** Direct and Indirect modes
 - **IPC in Client-Server processes:**
 - Sockets, Remote Procedure Calls, Remote Method Invocation (Java)

The Process Concept

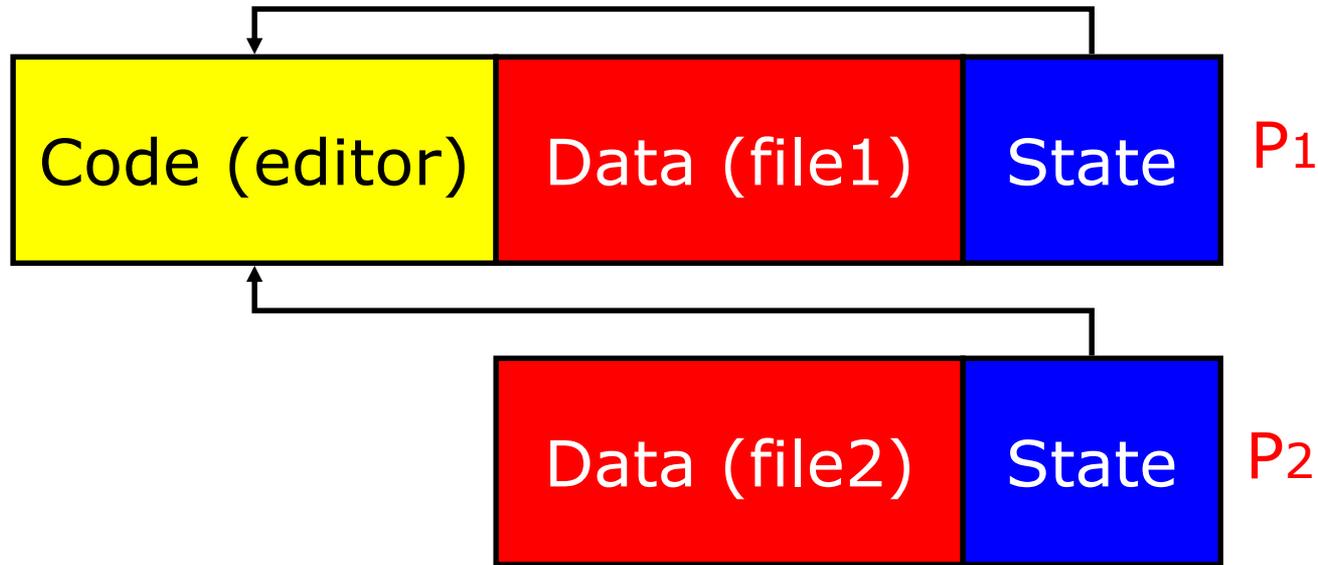
- A **Program** is a passive entity; just a sequence of instructions or lines of code to solve a certain problem.
- A **Process** is an instance of a program in execution.
 - A **Process** consists of information about the running program, i.e.:
 - Where in the execution sequence it is,
 - The state of the process,
 - A portion of memory allocated to it,
 - A bunch of resources allocated to it,
- **A Process in memory includes:**
 - **Code/Text** section (contains the compiled code of the program)
 - **Data** section (stores global and static variables, allocated and initialized prior to execution).
 - **State** (Newly created, Running, Waiting, ...).
 - The **heap** is used for dynamic memory allocation, and is managed via calls to new, malloc, delete, free, etc.
 - The **stack** is used for local variables.



A process in memory

- A **Thread** is a child process, or lightweight process, or a sequential flow of control within a process.

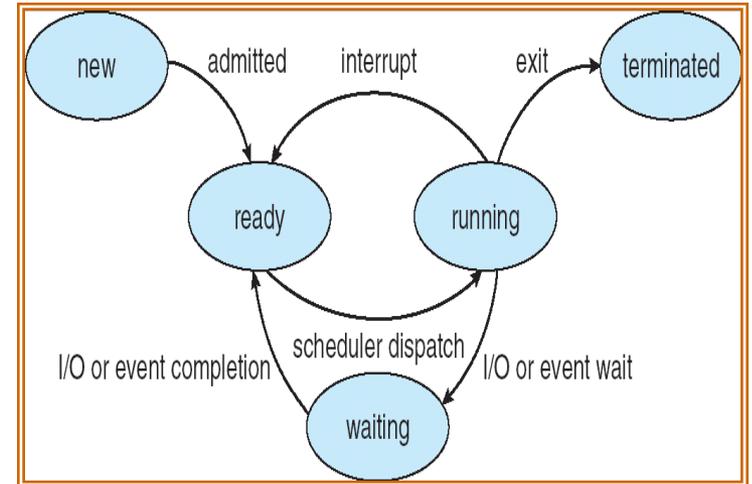
Can processes share the code section and **why?**



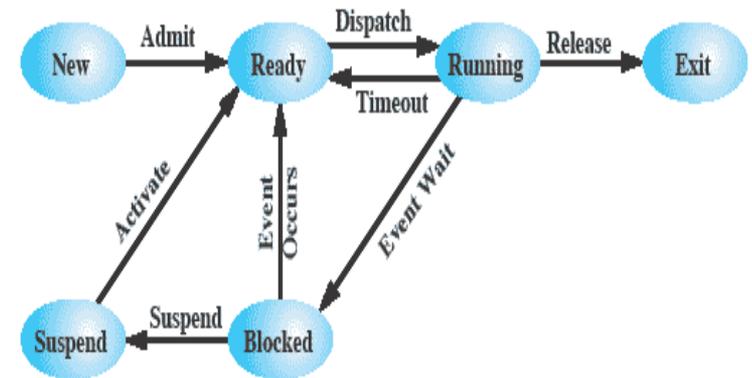
- Yes, two concurrent instances of processes can share the code section of the parent process.
 - **Example:** Opening two word files on the same machine, both will use the same code section while each will have its own data and state.
- **Why?** to maximize the memory utilization.

Process States: Five-Six State Model

- **Newly created:**
 - A process has been created but has not yet been admitted to the pool of executable processes, *i.e.*
 - Submission of a batch job
 - User logs on
 - Process creates a child process.
- **Ready**
 - Processes with all needed resources are available and can be allocated.
- **Running**
 - Dispatched to the CPU
- **Blocked/Waiting**
 - A process that cannot execute until a specified event such as an IO completion occurs (Waiting for I/O).
- **Suspended:** *i.e.* due to limited memory availability.
- **Terminated:**
 - Batch job issues a **Halt instruction**
 - User logs off
 - Quit an application
 - Error and fault conditions that can not resolved.



5-States Model



(a) With One Suspend State

6-States Model

Reasons for Process Termination

- **Normal completion** after finishing its job.
- **Time limit exceeded** due to some fatal reasons or the process waited longer than a specified maximum for an event.
- **A process requires more memory to execute** but the system fails to provide enough memory to the process for its execution.
- **Protection error** has been occurred, *i.e.* write to a read-only file, etc.
- **Arithmetic error**, *i.e.* div/zero, etc.
- **I/O failure**: When a process attempts to use an I/O device and I/O device is not working fine at the moment. *i.e.*, a process that wants to print a file on the printer, but the printer is defective.
- **Invalid instruction**: Happens when trying to execute data or to execute an instruction that is reserved for only OS.
- **OS intervention**: In some critical cases, the OS takes control of the process and stops the execution of the process. *i.e.*, if a deadlock occurs, or deadlock can occur,
- **Parent Request**: If a parent process request for terminating the child process. Then, the child process should be terminated.
- **Parent Termination**: When the parent is not in CPU, child process can't exist in CPU, child process also needs to be terminated.
- **Bounds violation**: When the process tries to access non assigned memory spaces or disks.

Process Control Block (PCB)

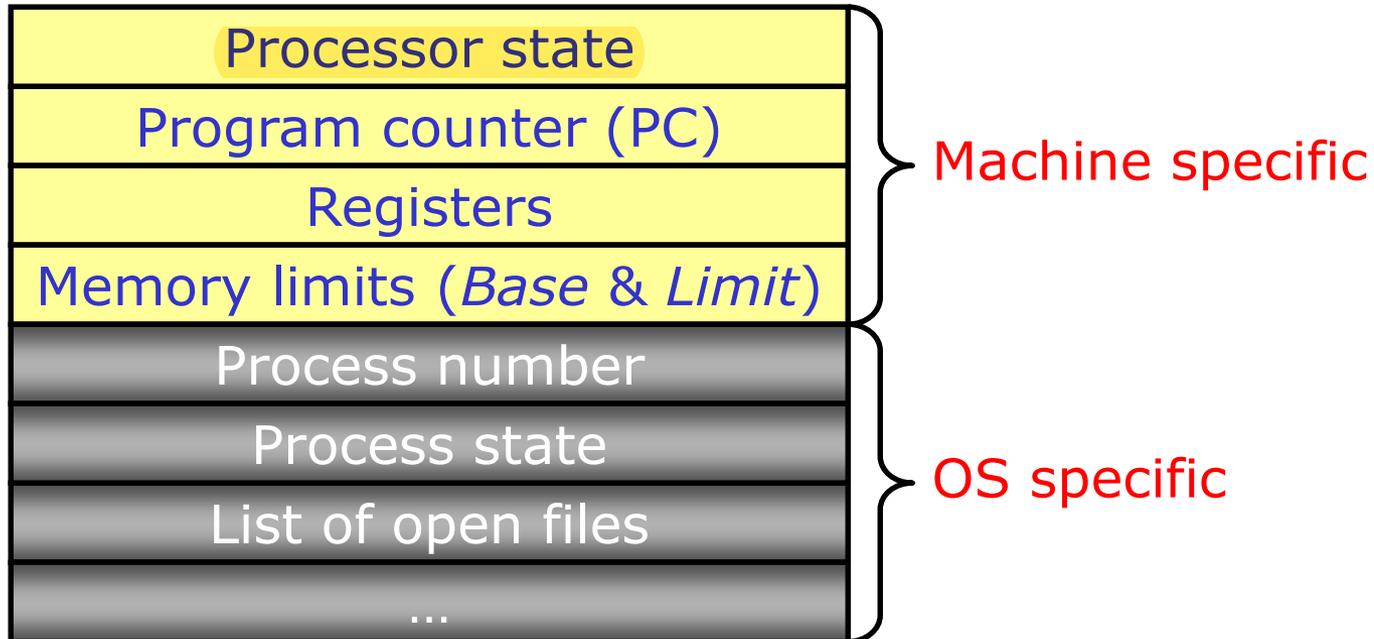
- The OS must know specific information about the process to manage it.
- PCB (Process Descriptor in Linux) is the data structure that stores the following information about the process:
 - **Process ID** (a unified name assigned by the OS, i.e. an integer or a table index, etc.),
 - **Process State** (the current state, such as new, running, waiting, ready, terminated),
 - **Owner** (identified by the owner's internal identification, such as user's login name),
 - **Program Counter** (the address of the next instruction to be executed by the process),
 - **CPU Registers** (contents of , accumulator register, index registers, stack pointers, condition-code bits and other general purpose registers)
 - **CPU Scheduling Information** (process priority, pointers to scheduling queues and other scheduling parameters),
 - **Process Privileges** (Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed),
 - **Parent Process** (a pointer to the PCB of the parent process),

Process Control Block (PCB)

- **List of children/siblings** (pointers to a list of children processes of this process),
- **Protection Information** (description of the access rights currently held by the process),
- **Memory Management Information** (such as contents of **base and limit registers**, pointers to **page tables** and **segment tables**),
- **Accounting** (usage information such as amount of CPU needed & used, time limits, memory space required),
- **I/O Information** (list of I/O devices allocated to this process, pointers to wait-queues etc),
- **Resources controlled by the process may be indicated**, such as opened files, history of processor utilization; this information may be needed by the scheduler.
- **Inter-process Communication** Various flags, signals, and messages may be associated with communication between two independent processes.

Process Control Block (PCB)

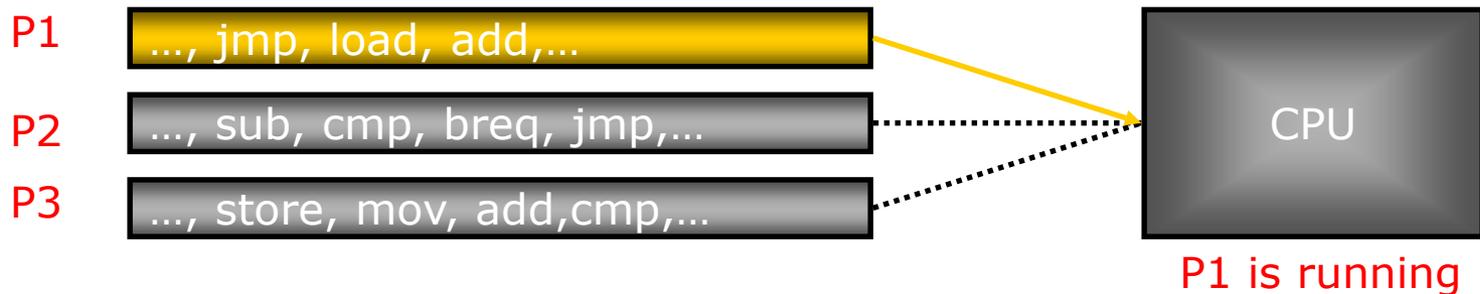
- Some of the PCB contents are machine specific and others are OS specific, i.e.



- When an interrupt occurs, the contents of [Acc, SP, general purpose registers, index register, base and limit registers] must be saved in the PCB to allow the process to resume correctly.

Concurrency of Execution

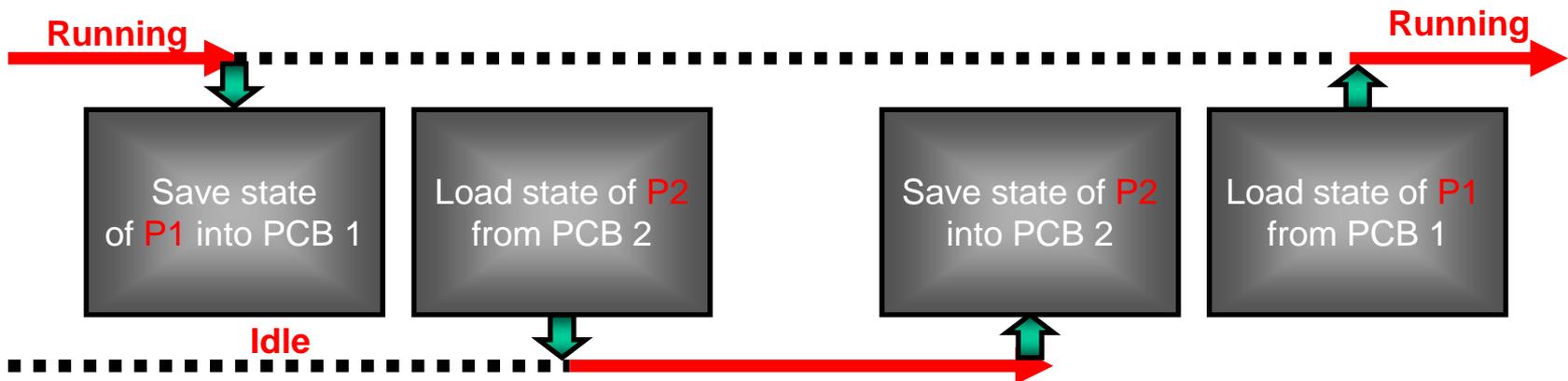
- In a **single processor** system, if one process can run at a time, this makes poor CPU utilization.
- The multiprocessing OS is recommended on single processor machines.
- The objective of multiprocessing is to maximize CPU utilization by **concurrent processing**:
 - Alternate the execution of more than one process.
- Concurrency can be achieved by switching the processor among several processes, i.e. **I/O and Computing Instructions interleaving**.
- Switching the CPU from one process to another is called **Context Switch**.



- **Processes can be described as either:**
 - **I/O-bound process:** Spends more time doing I/O than computations.
 - **CPU-bound process:** Spends more time doing computations.

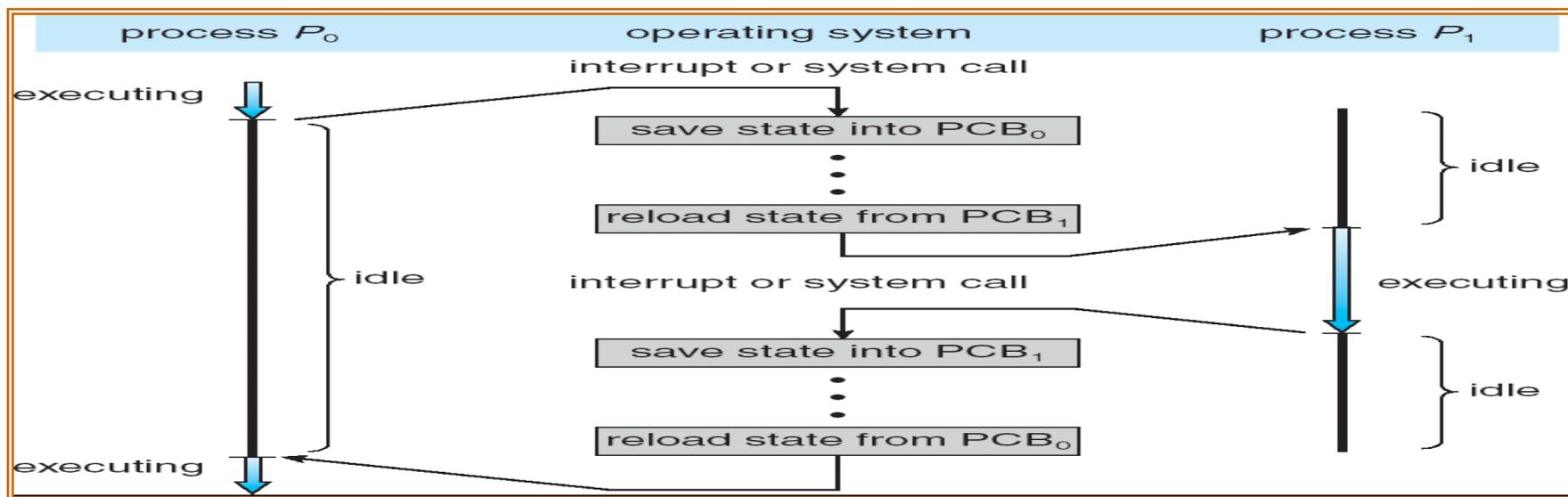
Process Context

- Switching from one process to another process in multiprocessing is called “**Process Context**”. This allows multiple **processes** to share a single CPU, and is an essential feature of a multiprocessing **operating system**.
 - The OS gives the CPU to another process whenever the running one is waiting for an I/O operation to complete.
- In a **context** switch, the OS stores the state of a **process/thread**, so that it can be restored and resumed from the same point later.
- We say that process **P** is active if and only if:
 - Its address space is in memory,
 - Its PCB data is loaded into the CPU registers.
- When a process **P** is interrupted (e.g. waiting for I/O), then its context is **not active**.
- **Process Context** requires a certain amount of CPU time and should not be frequently done.



Overhead of Context Switching

- A **context** switch occurs whenever an interrupt or an exception occurs, or when a process issues a system call.
- Switching from one process to another process requires a certain amount of time for (saving and loading registers, memory maps, updating various tables and lists, etc.)



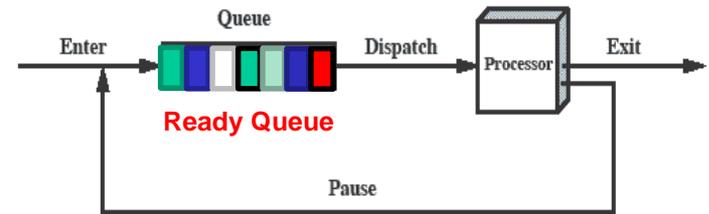
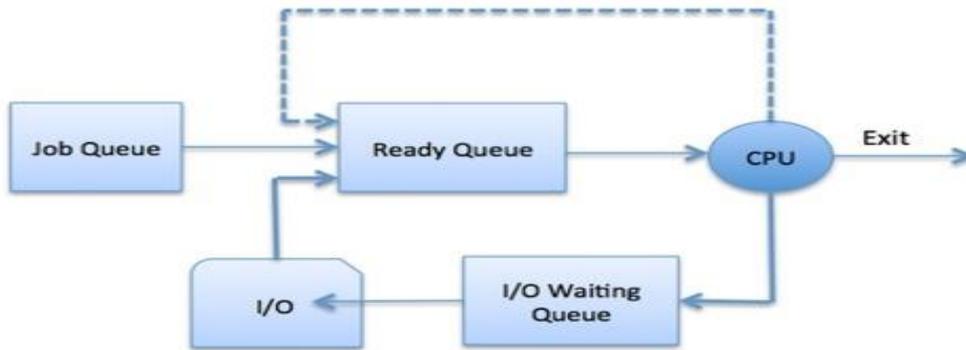
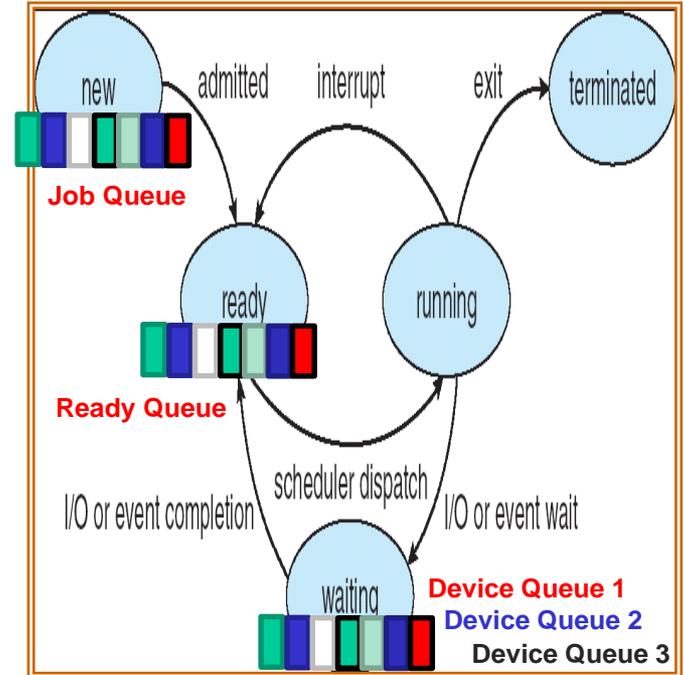
- The system does no useful work while switching.
 - Losing CPU time in loading and storing registers from/into main memory.
 - Switching-time depends on hardware support.
- Context switches are usually computationally intensive and much of the operating systems design is to optimize the overhead of context switches.

Reasons for Context Switch

- There are **three** situations where a **context switch** needs to occur:
 1. **Multitasking/Multiprocessing**: according to the scheduling policy, one process needs to be switched out of the CPU so another process can run.
 2. **Interrupt/Exception handling**: Modern OSs are interrupt driven. This means if the CPU requests data from a disk, for example, it does not need to busy-wait until the read is over, it can issue the request and continue with some other execution; when the read is over, the CPU can be interrupted and presented with the read.
 3. **User and kernel mode switching**: When a transition between user mode and kernel mode is required in an OS. Some OSs may not consider a mode transition itself a context switch.

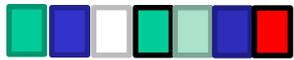
Process Scheduling Queues

- The OS maintains different queues to store the pointers to the PCBs of all processes in the same execution state.
- Job queue:** Stores the pointers of all newly created processes in the system.
- Ready queue:** Stores the pointers of all processes residing in main memory, ready and waiting to be executed by the CPU.
- Device queues:** Stores the pointers of all processes waiting for a certain I/O device. **A queue for each device.**
- Processes migrate among the various queues.

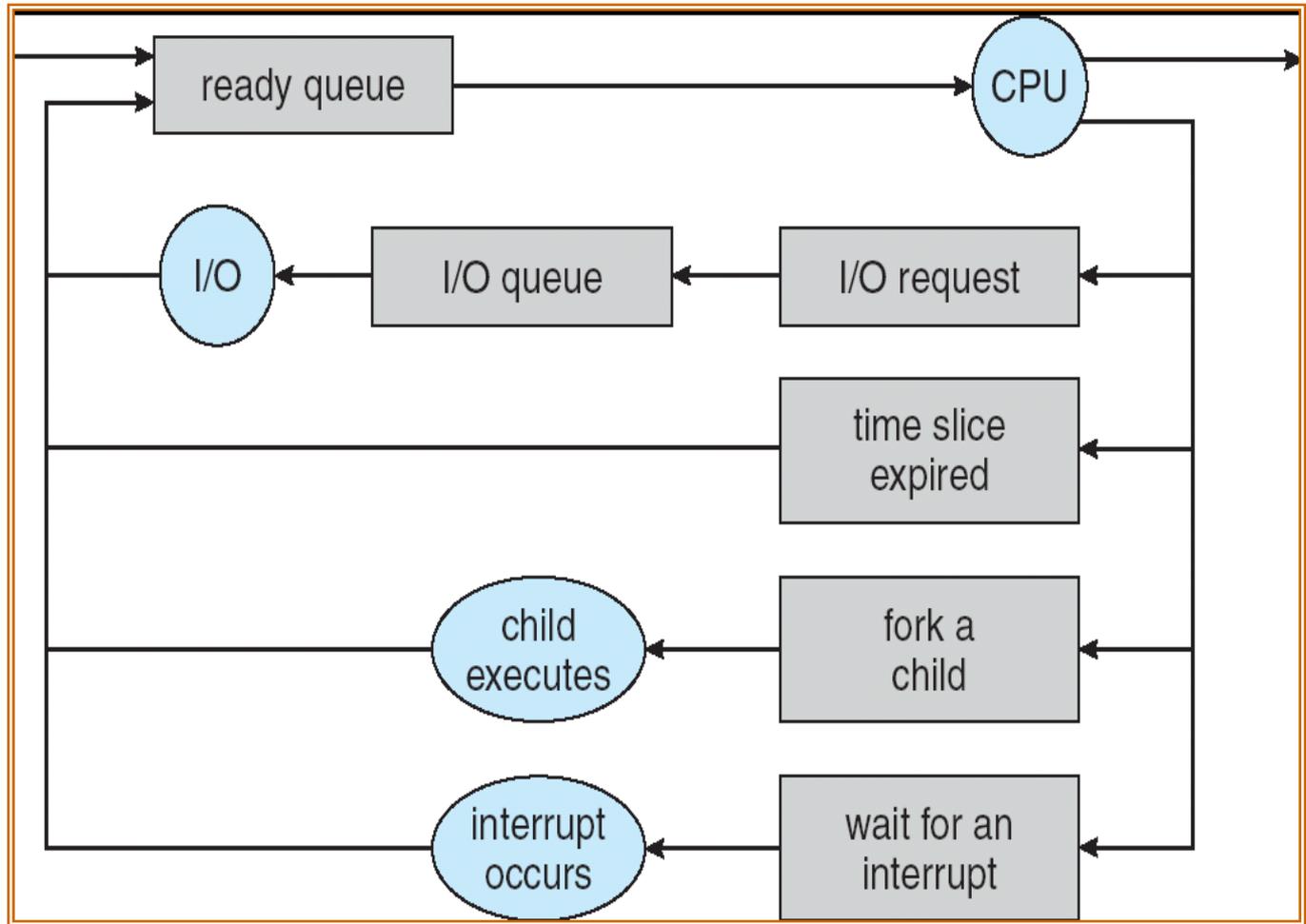


(b) Queuing diagram

Representation of Process Scheduling

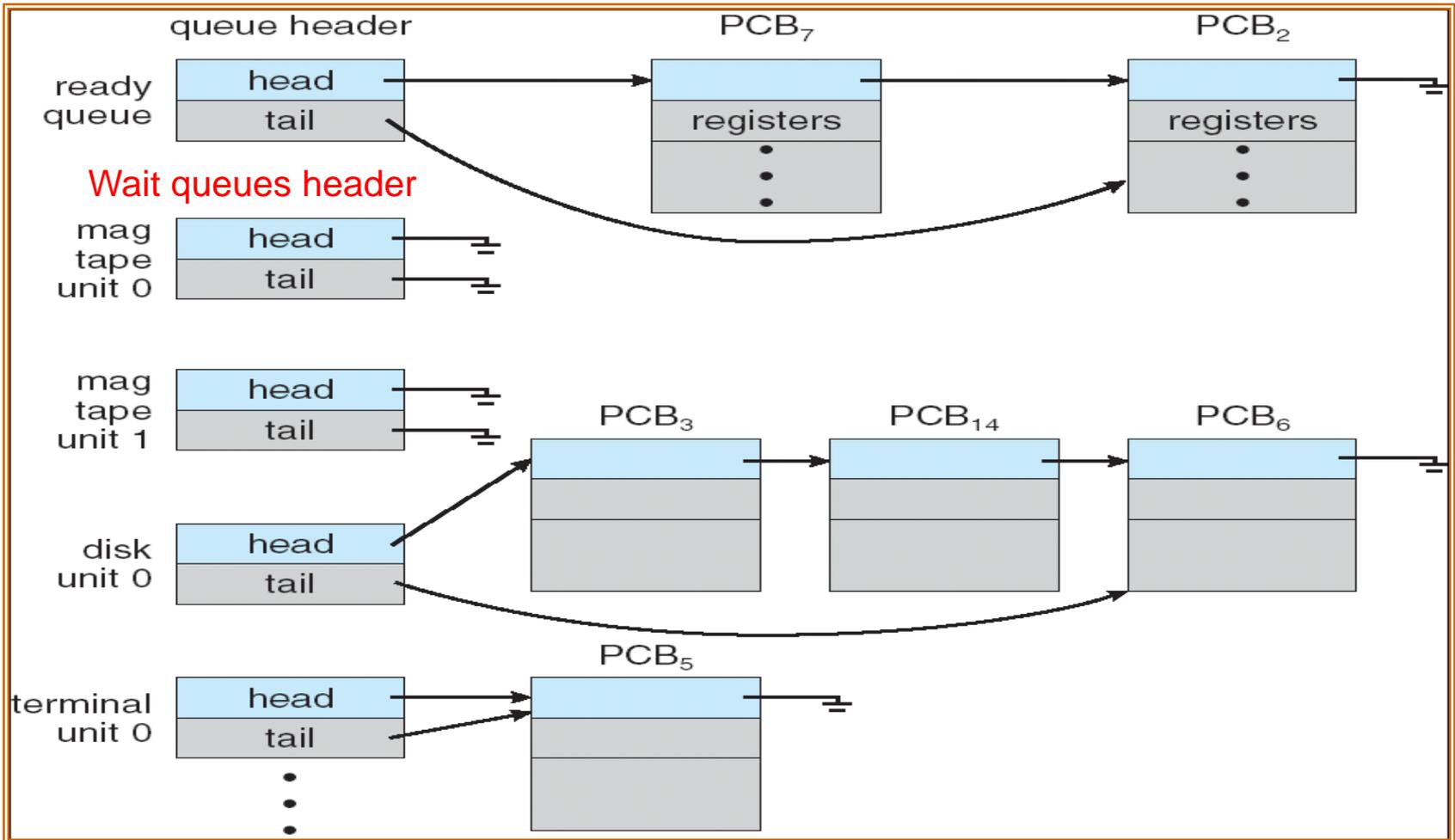


Job Queue



Ready Queue & Various I/O Device Queues

Ready queue header



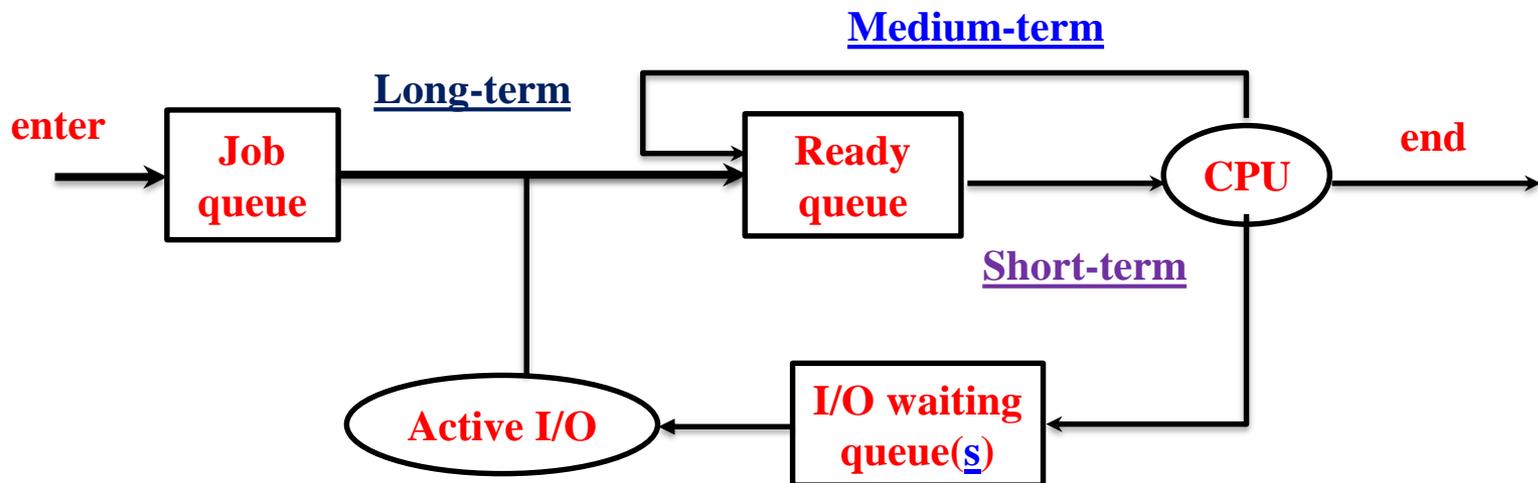
Each PCB includes a pointer field that points to the next PCB in the ready queue.

Process Scheduling

- **Process Scheduling:** means select one of the “**ready**” processes to run next, this decision must:
 - Improve/maximize CPU utilization (make it as busy as possible)
 - Improve user response time (user’s satisfaction)
 - Be fair among concurrently running processes or multi-users.
- **The efficiency of Process Scheduling is measured by two parameters:**
 1. CPU utilization = $\frac{\text{Time CPU is doing useful work for the processes}}{\text{Total time elapsed}}$
 2. Response time = (Process arrival time - Process start time)
- These two goals are often contradictory
- Given a set of processes, finding an optimal scheduling policy that maximizes CPU utilization, minimizes the response time and supports fairness among processes is a hot research issue in the OS field.

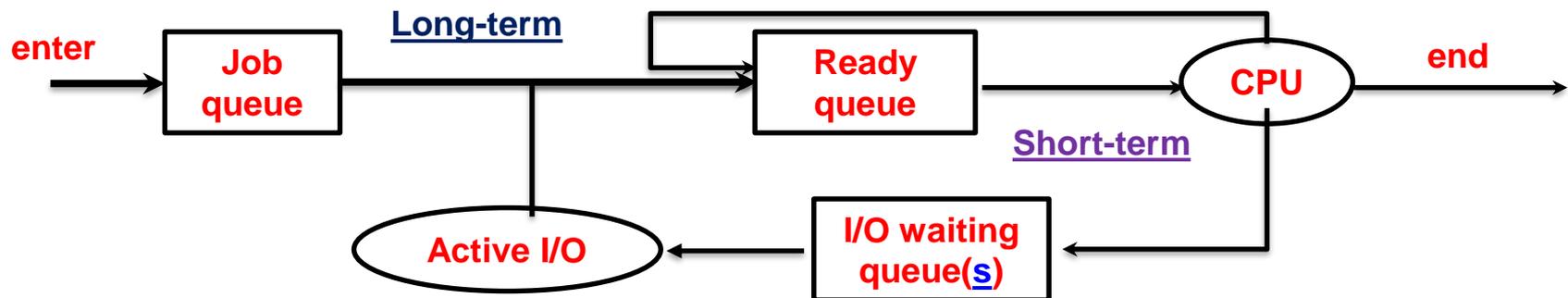
Schedulers

- Since, processes migrate among the various scheduling queues (**Job queue, Ready queue, I/O queues**) throughout their lifetime.
- The OS must select processes from these queues in some fashion.
- The process selection is carried out by the appropriate scheduler (i.e. **Long-term scheduler, short-term scheduler, medium-term scheduler**).



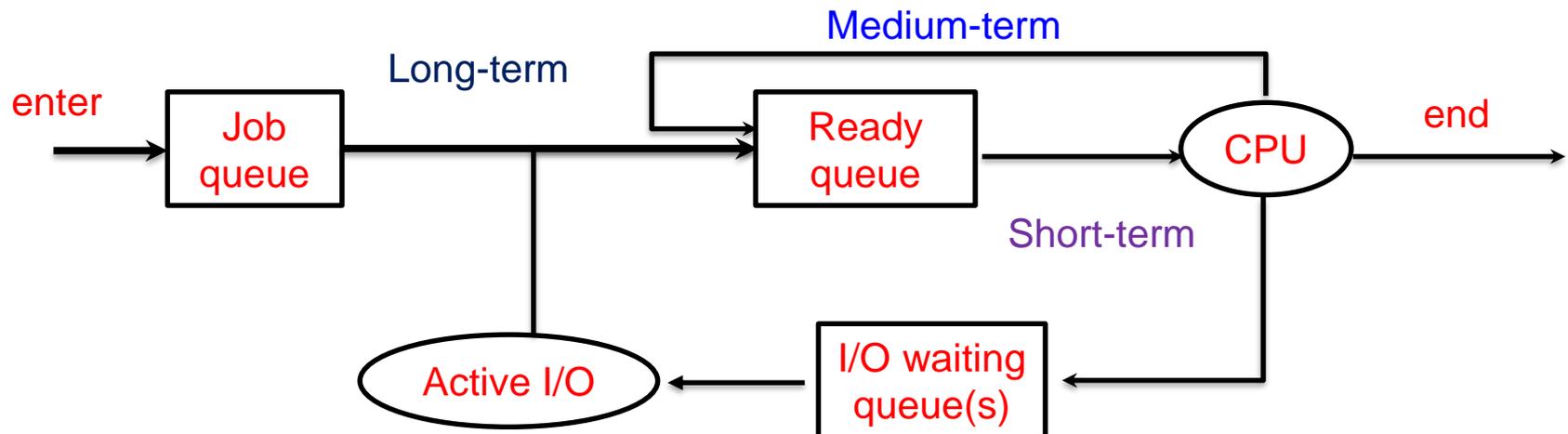
Long-term Scheduler

- **Long-term scheduler:** acts when a new process is created, it decides whether to be brought into the ready queue or no?
 - This scheduler dictates what processes are to run on a system and the degree of concurrency to be supported at any time – i.e. whether a high or low amount of processes are to be executed concurrently.
 - (no intelligence required) take the process that its required resources are available.
- **Long Term Scheduler:**
 - Runs rarely
 - Controls degree of multiprocessing/concurrency
 - Tries to balance arrival and departure rate through an appropriate process mix (I/O bound and CPU bound).



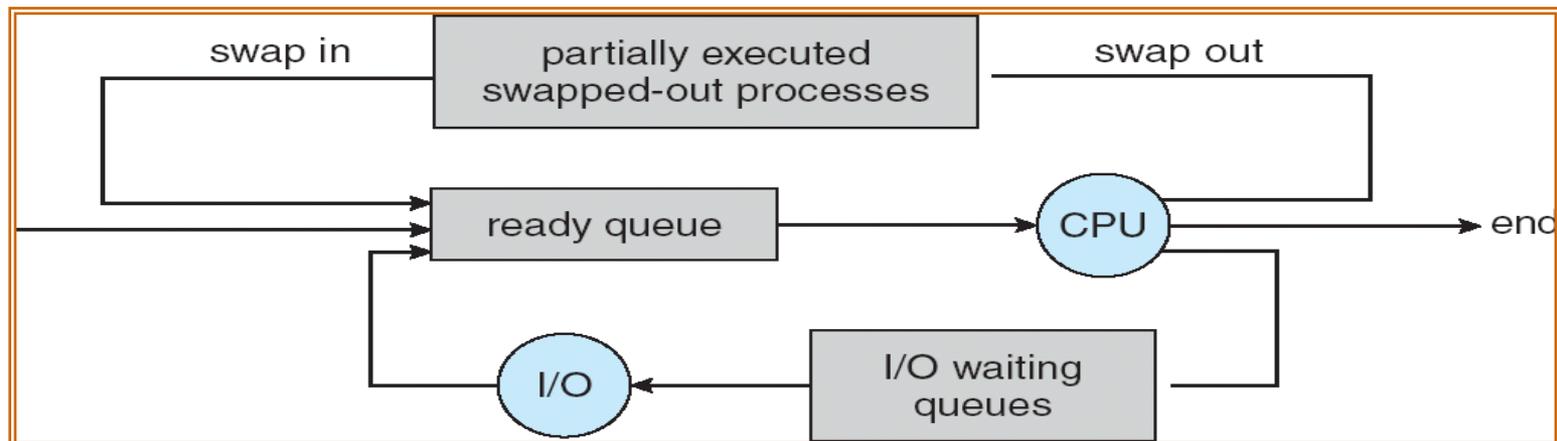
Short Term Scheduler

- **Short-term scheduler:** Selects which process should be executed next (to be dispatched into the CPU).
 - It needs a policy for the selection to make a balance (i.e. FCFS, SJF, SRTF, RR, Priority, etc. we will study them in chapter 5)
- **The Short Term Scheduler** runs very frequently and contains:
 - Code to remove a process from the processor at the end of its run.
 - Process may go to ready queue, or to a wait state or finish/quit.
 - Code to select a process from the ready queue.



Medium Term Scheduler

- The mid-term scheduler exists in all systems with virtual memory support, temporarily swaps out processes from the main memory and places them on virtual memory or vice versa.
- It may decide to swap out a process:
 - Has not been active for some time,
 - Has a low priority,
 - With high page fault frequently,
 - Taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or
 - etc....

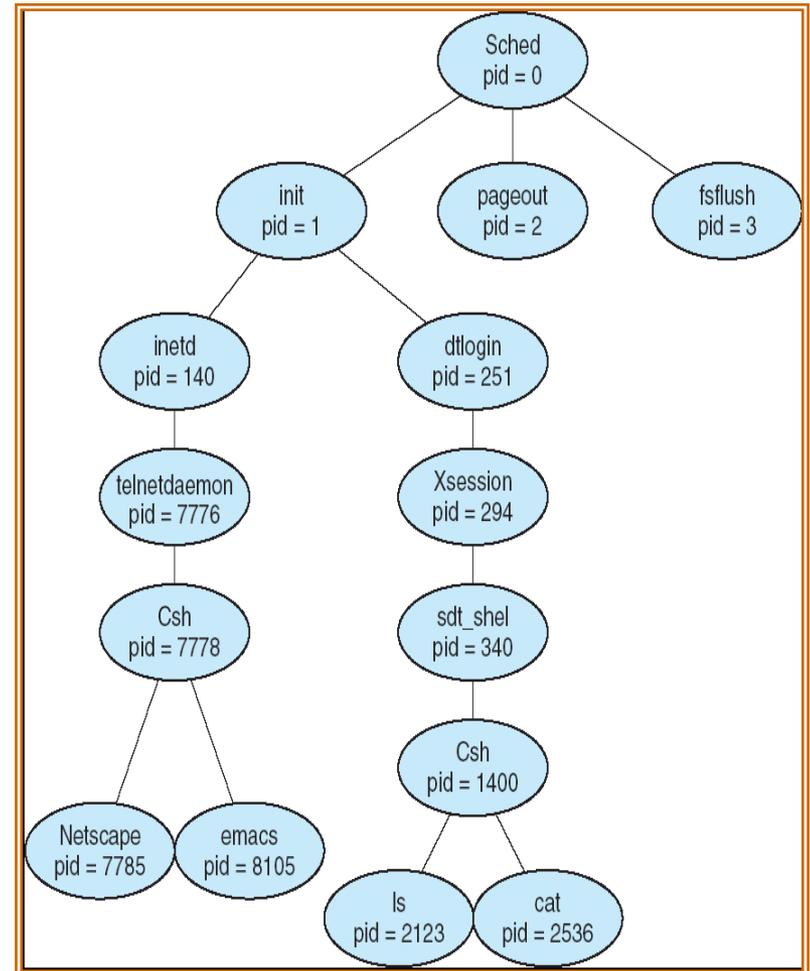


A Process from its Creation to the Termination

- Operations on Processes include: Interpretation, **Creation**, executing, **terminating**:
- User types a “**xxx**” at CLI’s shell or double click on “**xxx**” at GUI’s shell.
 - Note: shell is the interface where we can interact with the OS.
- The command will be parsed and interpreted by the “**shell**” command interpreter.
- The executable program “**xxx**” needs to be located on disk (**file system, I/O device driver for disk**).
- The content of the program **xxx** will be loaded (**load module**) into memory and the **control transferred to the OS ==> process comes alive!** .
- Required resources will be verified and **dispatch** the process into the CPU for running.
- During execution, the process may call OS to perform I/O (console, disk, printer, etc.) (**system call interface, I/O device drivers**).
- While running the process, it may create a child process.
- When the process terminates, the allocated memory and resources will be reclaimed. (**memory management**).

Parent and Children Processes

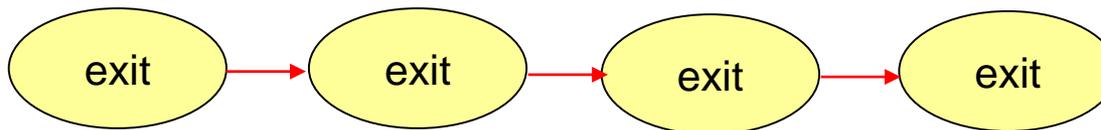
- The OS allows a parent process to create children processes, which, in turn may create other children processes, forming a tree of processes.
- **Resources sharing**
 - The parent and children share all resources.
 - The children share subset of parent's resources.
 - The parent and the child share no resources.
- **Execution**
 - The parent and children execute concurrently.
 - The parent waits until children terminate.
- **Address space**
 - A child process is a duplicate of a parent process.
 - A child has its address space loaded into it.



Child Process Termination

Reasons for a child Process Termination:

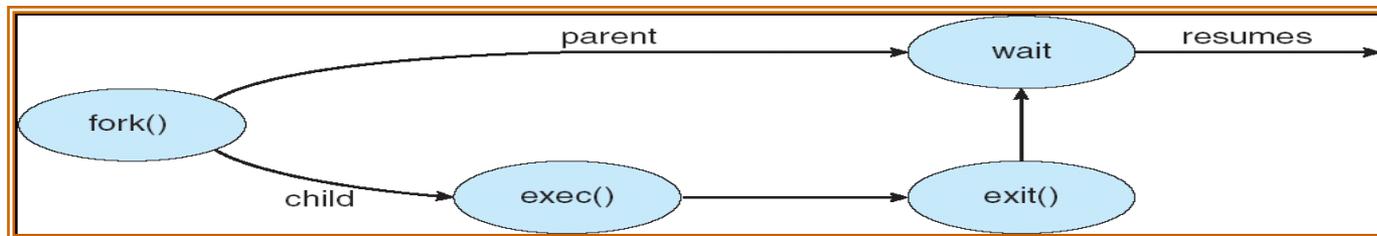
1. Normal exit (**voluntary**), due to the completion of its job.
2. Error exit (**voluntary**), an error caused by the child process and can not be served by the OS.
3. Fatal error (**involuntary**), trying to run a program that is not exist.
4. Killed by the parent process (**involuntary**) if:
 - The child has exceeded allocated resources.
 - The task assigned to the child is no longer required.
 - The parent is exiting.
 - Some OSs do not allow a child process to continue if its parent has been terminated (**cascading termination**).



Process Management in UNIX

Basic commands of process management in Unix: (i.e. creation, execution, waiting, exiting, killing)

- `fork()` command creates a child process that inherits copies of all parent's variables.
- `exec()` command allows a process to “load” a child and start executing it.
- `exit()` command causes normal process termination (closes all open files, connections, de-allocates memory, de-allocates most of the OS structures supporting the child process), and checks if the parent is alive then it holds the result value until the parent requests it.
- `wait()` command puts the parent to sleep waiting for a child's result.
- `ptrace()` command allow a parent process to observe and control the execution of a child process.
- `nice()` command can be used to reduced the priority of a process and thus be 'nice' to the other processes.
- `sleep()`, command delays the execution start time of a command by some number of seconds that the user specifies.
- `kill() <pid>` command will terminate a process with the process id `<pid>`. The `pid` of a process can be obtained using the 'ps' command.
- `ps` command gives information about the process including the `pid`, terminal name, time of creation and name of the process, etc...



Process Management in Linux-1

- Create a process by running a program (**writing its name then press enter key from the CLI.**)
- You can run many processes **either foreground or background** concurrently.
- You can also move the process from foreground to background by the commands **fg process name** or **bg process name** but you need to close the process CTRL+Z first.
- Use **top** command to tell the user about all the running processes in Linux, or **tasklist** in windows. It displays:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2016	root	20	0	70716	40m	4780	S	1.0	4.0	0:06.86	Xorg
2423	guru99	20	0	75144	29m	9888	S	0.3	3.0	0:02.35	ubuntuone
2568	guru99	20	0	73188	13m	10m	S	0.3	1.4	0:00.53	gnome-ter
2631	guru99	20	0	2820	1160	864	R	0.3	0.1	0:00.03	top
1	root	20	0	3328	1828	1260	S	0.0	0.2	0:01.13	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd
5	root	20	0	0	0	0	S	0.0	0.0	0:00.42	kworker/u

- Where, **PID** is the user ID number. **User** is the owner of the process. **PR** is the priority (20 high to -20 low).
- **NI** is the nice value (**priority index** and can be changed **like priority**) of the process, **VIRT** is the amount of the **virtual memory** taken by the process in KB.
- **RES** is the physical memory used in KB.
- **SHR** is the shared memory used.
- **S** is the status of the process (sleeping/S or running/R or traced/stopped/T or ...)
- **%CPU** and **%MEM** are the % of CPU time and memory used. **TIME+** is the total time.

Process Management in Linux-2

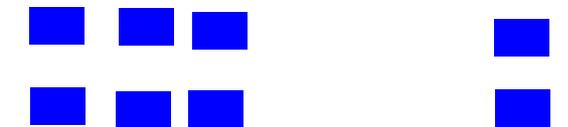
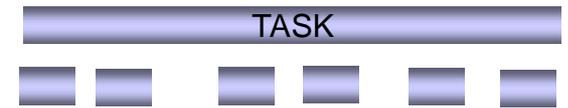
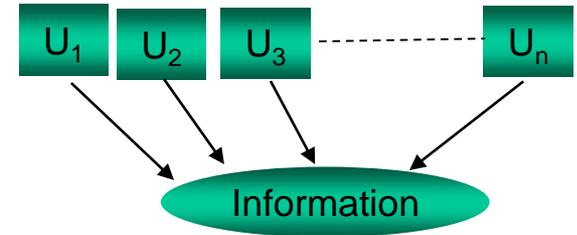
- Use **ps** command displays the process status. Like task manager in windows machine.
- **ps ux** to display the status of all the processes under this user, or **ps PID** for only one process. You can get the PID of the process by **pidof process name**.
- **kill** command terminates a process running on the Linux machine. i.e. **kill PID, or taskkill /PID in windows**
- **nice** command starts a process with the given priority.
- **renice** changes the priority of a process (changes from -20 to 19, the default value is 0). Its syntax is **nice -n nice-value process-name**.
- **df** gives us the free HD space on the system.
- **free** gives us the details of the free RAM space on the system.

Independent & Cooperating/Dependent Processes

- **Independent process:** a process that is independent of the rest of the processes. It does not affect or be affected by the execution of another process. **OS support is:**
 - Its state is not shared in any way by any other process.
 - It does not share any information with other processes.
 - It is ok to run independent processes in parallel on separate processors.
- **Dependent/Cooperating process:** a process that affect or be affected by the execution of another process. **OS support is:**
 - Passing information between processes
 - Making sure that processes do not interfere with each other
 - Ensuring proper sequencing of dependent operations

Why do we need Cooperating Processes?

- **Information sharing:** several processes may be interested in the same piece of info.
- **Computation speed-up:** for a particular process to run faster, it could be broken into sub-processes, each of which executes in parallel especially if the computer has multiple processing elements (CPU's or I/O channels).
- **Modularity:** helps construct the program in a modular fashion dividing the system functions into separate cooperating processes.
- **Convenience:** even an individual user may have many processes on which to work at one time. The user may be editing, printing, compiling in parallel. This will enhance the user's satisfaction.



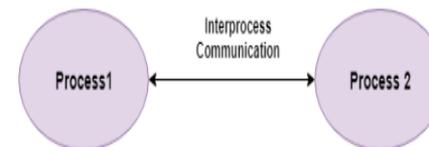
Inter-Processes Communication Models

- Inter-process communication is the mechanism provided by the OS **that allows cooperating processes** to communicate with each other. i.e.
 - A process letting another process know that some event has occurred or transferring of data from one process to another

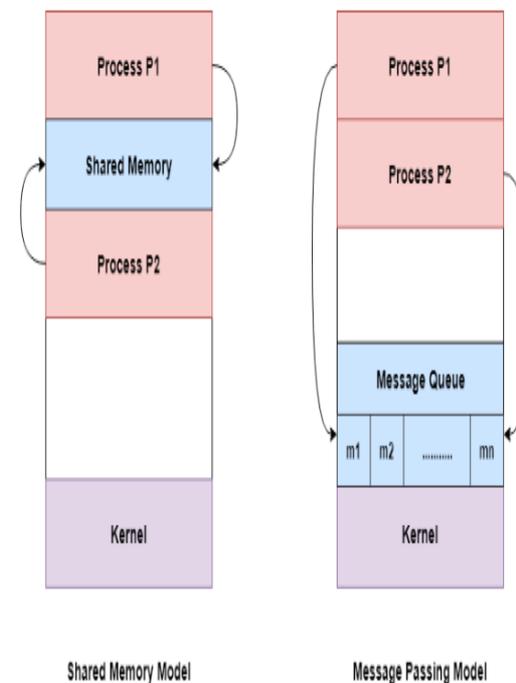
The models of inter-process communication:

❑ Shared Memory Model

- Shared memory is the memory that can be simultaneously accessed by multiple processes.
- **Advantage of Shared Memory Model**
 - Memory communication is faster on the shared memory model as compared to the message passing model on the same machine.
- **Disadvantages of Shared Memory Model**
 - All the processes that use the shared memory model **need to make sure that they are not writing to the same memory location.**
 - Required synchronization and memory protection that need to be addressed.

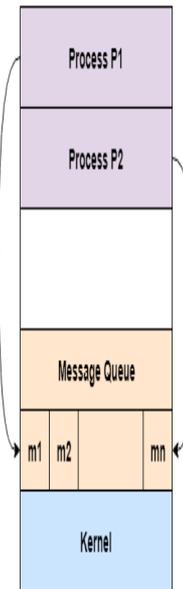


Models of Interprocess Communication

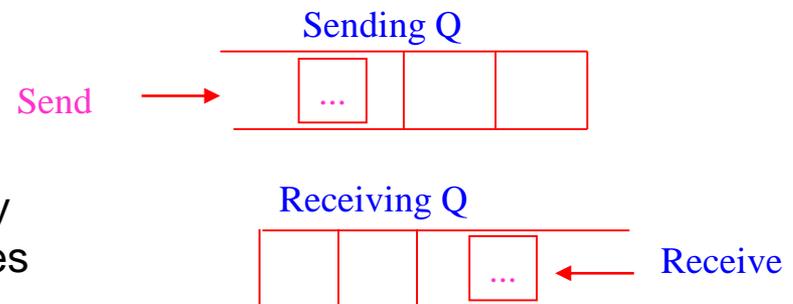


Inter-Processes Communication Models

- **Message Passing Model:** Is the mechanism provided by the operating system that allows processes to read and write data to the message queue without being connected to each other.
 - i.e. processes P1 and P2 can access the message queue and store and retrieve data.
- Messages are stored on the queue until their receiver retrieves them.
- Message queues are quite useful for inter-process communication and are used by most OSs.
- **Advantage of Messaging Passing Model**
 - The message passing model is much easier to implement than the shared memory model.
- **Disadvantage of Message Passing Model**
 - The message passing model **has slower communication** than the shared memory model because the connection setup takes time.

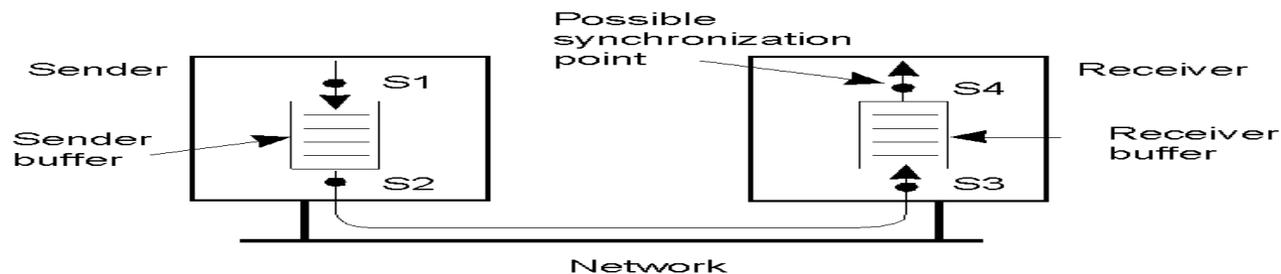


Message Passing Model



Cooperating Processes: Possible Synchronization

- Messages exchanged by communicating processes reside in a temporary **queue/buffer**.
- A **buffer/queue of messages** could be provided by the **sender's kernel**, **receiver's kernel**, and/or in the **communication network**.
 - Can be logically combined into one big buffer.
- A queue/buffer assigned to the processes implemented in one of three ways:
 1. **Zero capacity**: Max. length is **0** message (**means no buffering**), sender must wait for receiver (**rendezvous**).
 2. **Bounded capacity**: Finite length of **n** messages can be buffered, sender must wait if the queue is full.
 3. **Unbounded capacity**: Infinite length, **means can buffer any produced message**, sender never waits.



Blocking and Non-Blocking Message Passing

- Message passing supported in two different modes (either **blocking/synchronous** or **non-blocking/asynchronous**).
- Send and receive primitives may be either blocking or non-blocking.
- **Blocking/synchronous send**: means the sending process is blocked until the message is received by the mailbox/**buffer** or the receiving process.
- **Non-blocking/asynchronous send**: means the sending process sends the message and restarts operation.
- **Blocking receive**: means the receiver process blocked until a message is available in the buffer.
- **Non-blocking receive**: means the receiver process retrieves either a valid message or null.

IPC Issues

- Major issues of IPC:
 - Is it direct or indirect addressing?
 - Is it blocking or non-blocking communication?
 - Is it reliable or unreliable communication?
 - Is it buffered or un-buffered communication?
- Purpose of IPC:
 - Data Transfer
 - Sharing Data
 - Event notification
 - Synchronization
 - Mutual Exclusion ; assure that only one is executing at a time
 - Process Control; notification through Signals (i.e. wait, exec, sleep, kill)

Direct Process Communication

- If **P** and **Q** processes wish to communicate, they need to:
 - Establish a **communication link** between them
 - Exchange messages via **send** and **receive** commands
- The **IPC mechanism** allows processes to communicate and to synchronize their actions.
- IPC facility provides **two operations**:
 - **Send**(message) to a process – message size may be static or variable.
 - **Receive**(message)
- Implementation of the communication link
 - **Physical** (e.g., shared memory, hardware bus)
 - **Logical** (e.g., initiating sockets, ports,..)
- The OS provides IPC mechanisms for processes to communicate and to synchronize their actions without sharing the same address-space.
- Good for distributed environment.

Direct Process Communication

- Processes must name each other explicitly:
 - **Send**(P, message) – send a message to process P.
 - **Receive**(Q, message)- receives a message from process Q
 - **Receive**(ID, message)- receives a message from the sender with ID.
- **Properties of direct 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.
 - Receiver may not need ID of the sender (**known by default**).

Disadvantage of Direct process Communication:

- A process must know the name or ID of the process it wishes to communicate with.
- They can't be easily changed since they are explicitly named in the send and receive.

Indirect Process Communication

- Messages are directed to and received from mailboxes or Ports.
- A mailbox is an object into which messages can be placed by processes and from which messages can be removed by other processes.
- Ownership of the Mailbox:
 - Process owns it:
 - Only the owner may receive messages through this mailbox.
 - Other processes may only send.
 - When the process terminates any “owned” mailboxes are destroyed.
 - System owns it:
 - Process that creates mailbox owns it and receives through it
 - When the process terminates the system transfers ownership to the parent process.
- IPC mechanism provides operations (system calls) to:
 - Create a new mailbox
 - Send and receive messages through the mailbox.
 - Destroy a mailbox
- Primitives (system calls) are defined as:
 - **Open** (mailbox_name);
 - **Send** (A, message) – sends a message to mailbox A.
 - **Receive** (A, message) - receives a message from mailbox A.

Indirect Process Communication

- **Properties of communication link:**
 - Each mailbox has a unique ID.
 - Processes can communicate only if they share a 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.

Mailbox sharing Problems:

- P_1 , P_2 , and P_3 share mailbox A.
- P_1 sends;
- Who gets the message (P_2 and/or P_3 ?)

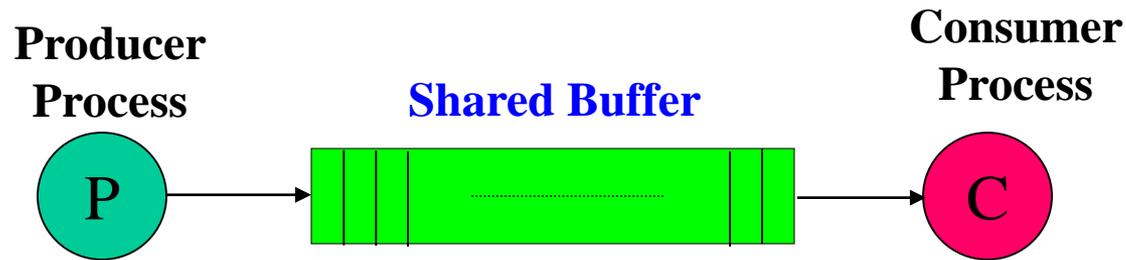
• Solutions:

- 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 randomly the receiver.
- Sender is notified who the receiver was.

Cooperating Processes: Producer-Consumer

- As an example of cooperating processes: a **producer** process produces information that is consumed by a **consumer** process.
- **Cooperation processes** must use IPC Mechanisms to coordinate their execution, i.e..
 - **Message Passing Interfaces**, i.e. **Sockets, Streams, Pipes**, etc.
 - **Shared Memory**: Non-message passing systems
- i.e., if a buffer is used by the producer and consumer process to communicate.
- The producer and consumer processes must be synchronized based on the size of the used buffer. i.e.
 - **With unbounded-buffer where no practical limit** on the size of the buffer.
 - The consumer process may have to wait for a new item, if the buffer is empty, **but the producer always produces items.**
 - **With bounded-buffer where there is a fixed buffer size.**
 - The producer process must wait if the buffer is full and the consumer must wait if the buffer is empty.
 - The buffer may be either provided by the OS IPC facility, or coded by the application programmer using shared memory.
- **Dangers of cooperating processes without synchronizing of their processing**
 - **Data corruption, deadlocks, increased complexity.**

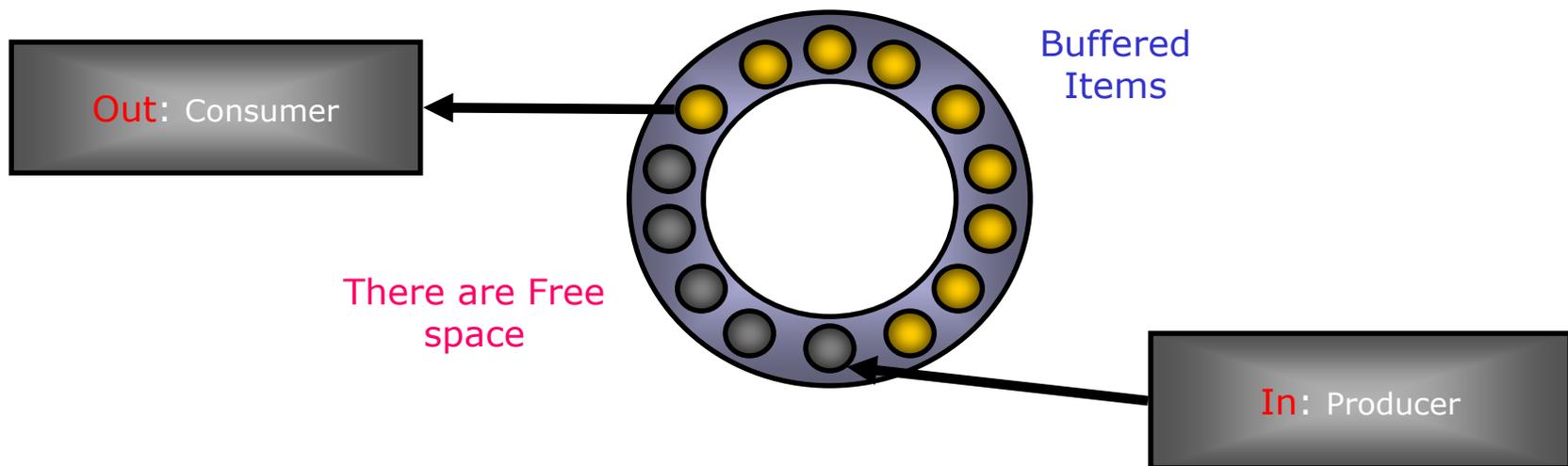
The Producer-Consumer Processes



- From time to time, the producer places an item into the buffer.
- The consumer removes an item from the buffer.
- Careful synchronization/coordination is required.
 - The consumer must wait if the buffer is empty.
 - The producer must wait if the buffer is full.
- Typical solution would involve a shared variable called count to monitor the buffer size.

Bounded-Buffer Solution

- If the shared buffer is implemented as a **circular array** with two logical pointers, **in** & **out**.
 - **in** points to the next free position in the buffer where the producer puts an item.
 - **out** points to the first full position in the buffer where the consumer can get.
- When $in = out$, the buffer is empty.
- When $((in+1) \% BUFFER_SIZE) = out$, the buffer is full.



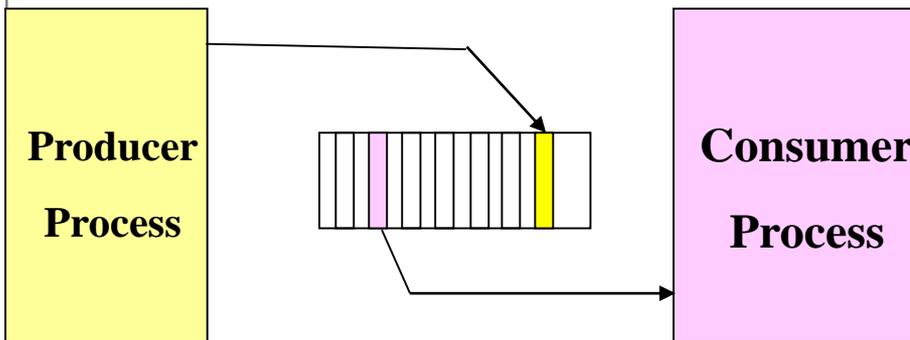
The Producer Consumer Processes

A **Producer** process "produces" information to be "consumed" by a **Consumer** process.

```
item    nextProduced;
while (1) {
    while (counter ==
    BUFFER_SIZE);
    /*do nothing*/
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    /*incremented every time we
    add element*/
    counter++;}
PRODUCER
```

```
#define BUFFER_SIZE 10
typedef struct {
    DATA        data;
} item;
item    buffer[BUFFER_SIZE];
int     in = 0;
int     out = 0;
Declaration
```

```
item    nextConsumed;
while (1) {
    while (counter == 0);
    /*do nothing*/
    nextConsumed = buffer[out];
    out = (out + 1)%
    BUFFER_SIZE;
    /*decremented every time we
    remove element*/
    counter--;
}
CONSUMER
```



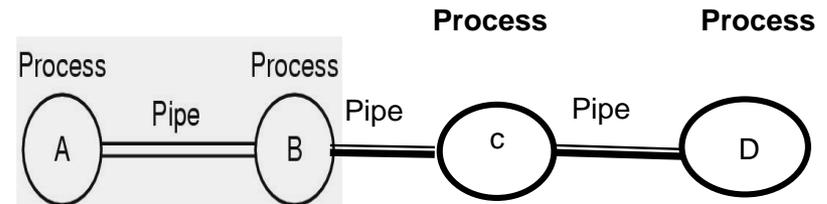
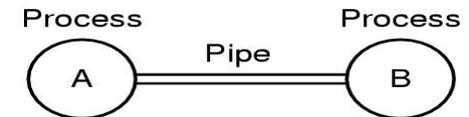
IPC Facilities in Linux

- The Linux kernel provides the following IPC mechanisms:
 - **Signals**: the kernel notifies a process when an event occurs by interrupting the process's normal flow of execution and invoking one of the signal handler functions registered by the process or the default signal handler by the kernel.
 - **Named Pipes or FIFOs**: Allows two processes that are not related to communicate. The processes communicate using named pipes by opening a special file known as a **FIFO file**. One process opens the FIFO file for writing while the other process opens the same file for reading.
 - **Anonymous Pipes**: Provides a mechanism for one process to stream data to another process, which in fact can send it to another process. A pipe has two ends associated with a pair of file descriptors. One for reading and the other for writing.
 - **Message Queues**: One process writes a message packet on the message queue and exits. Another process can access the message **packet** from the same message queue at a latter point in time.
 - **Shared memory**: Allows one process to share a region of memory in its address space with another. This allows two or more processes to communicate data **more efficiently** amongst themselves with **minimal kernel intervention**.
 - **Network Sockets**: Network Sockets API provides mechanisms for communication between processes that run on different hosts on a network.
- For more information of how to create and use:
 - <https://www.tldp.org/LDP/lpg/node7.html> or
 - http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf or
 - https://www.tutorialspoint.com/unix_commands/ipcs.htm

pipe system call for Processes Communication

- Pipe sets up communication channel between two processes. Communication is achieved by one process writing into the pipe and other reading from the pipe. **You need to do the following steps:**
- **Step 1:** Create a pipe.
- **Step 2:** Send a message to the pipe.
- **Step 3:** Retrieve the message from the pipe and write it to the standard output.
- **Step 4:** Send another message to the pipe.
- **Step 5:** Retrieve the message from the pipe and write it to the standard output.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
char *message = "This is a message!!!" ;
main()
{ char buf[1024] ;
  int fd[2]; // file descriptor (fd) and 2 means can act for reading and writing
  pipe(fd); /*create pipe*/
  if (fork() != 0) { /* I am the parent */
    write(fd[1], message, strlen (message) + 1) ;
  }
  else { /*Child code */
    read(fd[0], buf, 1024) ;
    printf("Got this from MaMa!!: %s\n", buf) ;
  }
}
```



- Sometimes useful to connect a set of processes in a pipeline.
- Process A writes to pipe AB,
- Process B reads from AB and writes to BC
- Process C reads from BC and writes to CD.

Shared Memory Processes Communication

- Common chunk of read/write memory among processes, here's a c code example to create a shared memory:

```
int shmget(key_t key (id), size_t size, int shmflg);
```

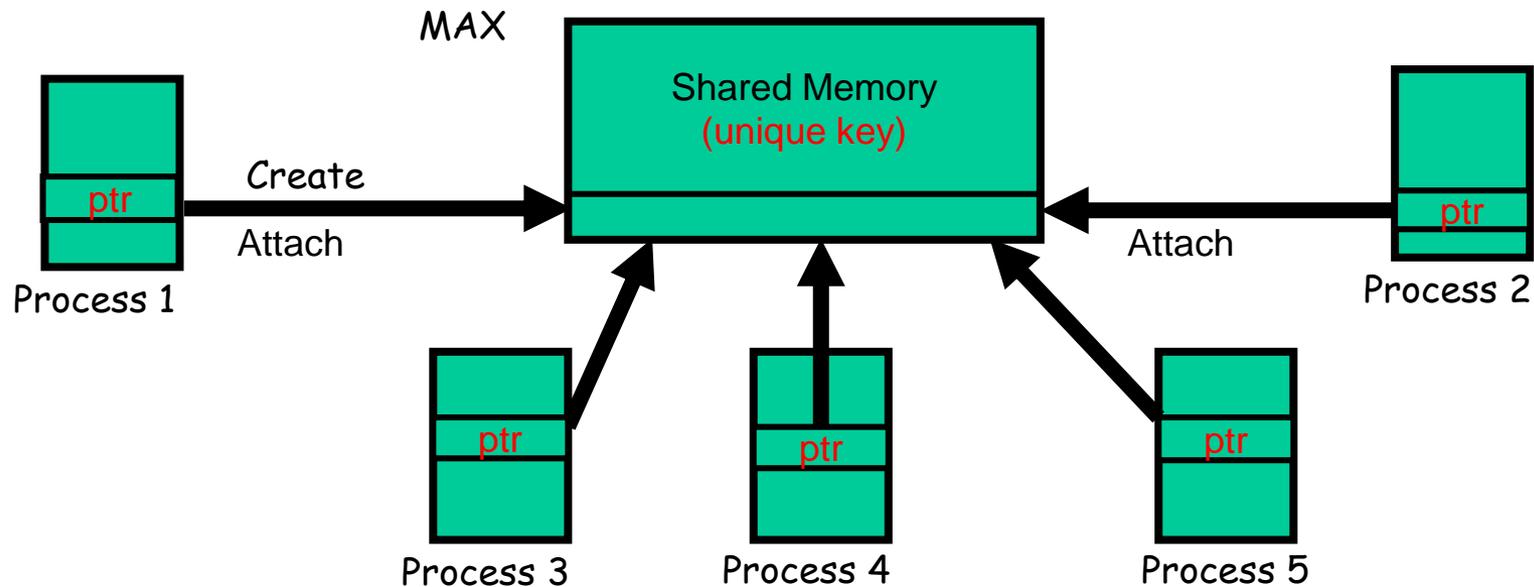
Example:

```
key_t key;
```

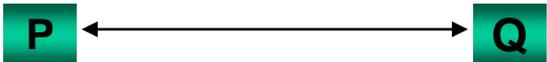
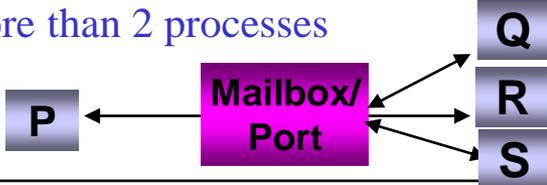
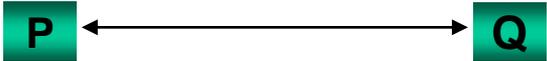
```
int shmid;
```

```
key = ftok("<somefile>", 'A');
```

```
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```



Implementation Questions

Questions	Direct Communications	Indirect Communications
How are links established?	Automatically established between every pair of processes that want to communicate 	Messages are sent to and received from mailboxes/ port. 
Can a link be associated with more than two processes?	No – a link is associated with exactly 2 processes 	Yes - a link may be associated with more than 2 processes 
How many links can there be between every pair?	Exactly one link exists between each pair of processes 	A number of links may exist between each pair of communicating processes, with each link corresponding to one mailbox 
Send Receive Primitives?	Send (P, message) // send a message to P Receive (Q, message) //receive message from Q	Send (A, message) //send a message to mailbox A Receive (A, message) //receive a message from mailbox A

Client & Server Processes Communication

- The server's process provides some services
 - It must be started first,
 - It waits for connections,
 - Must be secure, reliable and perform well,
 - It does not know who will connect with or when it will connect?
- The client's process connect to a running server's process
 - It knows who it is connecting to,
 - It initiates interaction,
 - Must be easy to use,
 - Should be portable to run on many platforms.
- Both client and server processes agree on how to exchange data, i.e. socket.
- Once a socket (**communication channel**) is created and a connection established.

Server's process:

- Listens for connection requests on a specified port,
- Accepts connection requests and gets a socket for each connection,
- Reads and writes data as required,
- Closes the connections,
- Deletes the sockets.

Client's process:

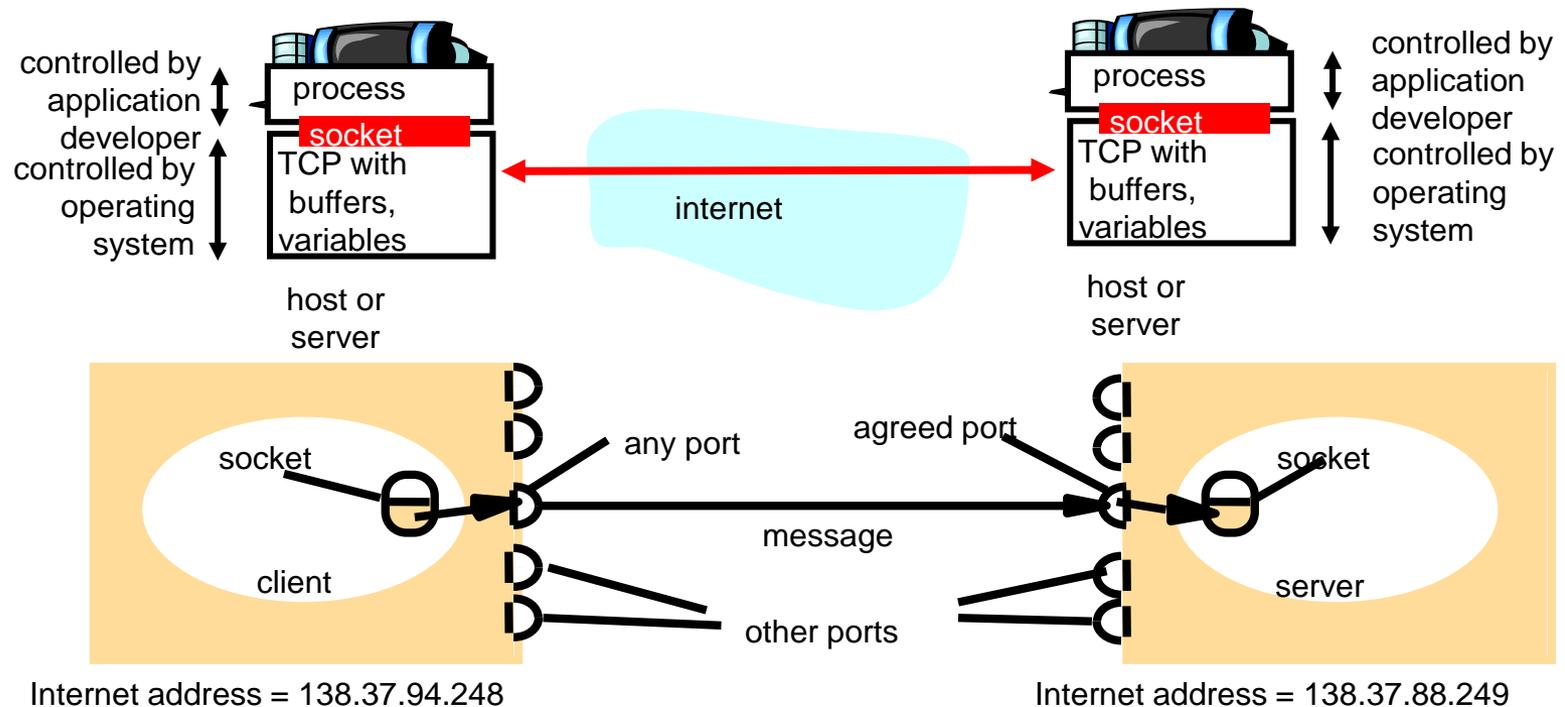
- Connects to a server on a specified IP and Port (**client's port is dynamically assigned**)
- Reads and writes data as necessary,
- Disconnects from the server,
- Deletes socket.

Client-Server Processes Communication

- Using Sockets
- Using Remote Procedure Calls
- Using Remote Method Invocation (**Java**)

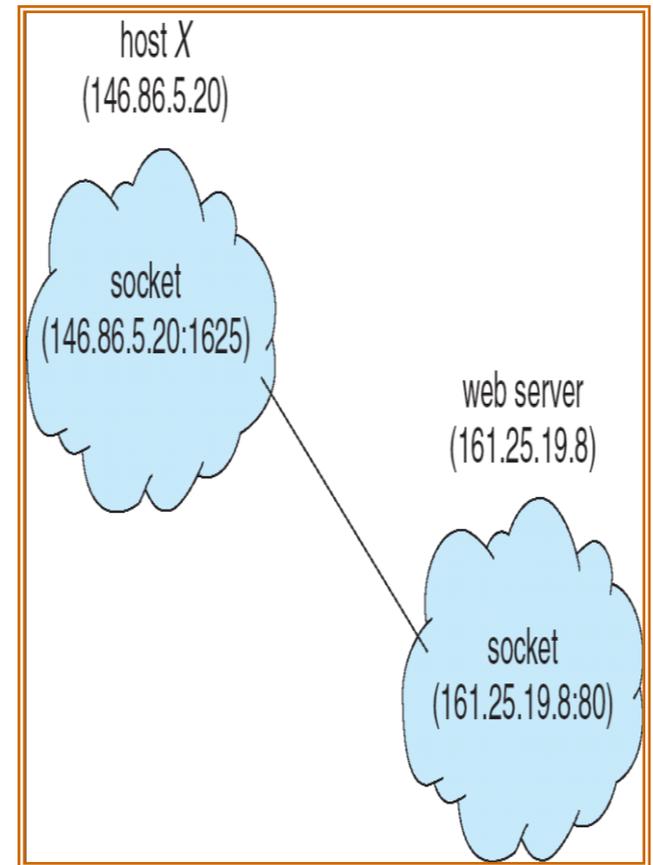
Socket Programming using TCP

- **Socket**: a door between a process and a Transmission Protocol (i.e. **UDP** or **TCP**):
- **UDP** (User Datagram Protocol) [does not guarantee a reliable transfer of bytes] it offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet.
- **TCP** guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

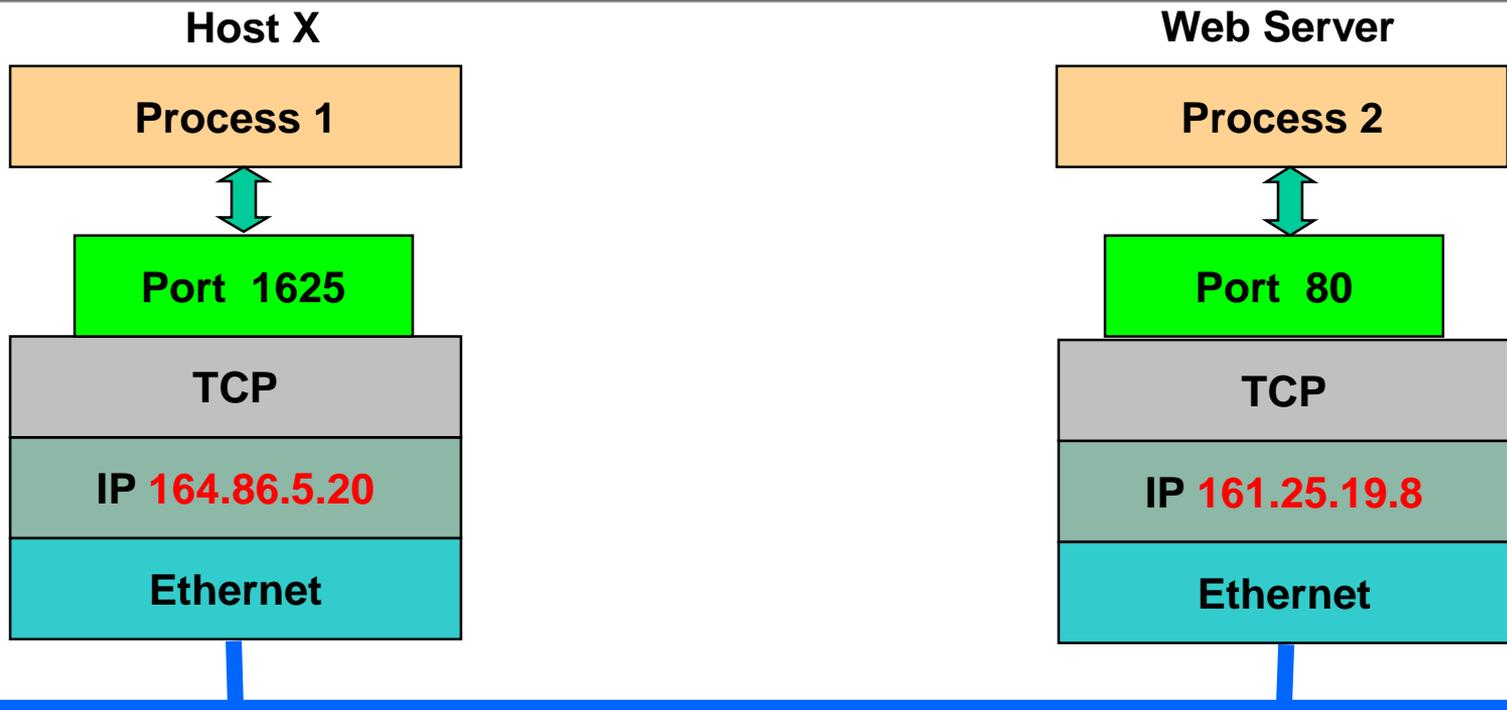


Socket “Communication Channel”

- A **socket** is identified by an **IP address** concatenated **with a port**.
 - The socket 161.25.19.8:**1625** refers to port 1625 on host 161.25.19.8
- The server waits for incoming client requests by listening to a specified port. **Usually many ports below 1024 are well known and used for standard services.** FTP [21], Telnet server [23], http server [80], SMTP [25], POP3 [110],
- When a client initiates a request for connection, it is automatically assigned a port (>1024) by the host computer.



Socket “Communication Channel”



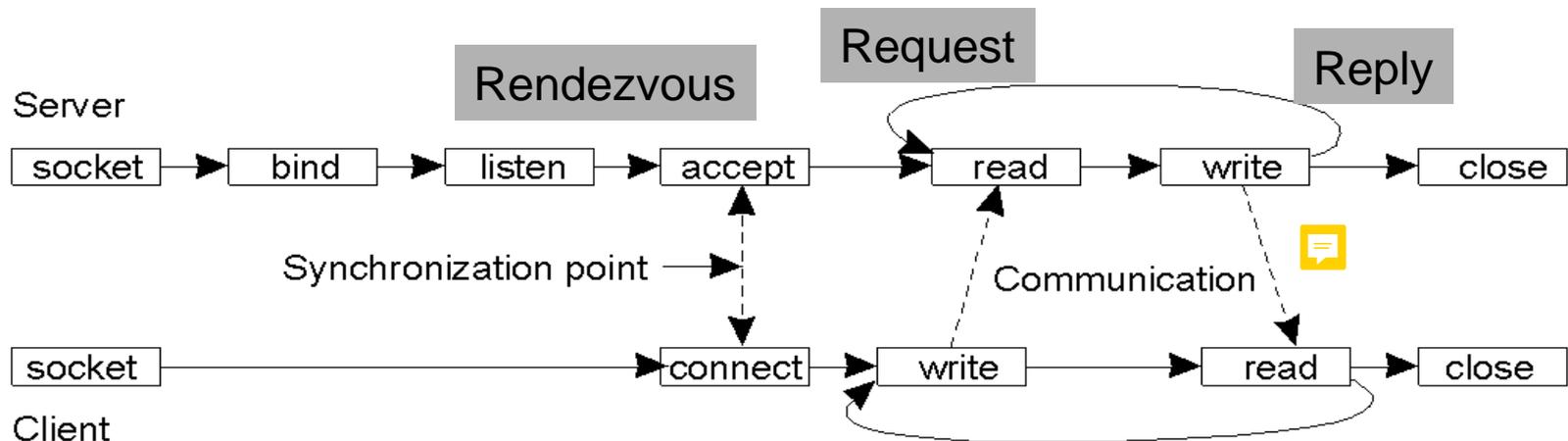
- A client on host X [164.86.5.20] wants to establish a connection with the Web server [161.25.19.8] which is listening to port 80.
- The host X may be assigned port 1625.
- The connection will consist of a pair of sockets:
- The socket 164.86.5.20:1625 refers to port 1625 on host X and 161.25.19.8:80 on the Web server.

- **Step1:** Create a ServerSocket Object
 - `ServerSocket s = new ServerSocket(port)`
- **Step2:** Create a Socket and Wait for a Connection
 - `Socket connect = s.accept()`
- **Step3:** Associate Input and Output Stream with the Socket
 - `connect.getInputStream`
 - `connect.getOutputStream`
- **Step4:** Process Connection
- **Step5:** Close Connection

- **Step1:** Create a Socket to make connection
 - Socket connect = new Socket(Server IP, port)
- **Step2:** Associate Input and Output Stream with the Socket
 - connect.getInputStream
 - connect.getOutputStream
- **Step3:** Process Connection
- **Step4:** Close Connection

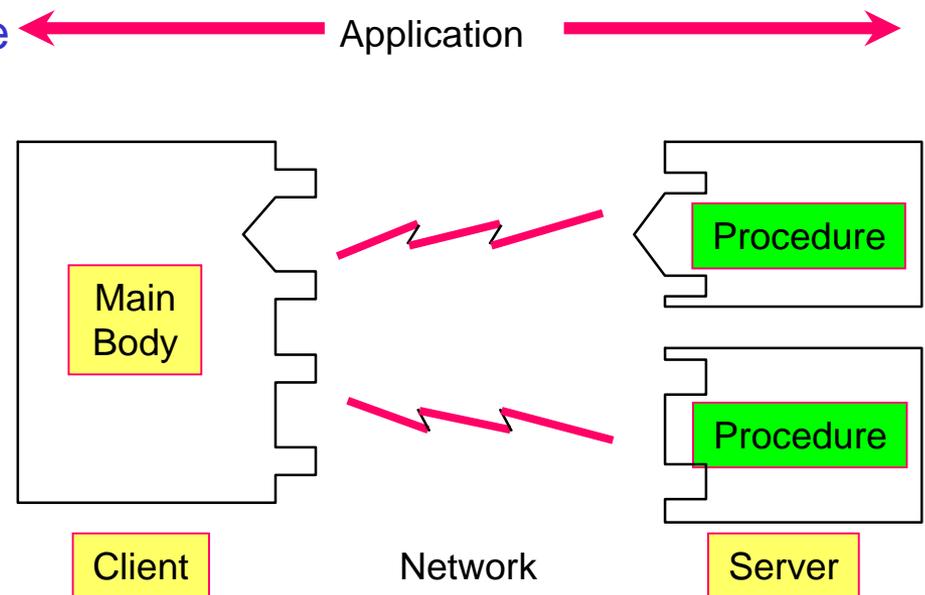
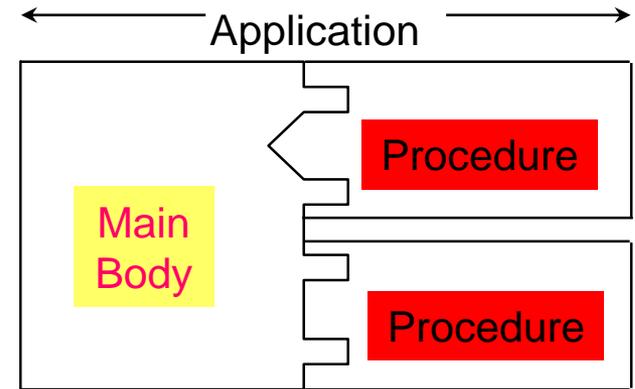
Socket Communication Steps

- **Socket system call:** Create a socket
- **Bind system call:** A name and an address are bounded to a socket.
- **Listen system call:** The server must listen to its socket, by telling the kernel that it is ready to accept connections from clients.
- **Accept system call:** The server can accept or select connections from clients.
- **Connect system Call:** The client connects to the socket. It needs to provide the socket address by which it can reach the server.
- **Read/Write system call:** Client and server communicate through read/write operations on their respective sockets.
- **Close system call:** Terminates a connection and destroy the associated socket.



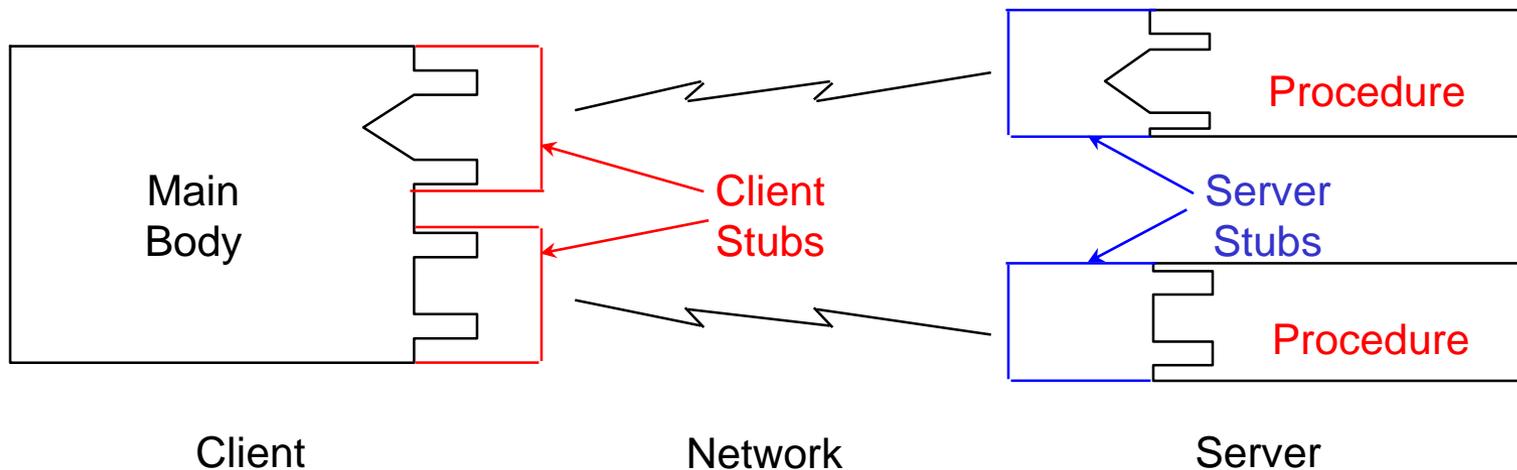
Remote Procedure Call (RPC)

- Remote Procedure Calls are much more complex than a local subroutine or a method call.
- Subroutine and co-routine calls are generally made within an application or between co-applications that run on a the same system.
- RPC's, on the other hand, are made between systems that are interconnected by a network.
- The server also must handle simultaneous requests from many clients.
- This also implies a need for synchronization among requests.



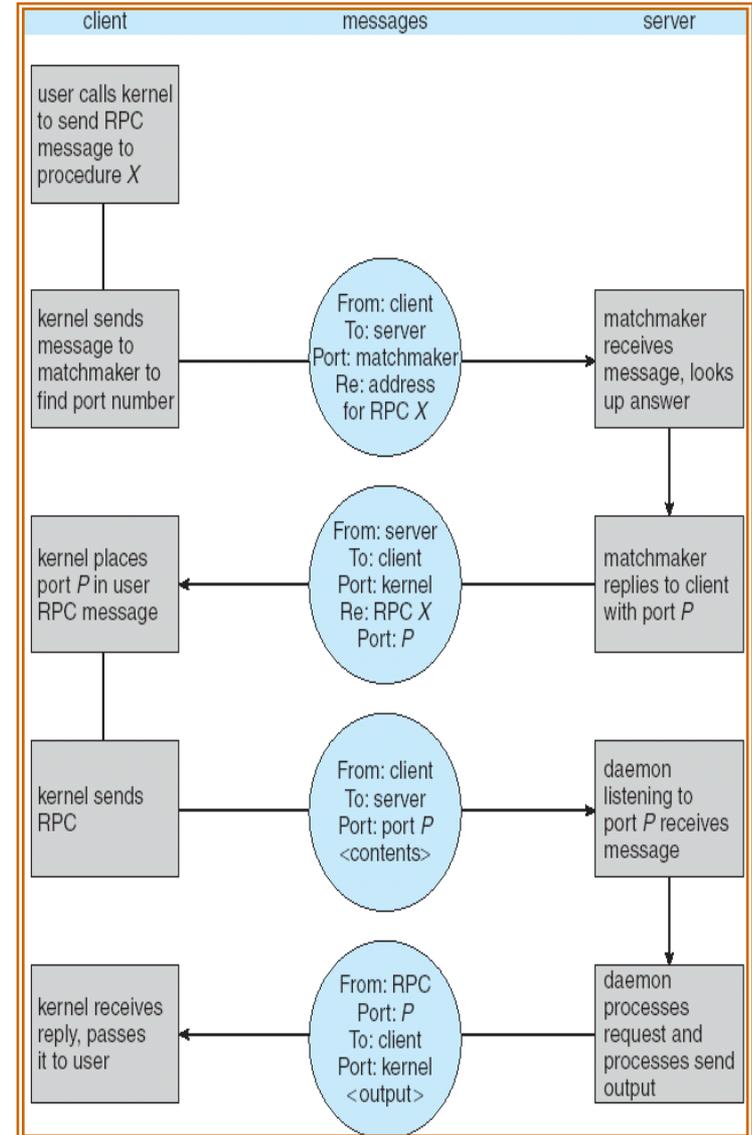
Remote Procedure Calls: Stubs

- **Client** makes procedure call (like a local procedure call) to the **client stub**.
- A **stub** is a piece of code that converts parameters passed between the client and the server during a remote procedure call (**RPC**).
- **Client Stub** locates the server and the port on the server.
- **Client Stubs** take care of packaging arguments and sending messages.
- Packaging the parameters is called “**marshalling**”.
- The server-**stub** receives the message, **unpacks** the marshaled parameters, **invokes the procedure** on the server and then **returns the value**.



Steps of a Remote Procedure Call

1. Client procedure calls client's **stub** in normal way.
2. Client's **stub** builds/**packs** a message and calls its local OS.
3. Client's OS sends the message to the remote OS.
4. The server's OS gives the message to its **stub**.
5. Server's **stub** **unpacks** the parameters, and calls the server procedure.
6. The server procedure does the work, and returns the result to the **stub**.
7. The server's **stub** **packs** it in message, and calls its local OS.
8. The server's OS sends message to client's OS.
9. Client's OS gives the message to the client **stub**.
10. Client's **stub** **unpacks** the result, and returns it to client.

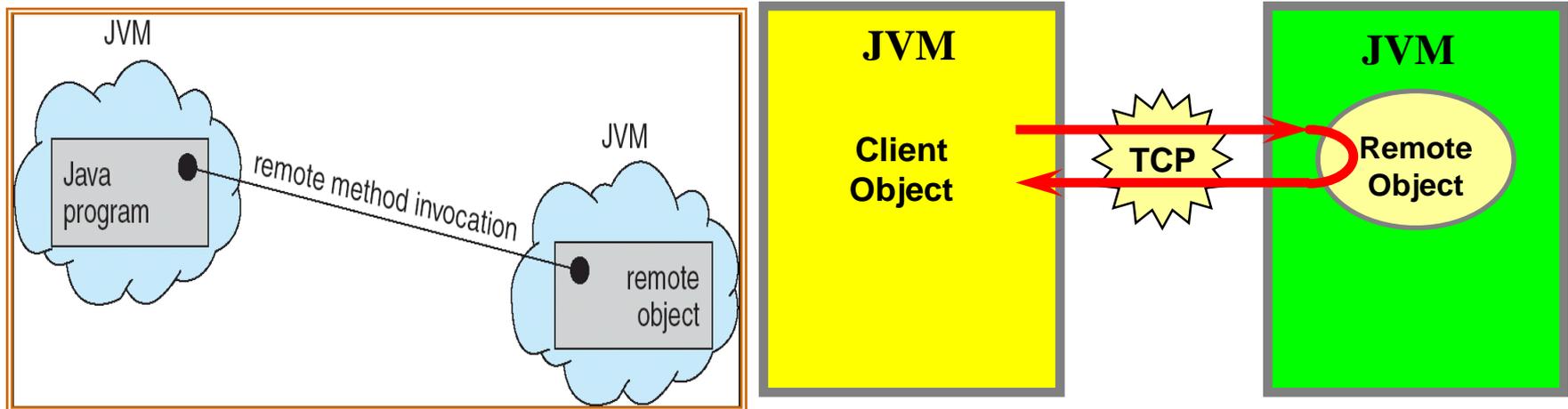


Marshalling

- **Problem:** Different machines have different data formats.
 - **Intel:** little endian, **SPARC:** big endian
- **Solution:** Use a standard machine independent representation.
 - Example: e**X**ternal **D**ata **R**epresentation (**XDR**)
 - **XDR** is a data abstraction needed for machine independent communication.
- **Marshalling:** Transform parameters/results into a byte stream.

Java RMI

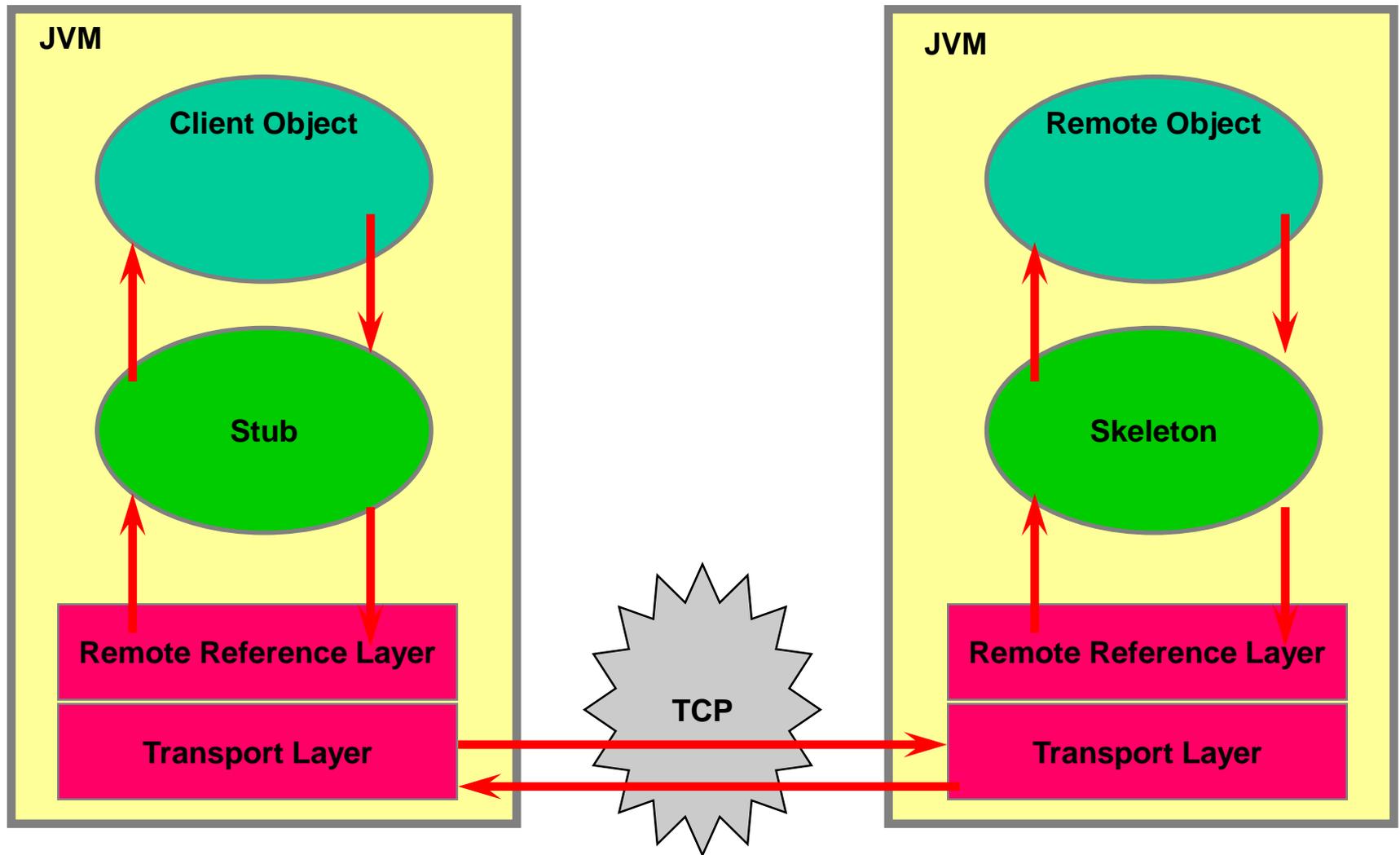
- RMI allows an object running in one JVM to invoke methods on an object running in another JVM.
- RMI can be used to allow object's methods to invoke other object's methods running in the same or remote machine.
- The RMI mechanism is basically an object-oriented RPC mechanism.
- Objects can be passed as arguments and returned as results
- Any Java object can be passed during invocation including primitive types, core classes, user-defined classes and Java-Beans
- Syntax of RMI is same as the local method invocations
- RMI operates only in Java-Java domain.



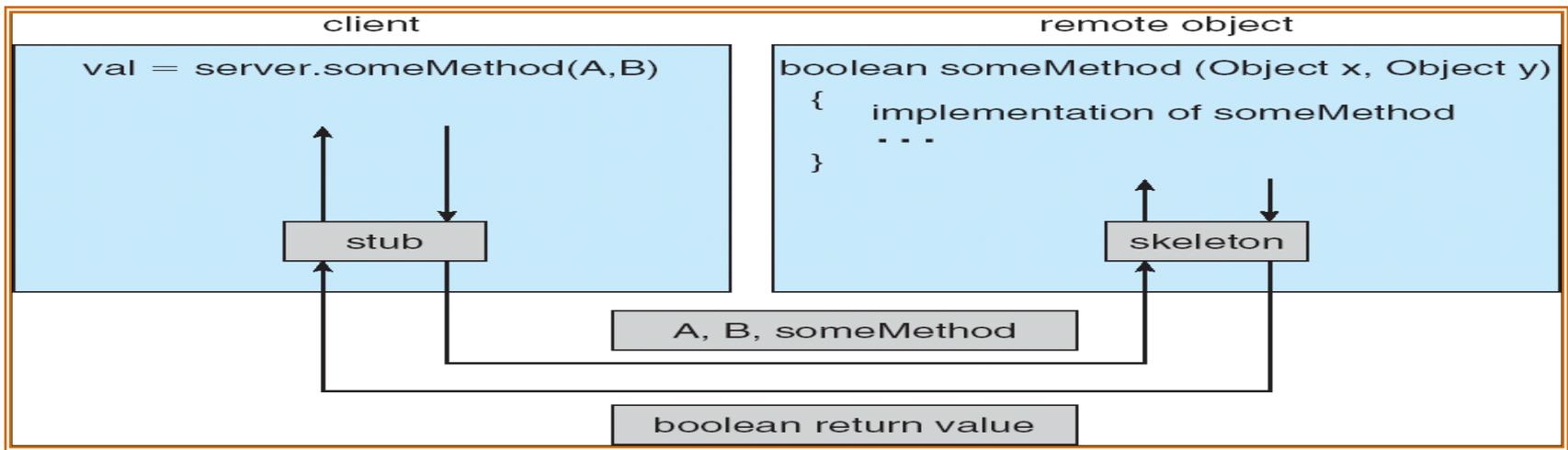
RMI Registry

- The server's object must register itself under some name where it can be reached.
- Under RMI, this is done with the RMI Registry, a separate process that must be running, usually on the server machine.
- Once an object has been registered, any other objects can use the Object Registry to obtain access to its methods remotely using the name of the object.

RMI Layers



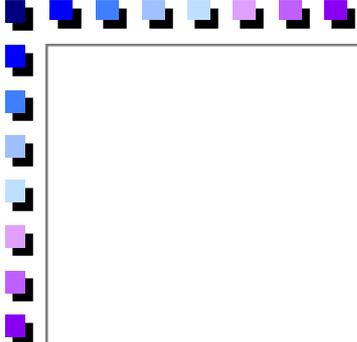
Marshalling Parameters



- **Stub:** is responsible of creating a parcel consisting of the name of the method to be invoked on the server and the marshaled parameters for the method.
- **Stub** locates the server, sends the parcel to the **skeleton** of the server.
- The **skeleton** is responsible for **un-marshalling the parameters** and invoking the desired method on the server.
- The **skeleton** then marshals the return value into a parcel and send it to the **Stub**.
- The **stub** receives this message, unpacks the marshaled return value and passes it to the client.

RPC/RMI Differences

- Client /Server?
 - **RPC** is typical client/server application
 - Server defines procedure.
 - Client invokes it.
 - But **RMI** provides more flexible way.
 - Dynamic class loading
- Safe & Security?
 - **RPC** Security mechanism
 - Operating System based
 - **RMI** Security mechanism
 - Uses Java's built in Security features
 - A security manager has to be installed before RMI can be used
- Object -Oriented?
 - **RPC** is not Object-Oriented.
 - **RMI** is Object-Oriented.
- Language independent?
 - **RPC** was designed for a heterogeneous environment. Use e**X**ternal **D**ata **R**epresentation (**XDR**) protocol to standardizes the representation of data.
 - **RMI** was designed for **JAVA to JAVA** environment.



The End!!

Thank you

Any Questions?

