Parallel & Distributed Computing (WQD7008)

Week 3

2019/2020 Semester 1

Dr. Hamid Tahaei

Process

- Process in a general term means execution of data and instruction.
- An instance of a computer program in execution.
- A process is often defined as a program in execution.
- A process is a program that is currently being executed on one of the operating **system's virtual processors**.
- To execute a program, an operating system creates a number of virtual processors, each of which runs a different program.
- **A Process table:** to keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information. privileges, etc.
- Each program gets executed on a virtual processor.
- Multiple processes may be concurrently sharing the same CPU and other hardware resources.
- A process consists of an execution environment together with one or more threads.

Execution Environment:

- An *execution environment* is the unit of resource management: a collection of local kernel managed.

An execution environment primarily consists of:

- An address space.
- thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets).
- higher-level resources such as open files and windows.

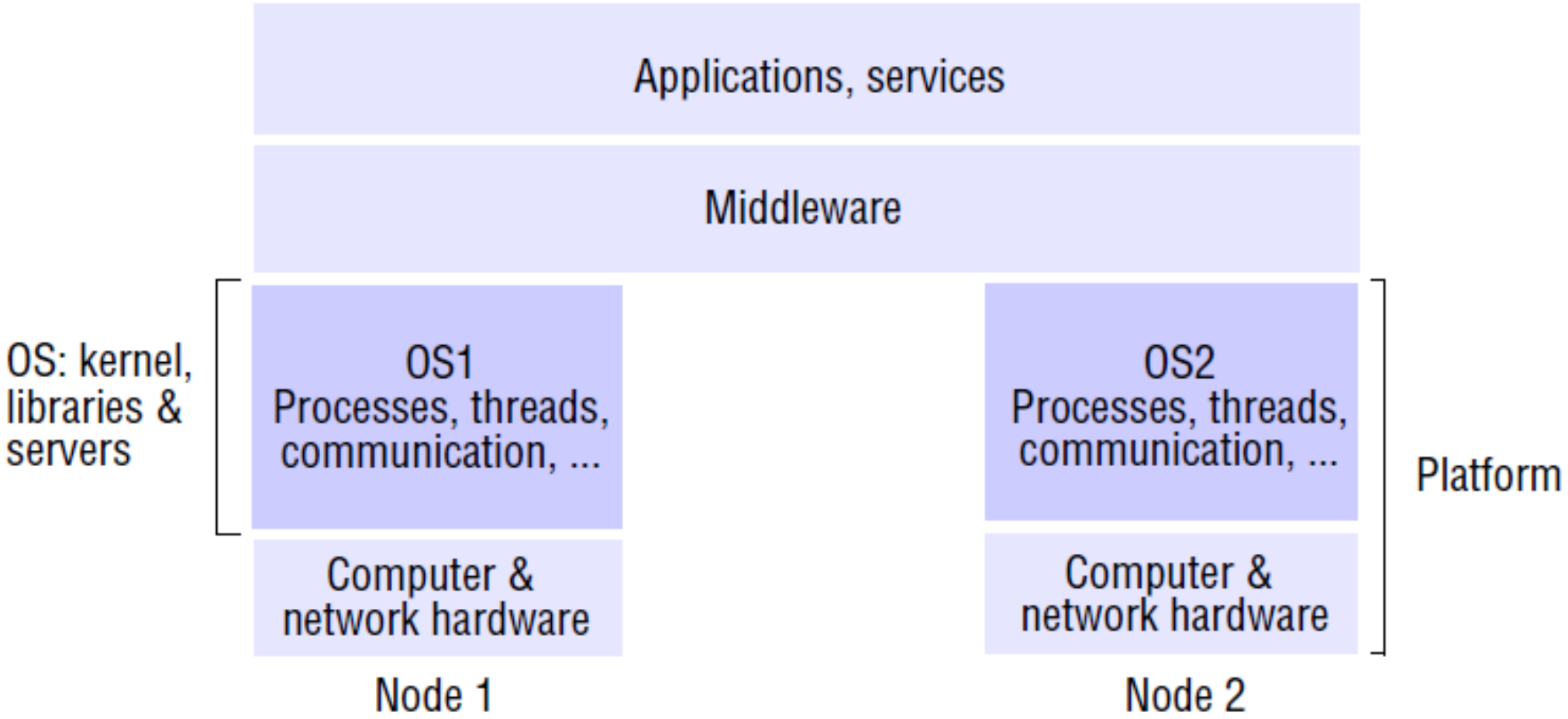
Execution environments are normally expensive to create and manage, but several threads can share them – that is, they can share all resources accessible within them.

Address Space:

- A unit of management of a process's virtual memory.

Cost of process and processing (concurrency transparency)

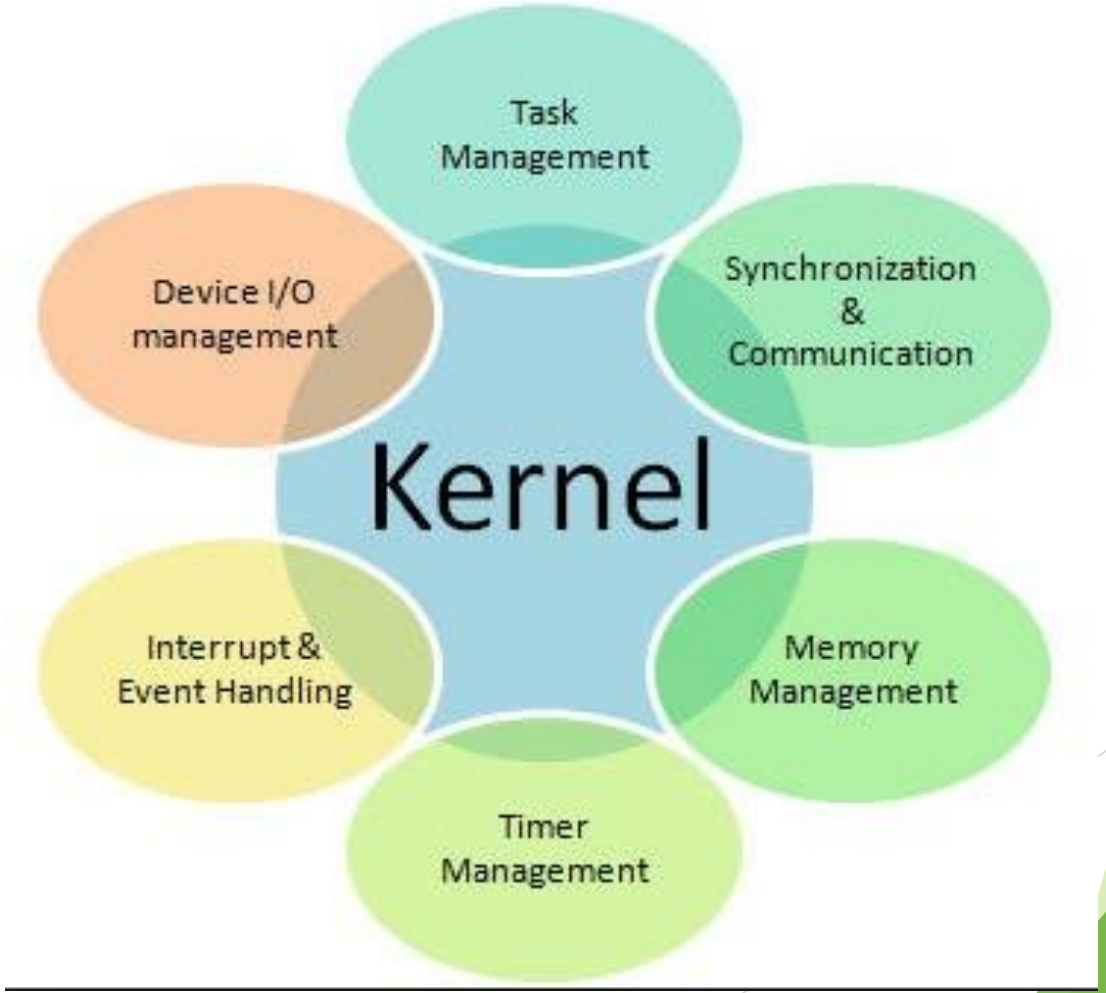
- Address space: Each time a process is created, the operating system must create a complete independent address space.
- Allocation: initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data.
- Switching the CPU between two processes.
- Modify registers of the **memory management unit** (MMU): the operating system has to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the **translation lookaside buffer** (TLB).
- swap processes between main memory and disk: if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.


Applications, services
Middleware
OS: kernel, libraries & servers
OS1
Processes, threads, communication, ...
OS2
Processes, threads, communication, ...
Platform
Computer & network hardware
Computer & network hardware
Node 1
Node 2

- Kernels and server processes are the components that manage resources and present clients with an interface to the resources.

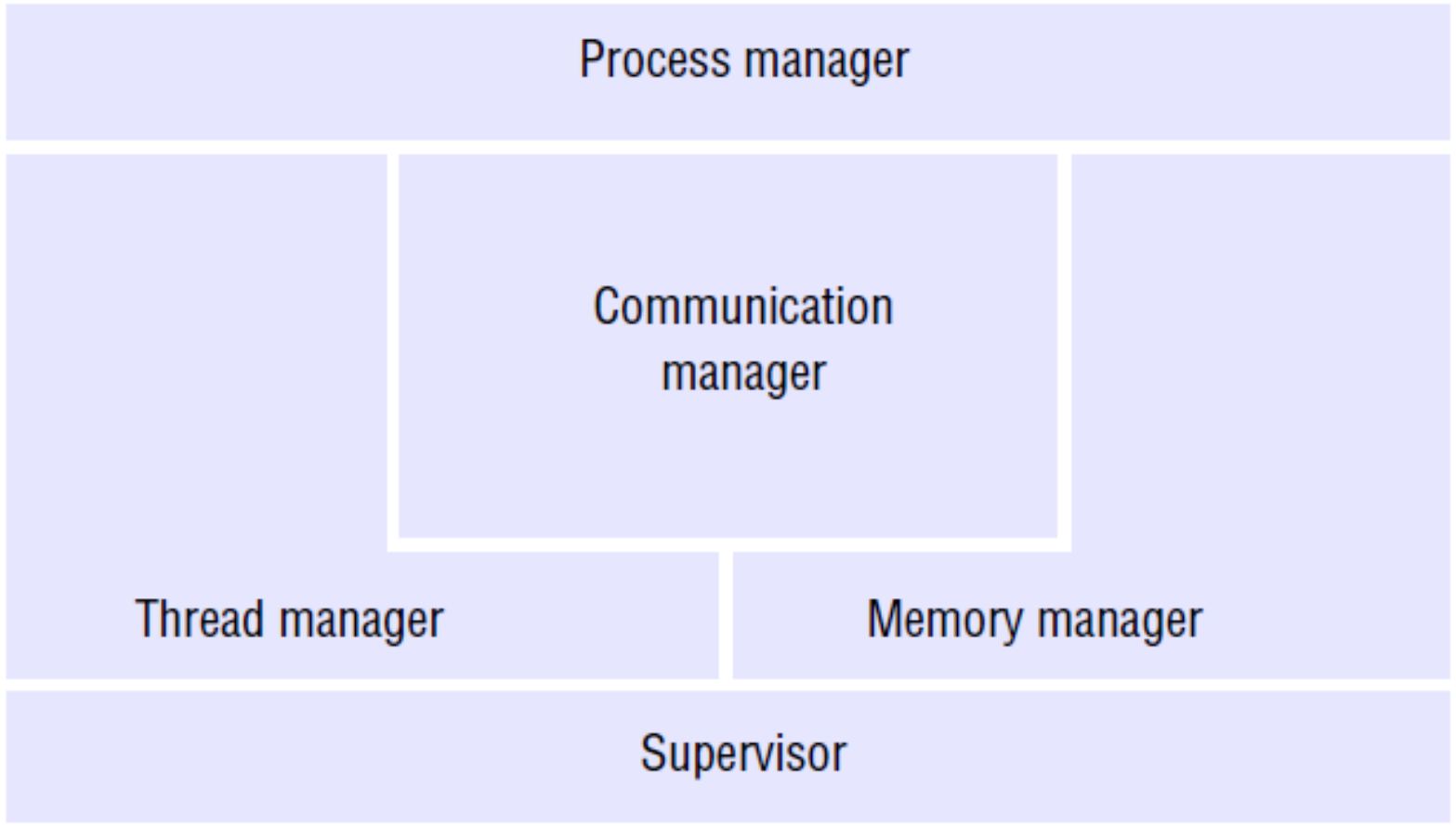
A combination of libraries, kernels and servers may be called upon to perform the following invocation related tasks:

- *Communication*
- *Scheduling*



The core OS components and their responsibilities are:

- *Process manager*: Creation of and operations upon processes.
- *Thread manager*: Thread creation, synchronization and scheduling.
- *Communication manager*: Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication.
- *Memory manager*: Management of physical and virtual memory.
- *Supervisor*:Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.



OS functionality

Thread

- A *thread* is the operating system abstraction of an activity.
- Like a process, a thread executes its own piece of code, independently from other threads.
- Threads can be created and destroyed dynamically, as needed.
- The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors.
- This can be particularly helpful within servers, where concurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

- Many older operating systems allow only one thread per process.
- Like a process, a thread executes its own piece of code, independently from other threads.
- ~~In contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation.~~
- Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads. In particular, a thread context often consists of nothing more than the CPU context, along with some other information for thread management.
- For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution.
- Information that is not strictly necessary to manage multiple threads is generally ignored. For this reason, protecting data against inappropriate access by threads within a single process is left entirely to application developers.

If two applications are run on a computer (MS Word, MS Access),

- Two processes are created.
- Multitasking of two or more processes is known as ***process-based multitasking***.
- Multitasking of two or more threads is known as ***thread-based multitasking***.
- The concept of multithreading in a programming language refers to thread-based multitasking.
- **Process-based multitasking** is totally **controlled by** the **operating system**.
- But **thread-based multitasking** can be controlled by **the programmer** to some extent in a program.

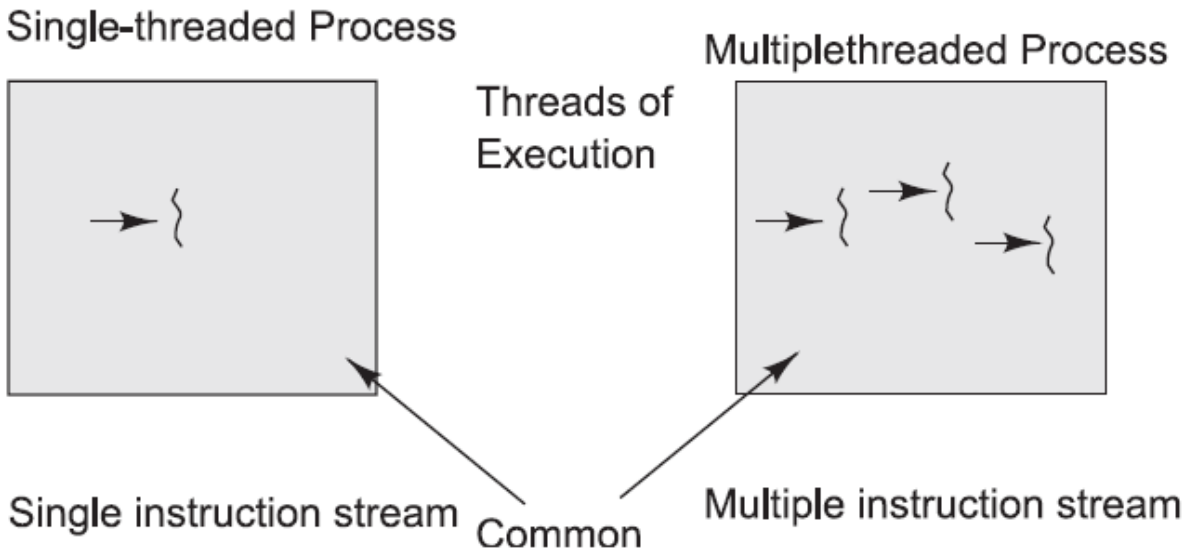


Fig. 14.1 A process containing single and multiple threads

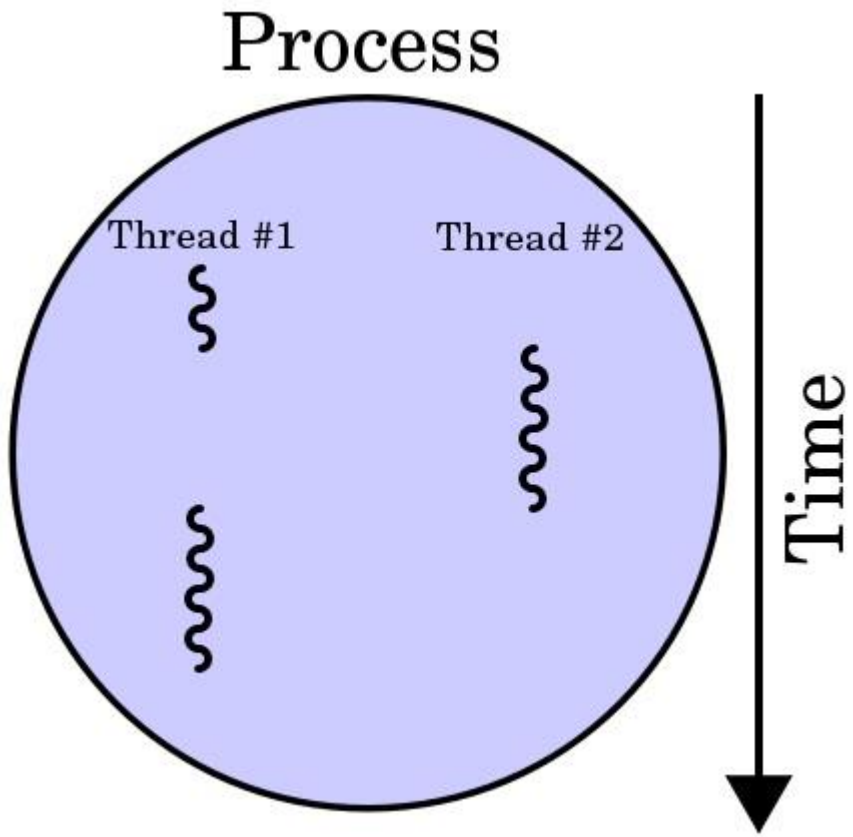
Process vs Thread

Process

- Typically independent.
- Has considerably more state information than thread.
- Separate address spaces.
- Iinteract only through system IPC.

Thread

- Subsets of a process.
- Multiple thread within a process share process state, memory, etc.
- Thread share their address space.



Threads are cheaper to create and manage than processes, and resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment.

Thread Implementation

- Threads are often provided in the form of a thread package.
- Thread package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables.
- Two approaches to implement a thread package:
  - to construct a thread library that is executed entirely in user mode.
  - to have the kernel be aware of threads and schedule them.

Advantage of the first approach:

- Cheap to create and destroy threads
  - Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack.
- Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used.
- Both operations are cheap.

Advantage of the first approach **(Continue)**

- Switching thread context can often be done in just a few instructions.
- only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched.
- There is no need to change memory maps, flush the translation lookaside buffer (TLB), do CPU accounting, and so on. Switching thread context is done when two threads need to synchronize, for example, when entering a section of shared data.

Major drawback of user-level threads

- invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.
- Threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime. For such applications, user-level threads are of no help.

Thread Implementation (Continued)

Second approach to implement a thread package:

Implementing threads in the operating system's kernel.

- every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel, requiring a system call.

A high price to pay

- Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

Thread Implementation (Continued)

Hybrid approach form of user-level and kernel-level threads:

- It generally referred to as **lightweight** processes (LWP).
- An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.
- The system offers a user-level thread package. Offering applications the usual operations for creating and destroying threads.
- The package provides facilities for thread synchronization. such as mutexes and condition variables.
- The important issue is that the thread package is implemented entirely in user space. In other words. all operations on threads are carried out without intervention of the kernel.
- The thread package can be shared by multiple LWPs.
- Each LWP can be running its own (user-level) thread.
- Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP. Assigning a thread to an LWP is normally implicit and hidden from the programmer.

Combination of (user-level) threads and LWPs:

- The thread package has a single routine to schedule the next thread.
- When creating an LWP (which is done by means of a system call), the LWP is given its own stack. and is instructed to execute the scheduling routine in search of a thread to execute.
- The thread table, which is used to keep track of the current set of threads, is thus shared by the LWPs. Protecting this table to guarantee mutually exclusive access is done by means of mutexes that are implemented entirely in user space.
- Synchronization between LWPs does not require any kernel support.
- When an LWP finds a runnable thread, it switches context to that thread. Meanwhile, other LWPs may be looking for other runnable threads as well.
- If thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the scheduling routine. 'When another runnable thread has been found.
- the context switch is implemented completely in user space and appears to the LWP as normal program code.

Threads Programming

- Threads programming is concurrent programming.
- The C Threads package known as *pthreads*, has been widely adopted with the standard IEEE 1003.1c-1995.

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Sets and returns the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Changes the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(long millisecs)
Causes the thread to enter the *SUSPENDED* state for the specified time.

yield()
Causes the thread to enter the *READY* state and invokes the scheduler.

destroy()
Destroys the thread.

Thread lifetime

- Threads programming is concurrent programming.
- The C Threads package known as *pthreads*, has been widely adopted with the standard IEEE 1003.1c-1995.

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Sets and returns the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Changes the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(long millisecs)
Causes the thread to enter the *SUSPENDED* state for the specified time.

yield()
Causes the thread to enter the *READY* state and invokes the scheduler.

destroy()
Destroys the thread.

Thread synchronization

thread.join(long millisecs)

Blocks the calling thread for up to the specified time or until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, the thread is interrupted or the specified time has elapsed.

object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

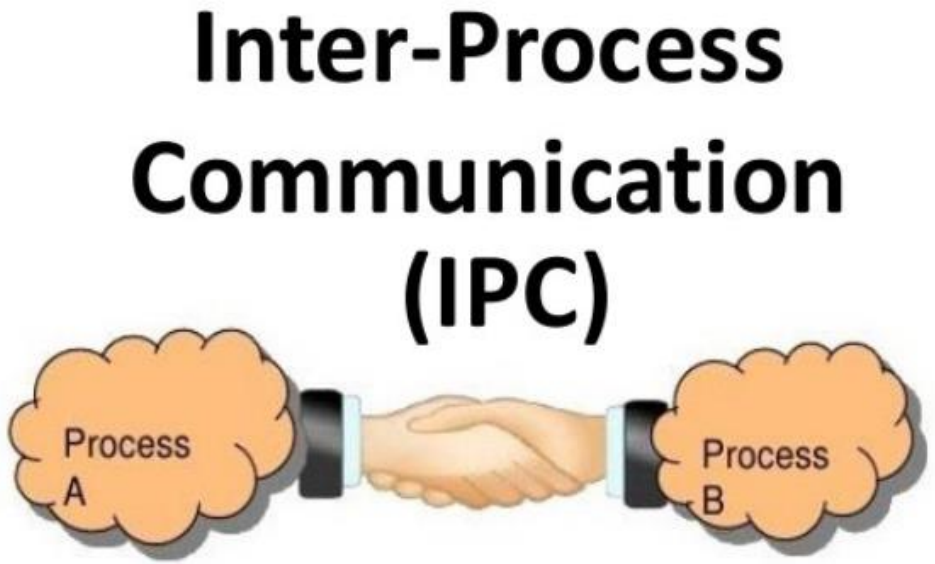
Inter-process Communication (IPC)

A Process:

- Independent
  - Not affected by the execution of the other processes.
- Cooperative
  - The processes cooperate with each other.
- Can affect or be affected by other processes, including sharing data

IPC: Mechanisms to transfer data between processes

It allows the processes communicate to each other and synchronize their action.



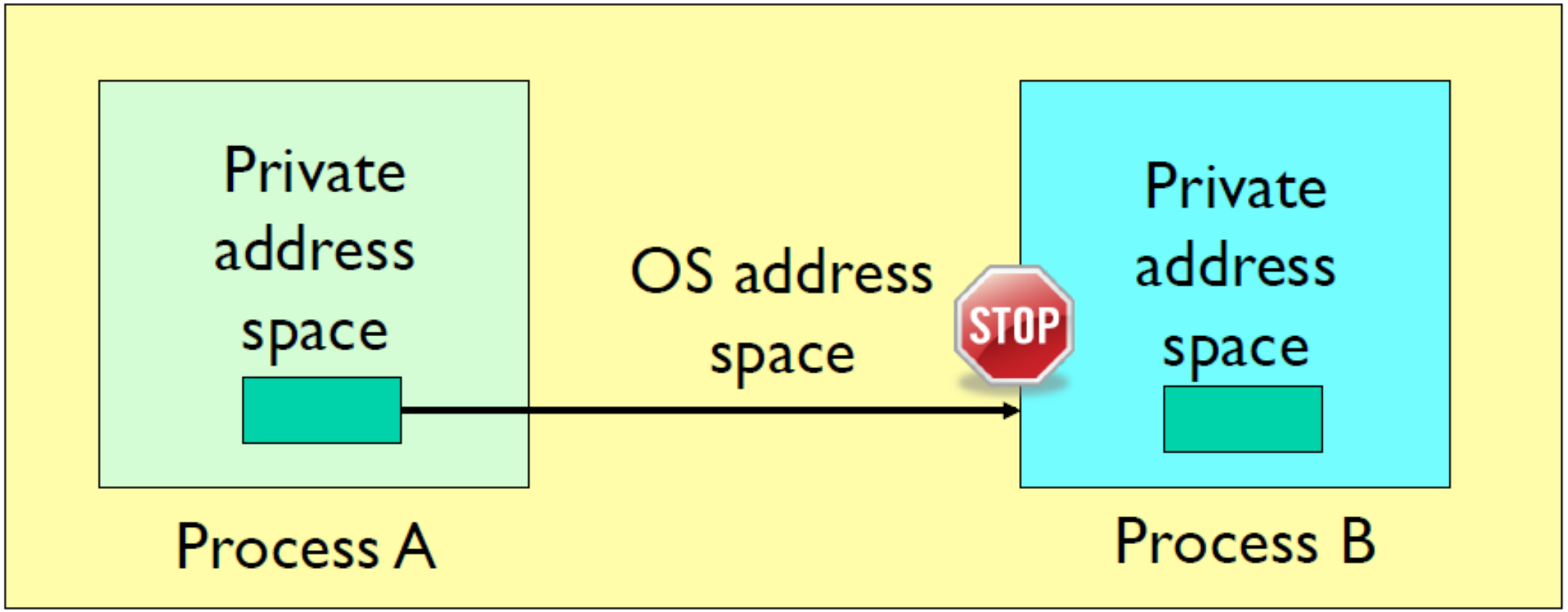
Why IPC

Why is it needed?

Not all important procedures can be easily built in a single process

Benefits:

- Information sharing
  - Will be there from one process to another process. Suppose maybe present in your system or a other system. They will be communicating with the help of IPC (within or between system).
- Resource sharing
- Computation speed up
  - If you use limited resources, those resources can be shared through other and increase the speed or computation.
- Synchronization
  - Synchronizing information among processes and provides:
- Modularity
- convenience

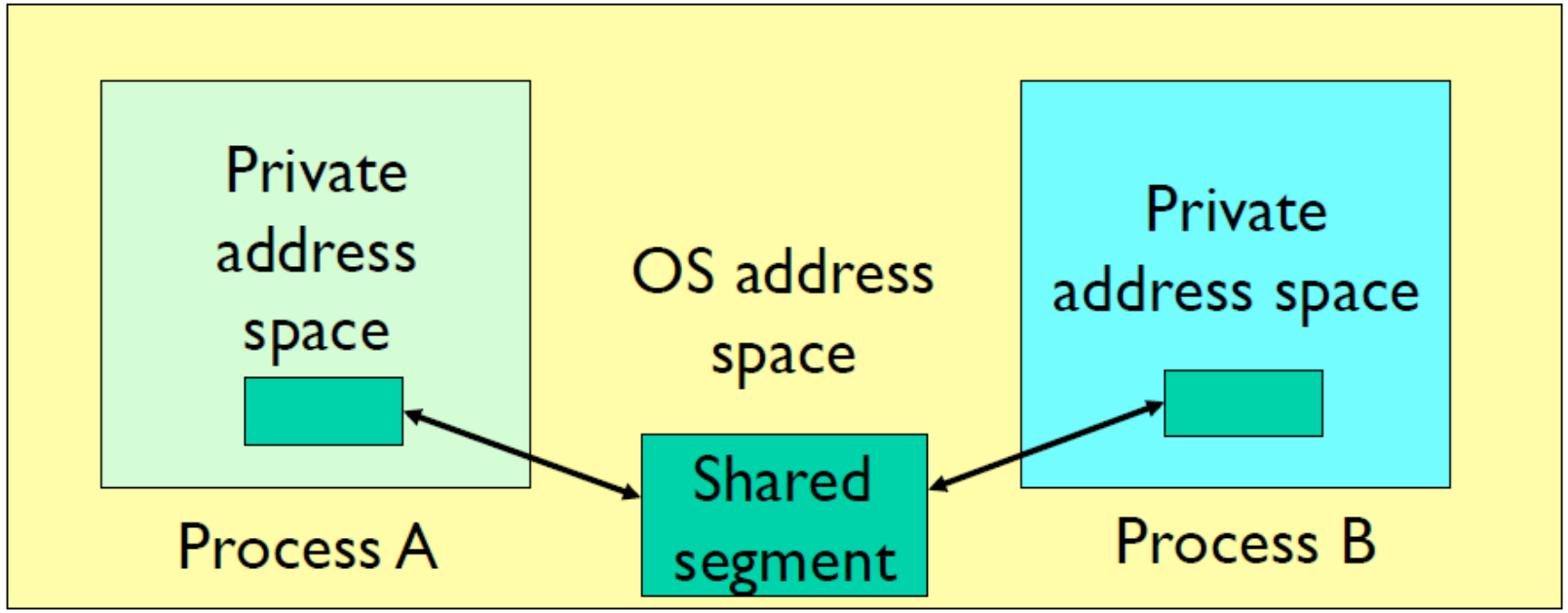


- Each process has a private address space
- No process can write to another process's space
- How can we get data from process A to process B?

IPC Mechanism

Processes communicate with each other using two ways: **Shared Memory & Message Passing**

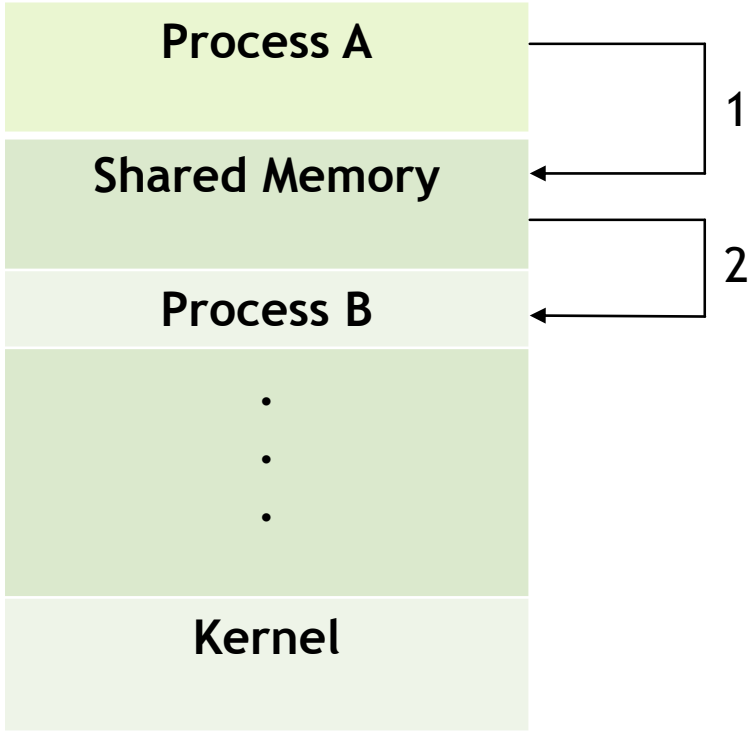
- Shared memory
  - Processes share the same segment of memory directly.
  - Memory is persistent beyond the lifetime of the creating or modifying processes (until deleted)
  - Mutual exclusion must be provided by processes using the shared memory

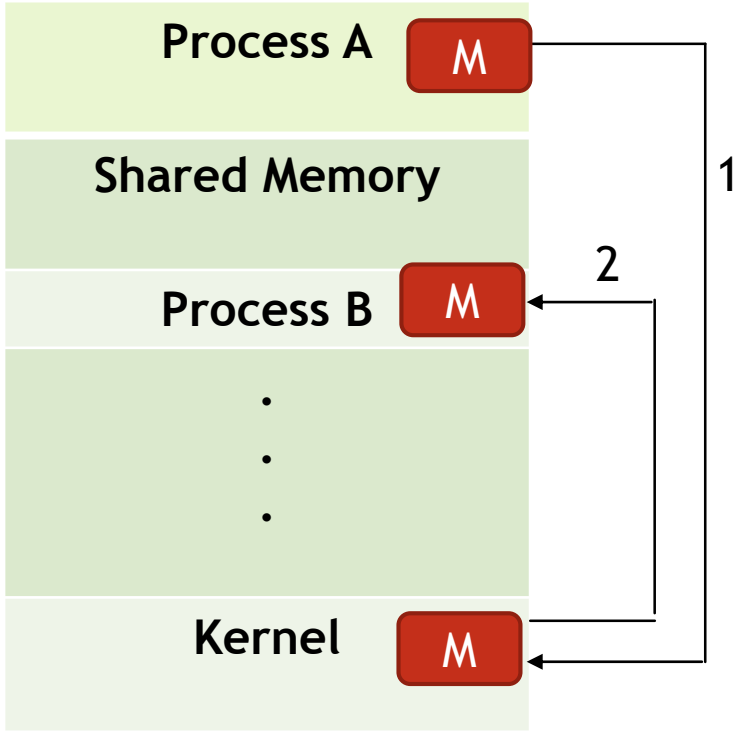


  - Processes request the segment
  - OS maintains the segment
  - Processes can attach/detach the segment

Shared memory (Continued)

Process A generates information about certain resources and keeps it as a record the a shared memory.





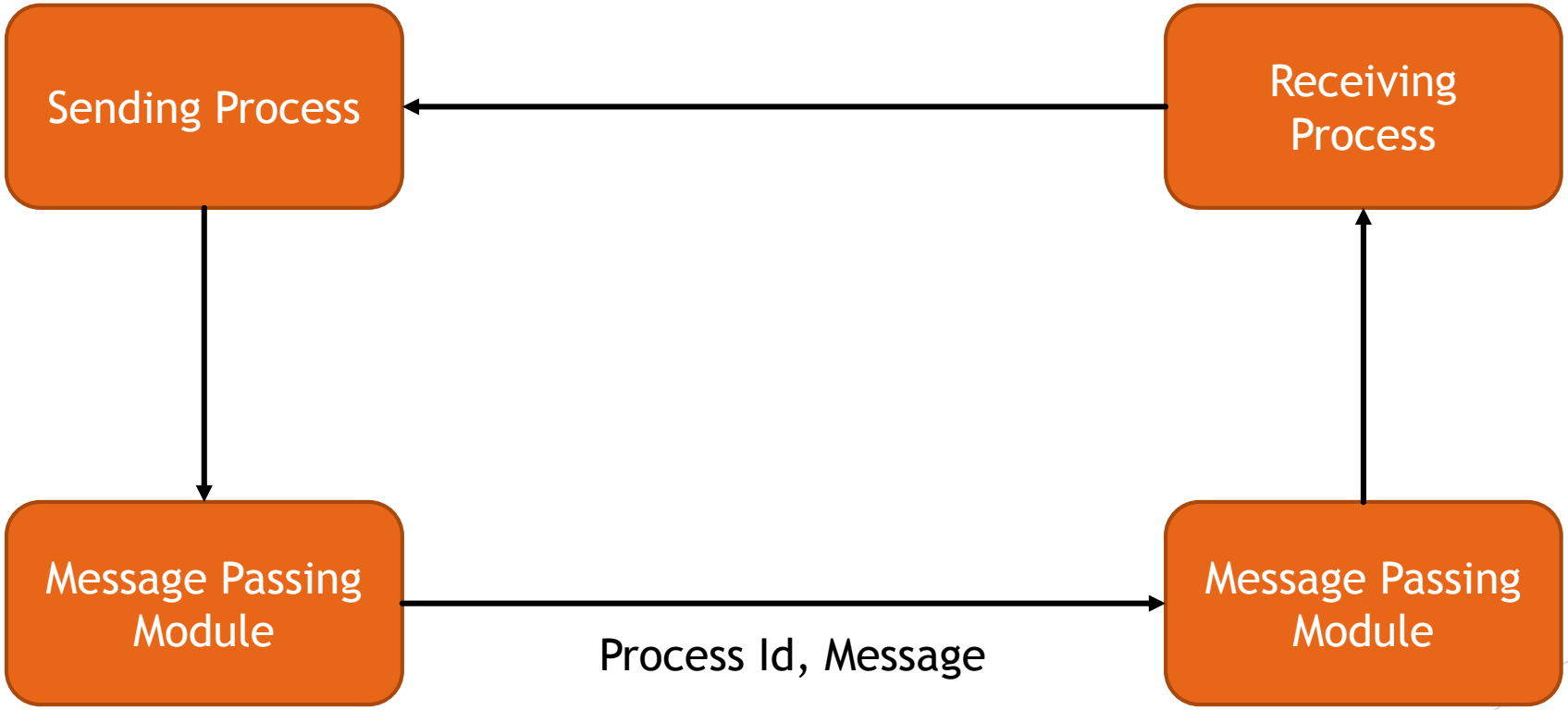
When process B, needs to use some resources (data & information) from A, it checks the record in the shared memory and uses that information.

Message Passing:

No use of shared memory.

If two process want to communicate:

- Stablish a communication link.
- Start exchanging message using basic primitives.
  - Send() : (message, destination)
  - Receive() (message, host)
- Direct OR indirect.
- Symmetric OR Asymmetric
- Automatic OR Exclusive buffering


Sending Process
Receiving Process
Message Passing Module
Message Passing Module
Process Id, Message

IPC Method

File

- Use to store the resources (information)
- Those resources will be saved and available after the completion of the program.

Signal

- Use in UNIX operating system.
- Passing a signal to a processor. i.e. interrupting a process.

Socket

- Provides point to point communication & two way communication between two processes.

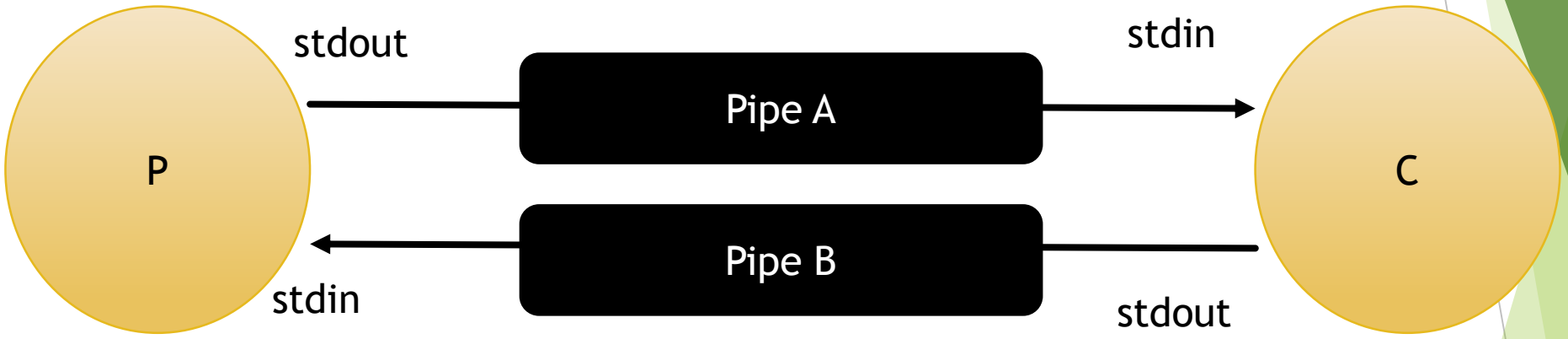
Message Queue

- It provides synchronized or communication protocol
- Messages placed on the queue. The receiver or sender retrieve the message from the queue.
- No need for sender or receiver to interact at the same time.

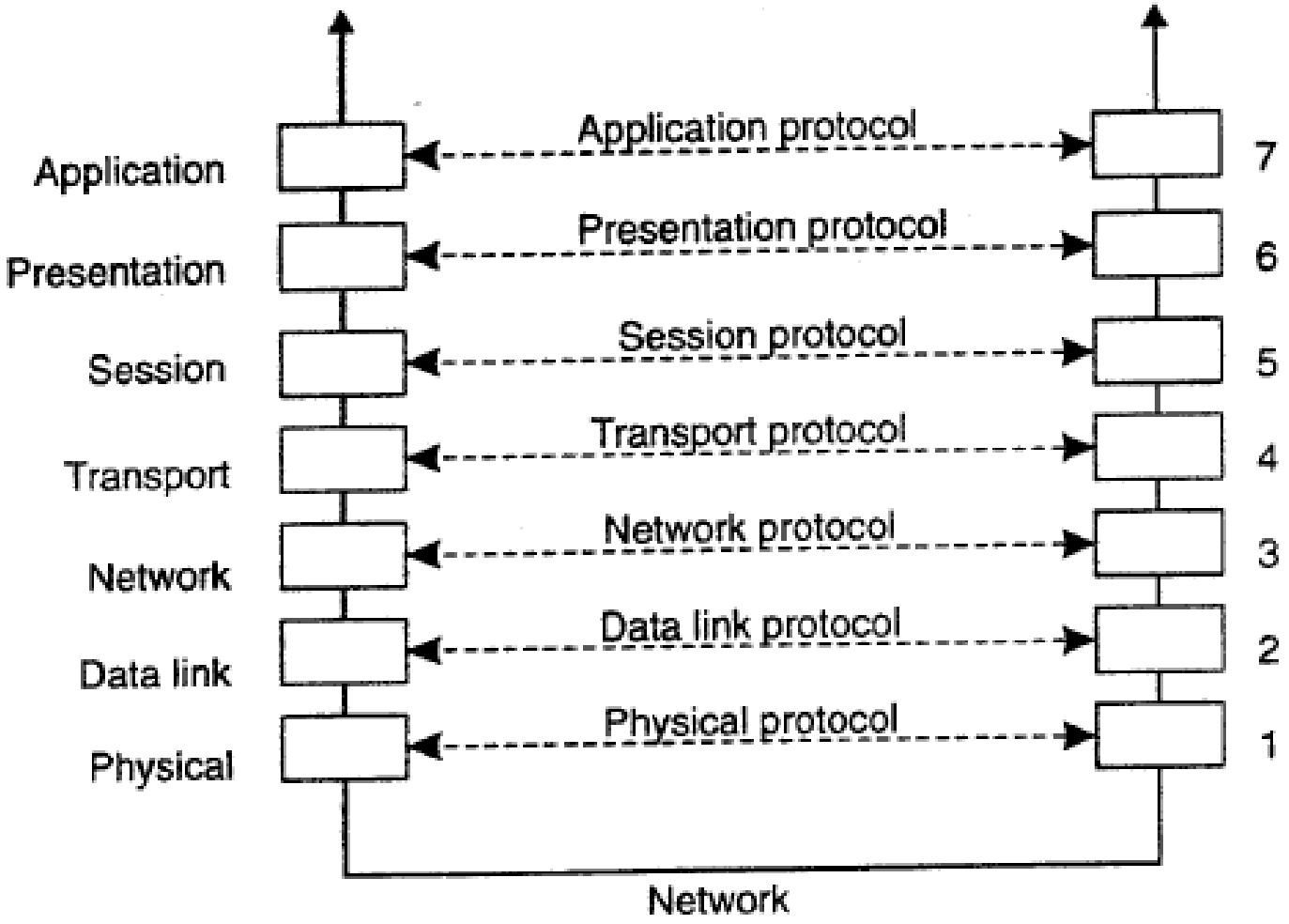
Pipe

- Access the out put of each process (stdout)
- Each process feeds directly as input of stdin.

Pipe (continued)



Communication in Distributed System



DNS, DHCP, FTP, HTTPS, IMAP, LDAP, NTP, POP3, RTP, RTSP, SSH, SIP, SMTP, SNMP, Telnet, TFTP

JPEG, MIDI, MPEG, PICT, TIFF

NetBIOS, NFS, PAP, SCP, SQL, ZIP

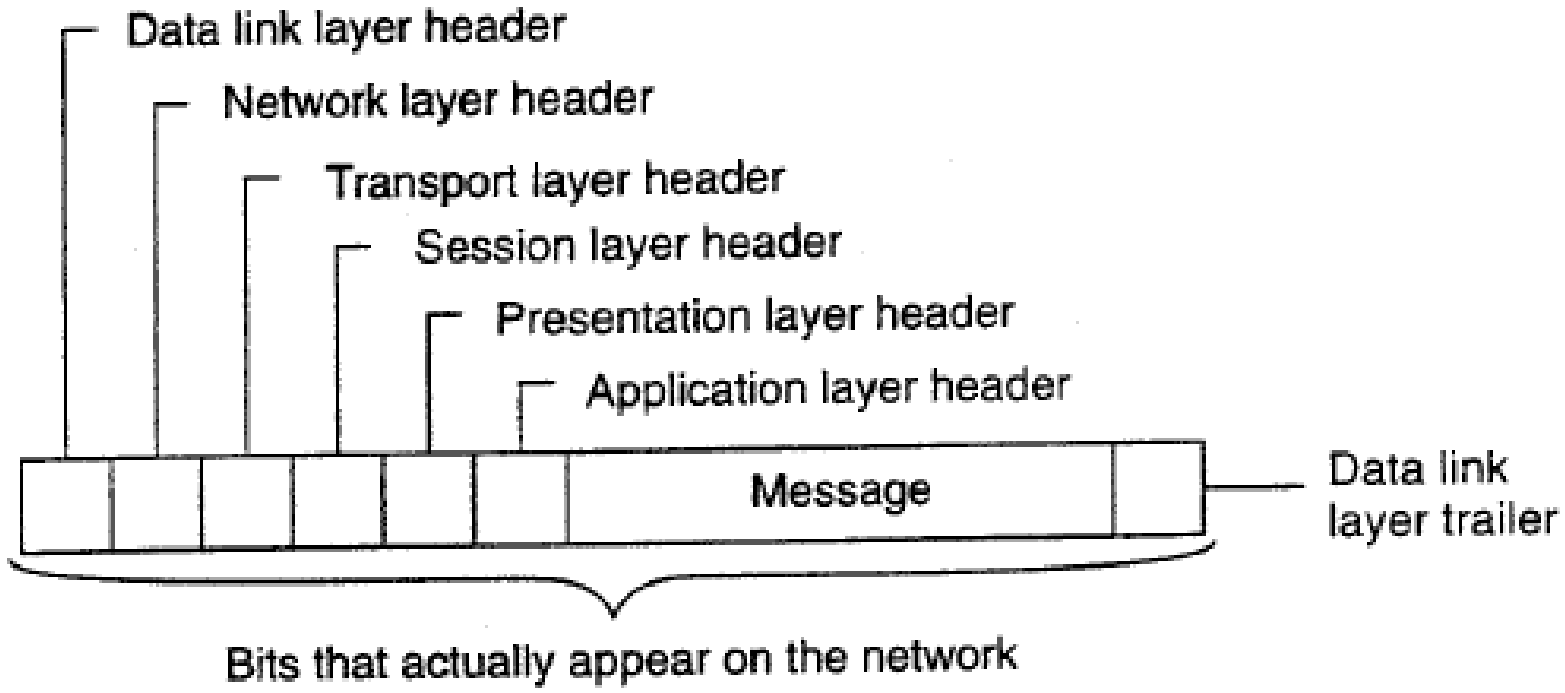
TCP, UDP

ICMP, IGMP, IPsec, IPv4, IPv6, IPX, RIP

ARP, ATM, CDP, FDDI, Frame Relay, HDLC, MPLS, PPP, STP, Token Ring

Bluetooth, Ethernet, DSL, ISDN, 802.11 Wi-Fi

Communication in Distributed System



Type of Communication in Distributed system

Remote Procedure Call

- **Message Oriented Communication**
- **Stream Oriented Communication**
- **Stream Synchronization**
- **Multicast communication**

Reference :

- Tanenbaum A.S. and van Steen M., Distributed Systems: Principles and Paradigms, 2nd Edition, Prentice Hall, 2007. Chapter 4.

Type of Communication in Distributed system

- **Remote Procedure Call**
- **Message Oriented Communication**
- **Stream Oriented Communication**
- **Stream Synchronization**
- **Multicast communication**

Java Thread

Threads are objects in the Java language. They can be created by using two different mechanisms.

- Create a class that extends the standard *Thread* class.
- Create a class that implements the standard *Runnable* interface.

1. Create a class by extending the Thread class and override the run() method:

```
class MyThread extends Thread {
public void run() {
// thread body of execution
}
}
```

2. Create a thread object:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of created thread:

```
thr1.start();
```

```
/* ThreadEx1.java: A simple program creating and invoking a thread object by
extending the standard Thread class. */
class MyThread extends Thread {
public void run() {
System.out.println(“ this thread is running ... ”);
}
}
class ThreadEx1 {
public static void main(String [] args ) {
MyThread t = new MyThread();
t.start();
}
}
```

Java Thread (Continued)

Create a class that implements the interface Runnable and override run() method:Create a class that extends the standard *Thread* class.

- Create a class that implements the standard *Runnable* interface.

```
class MyThread implements Runnable {
…
public void run() {
// thread body of execution
}
}
```

2. Creating Object:

```
MyThread myObject = new MyThread();
```

3. Creating Thread Object:

```
Thread thr1 = new Thread(myObject);
```

4. Start Execution:

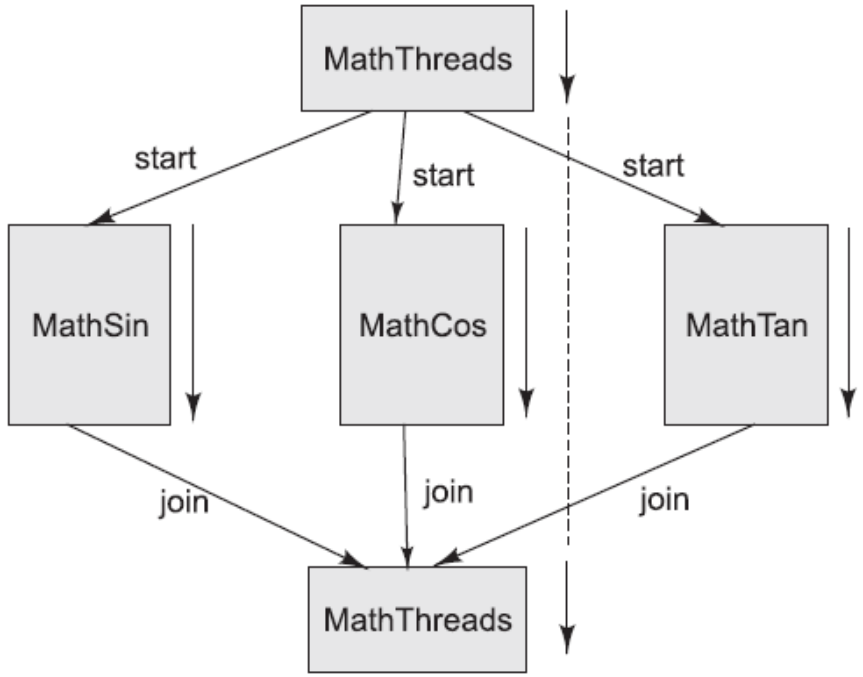
```
thr1.start();
```

A JAVA PROGRAM WITH MULTIPLE THREADS

To illustrate creation of multiple threads in a program performing concurrent operations, let us consider the processing of the following mathematical equation:

$$p = \sin(x) + \cos(y) + \tan(z)$$

- As these trigonometric functions are independent operations without any dependencies between them, they can be executed concurrently. After that their results can be combined to produce the final result



THREAD PRIORITY

- All the thread instances the developer created have the same priority.
- The process will schedule fairly without worrying about the order.
- It is important for different threads to have different priorities.
- Important threads should always have higher priority than less important ones, while threads that need to run quietly as a Daemon may only need the lowest priority.
- Example: the garbage collector thread just needs the lowest priority to execute, which means it will not be executed before all other threads are scheduled to run.
- The Thread class provides 3 constants value for the priority:

```
MIN_PRIORITY = 1, NORM_PRIORITY = 5, MAX_PRIORITY = 10
```