Distributed System**:**

**·**A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

**·autonomous nodes** need to collaborate in order to make users believe they are dealing with a single system in a single coherent system.

**·**Computation is **concurrent**: More than one application running.

**·No shared** state: **No single global clock** that all programs follow.

**·**Communication have **large** duration.

**·**Components may exchange data **at variable rates**. Different components process data **at different rates**. Hence **asynchrony** exists.

node/computing element**:**

**·**can be a **hardware device** or a **software process**.

**·**Nodes can be **interconnected** in anyway.

**·**Although nodes can act **independently** from one another, they **cannot ignore** one another. Otherwise, it will be meaningless to put them to compose the same system.

Network may NOT reliable or secure or homogeneous**:**

**·lack** of **software reliability**: reliable message exchange between nodes such as retrying messages, acknowledging messages, verifying message integrity.

**·Hardware** associated **power failures** – network switches have a mean time between **failures** (e.g., the route between you and the server you get data from)

**·**the result is a network of networks, each network is managed with local scope by a **different** group of administra**:**

**·lack** of **software reliability**: reliable message exchange between nodes such as retrying messages, acknowledging messages, verifying message integrity.

Topology may change**:**

**·**In the **lab**, topology NOT change.

**·**In the **wild**, topology is changing constantly, because **servers** servers are added and removed; **clients**(laptops, networks...) are coming and going.

Latency NOT zero**:**

**·**The minimum round-trip time between two points on earth is

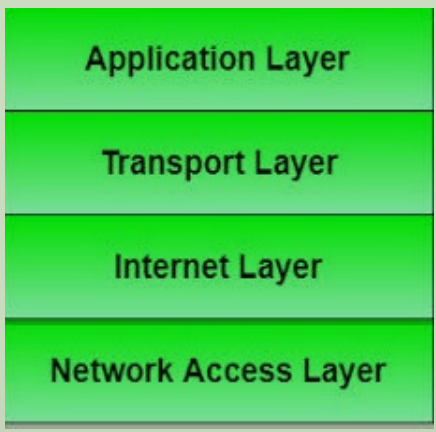
determined by the **maximum speed of information transmission**: the speed of light.

Bandwidth NOT infinite**:**

**·Bandwidth**: The *rate* at which data can be moved. bits/second

**·Throughput**: measure how many packets arrive their destination *successfully*. bits/second

Transport Cost NOT zero**:**



**·**Going from **Application Layer** to **Transport Layer** is NOT free:

Information needs to be **serialised** by **marshalling** to get data onto the wire. **Marshalling** is the process of transforming the memory representation of an object to a data format suitable for storage or transmission.

The cost(money) for setting and running the network NOT zero.

Synchronization**:**

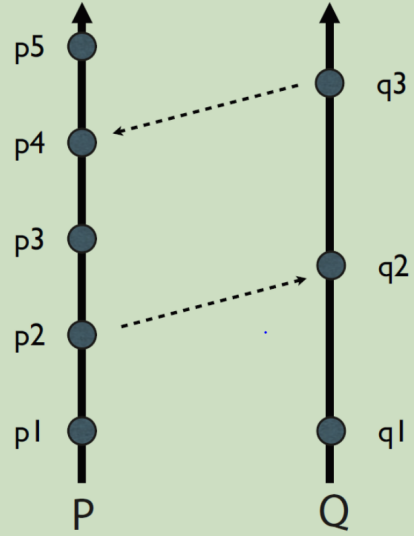
**·Data Synchronization**:keeping multiple copies of a data set in coherence with one another, given that the various copies are located in different nodes. e.g., when you ‘sync’ a mobile device to a laptop to copy photos back and forth, this is a form of data synchronization.

**·Process Synchronization**:multiple processes needing to act together to achieve a certain overall purpose.

**·Synchronization** is needed when Multiple processes need to agree on the ordering of events, such as whether message m1 from process P was sent before or after message m2 from process Q.

**·Synchronization** is also needed when multiple processes try to simultaneously access a shared resource, such as a printer, and should, instead, cooperate in granting each other temporary exclusive access

Obtaining the Ordering of Events:



**Assumptions:**

1. P and Q are two distinct processes running on different nodes, each executing a sequence of instructions, p1 to p5 and q1 to q3, respectively.

2. The dotted lines represent the transmission of a message from one process to another.

3. P and Q do not have any shared notion of time.

**Conclusions:**

• p1 and q1 could happen in any order, i.e., between the two, we do not know which one happens first.

• We can be sure that q2 happens after p2, because q2 can only execute after receiving a message from p2. But, we cannot know how long after. p2->q2 p2->p3

• We cannot say anything about p3 and q2. It could be that p3 is very fast, and happened immediately after p2, and before q2. Or it could be that there was a long delay after p2 and that it happened after q2. We do not know.

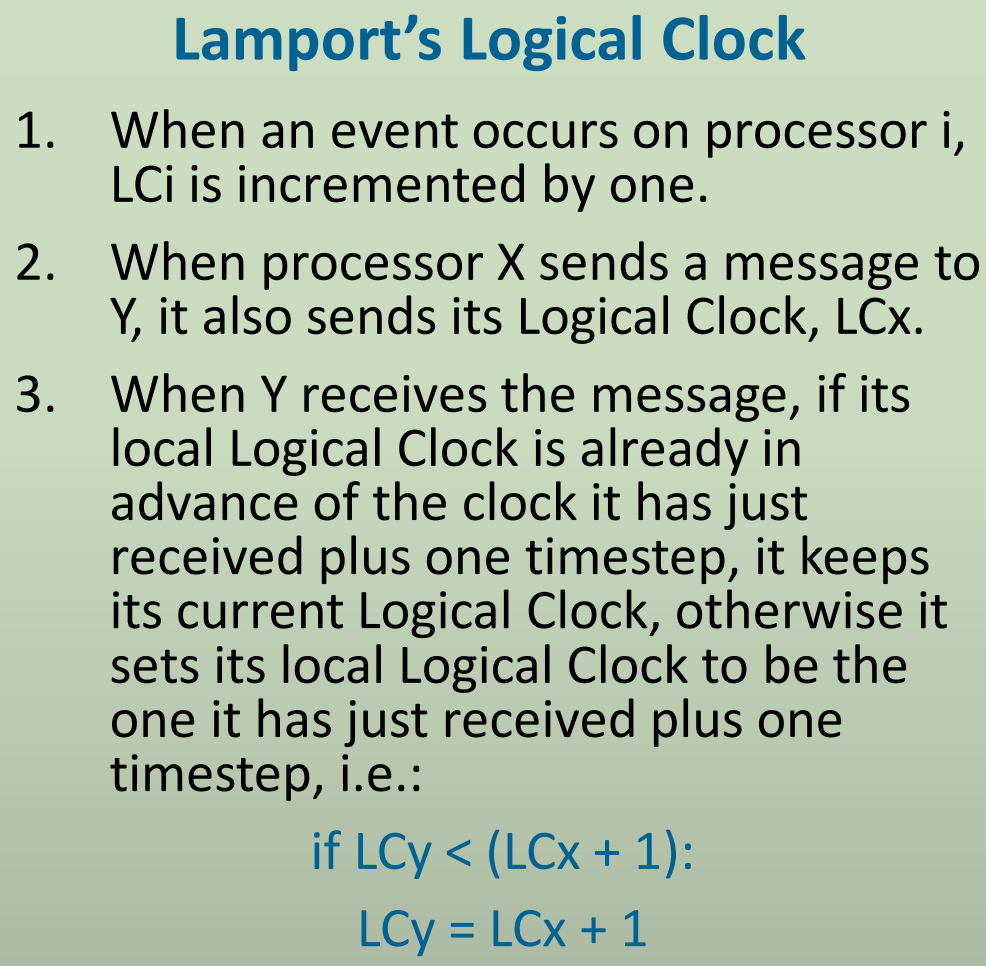
• In spite of the apparent ‘visual’ ordering, we know p4 actually happens after q3, because a message was sent from q3 and received at p4. q3->p4

• We can safely say that p5 happens after q3, since it happens after p4, which we know happens after q3.

Logical Clocks**:**

**https://youtu.be/q\_UZ532Os14**

**·Logical Clocks** use the fact that an implicit partial ordering of events can be obtained from the simple sending and receiving of messages between processes in a distributed system. As such, they do NOT measure ‘real time’, instead, provide a distributed incremental pseudo time to events in a distributed system.



Stop events from Happening Simultaneously when Sharing resources**:**

**·Two Solutions:** Centralized Lock Server (and Mutual Exclusion Locks (mutex)) ; Two Phase Commit Algorithm

Centralized Lock Server and Mutual Exclusion Locks (Mutex):

**Client:**

1. Sends a request to the lock server for a mutex on a given resource.

2. When a reply comes back, it starts executing its critical process over the resource.

3. When its finished, it sends a message to release the mutex.

**Lock Server:**

1. If the resource is available, it marks it as being used by the client and sends a reply to say that the lock has been granted. Otherwise, it puts the request in a queue.

2. When the mutex is released by the client, if another client wants the resource (i.e., is in the queue waiting), pass the mutex to them, otherwise mark it as being available.

**Limitation:** a single point of failure(Central Lock Server)

Two Phase Commit Algorithm:

**·Two Phase Commit Algorithm** lets a sequence of events *either* runs to successful completion, *or*, if the sequence fails, then the overall system is returned to its original state as though nothing had happened. In other words, it allows intermediate steps that have occurred to be undone (i.e., rolledback) to return things back to a safe, sensible state, if a fault is detected.

**1.** A coordinator node requests a transaction, and sends a request to all participants nodes

**2.** e.g., to node C1, it sends ‘request to remove X pounds from account’, and to C2 sends ‘request to add X pounds to account’.

**3.** All participants respond as to whether they are willing and able to execute the request, and send VOTE\_COMMIT or VOTE\_ABORT.

**4.** They log their current state, and then perform the transaction.

**5.** All participants log their vote

**6.** The coordinator looks at the votes. If everyone has voted to commit, then the co-ordinator sends a GLOBAL\_COMMIT to everyone; otherwise it sends a GLOBAL\_ABORT.

**7.** On receiving the decision from the coordinator, all participants record the decision locally. If it was an ABORT, participants ROLL BACK to their previous safe state.

Bully Algorithm choose a coordinator node:

**·Bully Algorithm** is used to choose a coordinator node from a set of candidate nodes.

1. P sends an ELECTION message to all nodes with higher numbers.

2. If no one responds,P wins the election and becomes the coordinator. It informs all the other nodes that it is now the coordinator.

3. If one of the higher-numbered nodes Q answers, P concedes that it is not the winner, and Q begins the election process again until one node eventually wins.

**Note1:** Assume nodes are ordered.

**Note2:** the algorithm relies on the use of timeouts to decide when to ‘give up’ waiting for responses from nodes that have potentially died (so the usual problem of ‘how long is it reasonable to wait’ applies here).

Clock Synchronization**:**

**·Two Solutions:** Cristian’s Algorithm ; The Berkeley Algorithm

Cristian’s Algorithm**:**

**·Cristian’s algorithm** works between a process P, and a time server S connected to a source of UTC (Coordinated Universal Time).

**·**It relies on the accuracy of an estimate based on the RTT(Round Trip Time), which is the elapsed time between the time when a request is sent from P to S, and the time when a response is received by P from S.

**1.**  P requests the time from S.

**2.** After receiving the request from P, S prepares a response and appends the time T from its own clock. The response is sent to P.

**3.** P then sets its time to be T + RTT/2, where RTT is Round Trip Time of the request P made to S.

This method assumes that the RTT is split equally between both request and response, which may not always be the case but is a reasonable assumption on a LAN connection.

Berkeley Algorithm**:**

**·**The server process is called the master process, polls other slave processes periodically.

**·**The clock synchronization method used in this algorithm, called the average, cancels out individual clock’s tendencies to drift.

**1.** A master is chosen via an election process such as the Bully Algorithm.

2. The master polls the slaves, who reply with their time in a similar way to Cristian’s algorithm.

**3.** The master observes the round-trip time (RTT) of the messages and estimates the time of each slave and its own.

**4.** The master then averages the clock times, ignoring any values it receives far outside the values of the others.

**5.** Instead of sending the updated current time back to the other process, the master sends out the amount (positive or negative) that each slave must adjust its clock. This avoids further uncertainty due to RTT at the slave processes.

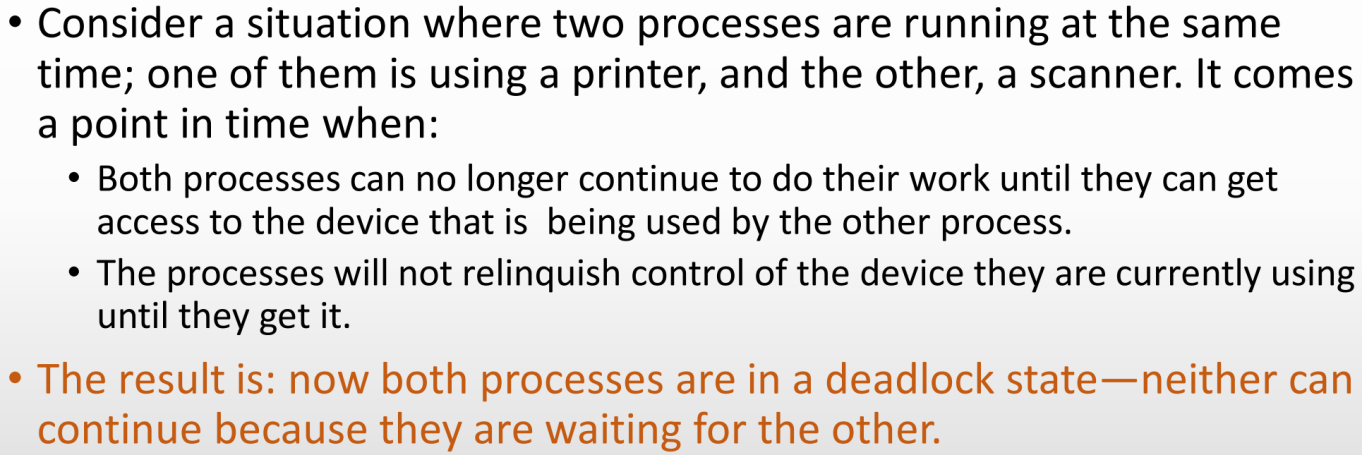
Deadlock**:**

**Mutual exclusion:** at least one resource must be non-shareable, i.e. only one process can use that resource at a given time.

**Hold and wait:** a process is currently holding at least one resource, and is requesting at least one more resource that is currently being held by another process (i.e., it is waiting for a process to release something).

**No pre-emption:** once a process has acquired a resource, nothing (e.g., the operating system or some external factor) can force it to relinquish that resource; it has to do so voluntarily.

**Circular wait:** a process must be waiting for a resource that is being held by another process, which in turn and possibly indirectly, is waiting for the first process to release a resource. Put another way, processes are waiting for one another in such a way that there is a cycle of dependencies between them. For example, if A waits for B, B waits for C, and C waits for A, we have a cycle of waiting that would fulfil this condition.



• Prevent deadlock from occurring in the first place by making sure that it is never possible for all four of the conditions to be true at the same time (you might like to think about what this means in each of the four cases).

• Avoid deadlock by making sure that, even though it is potentially possible for all four conditions to hold at the same time, that they never do.

• Detect a deadlock and deal with the consequences (this last approach is very similar to ‘avoid’ really, in that it requires having some mechanism for detecting whether all four conditions are about to hold (for avoid) or actually do hold (for detect)).

Whether you decide to go for a detect or avoid approach really depends on how likely or frequently a deadlock is likely to happen—if some property of your system means that deadlock is possible, but hugely unlikely, it might make sense to allow it to occur once in a while and to then deal with the consequences, rather than to incur the overhead of trying to avoid it from happening in the first place.

Naming**:**

**·**Names are used to uniquely identify entities.

**·**In **Internet Naming**, every computer connected to the Internet needs is addressable so that other computers on the net can “talk” to each other. Naming entities is the addressing mechanism via which a computer on the Internet is uniquely identified.

Centralised Naming Approach**:**

**·**Any name is handed out once and only once. In this approach, there is a single point of contact, that either validates that a name is unique, or makes up a unique name and hands that out on demand.

**·advantage:** easy to maintain&update.provide a uniform&consistent naming scheme.

**·disadvantage:** the single point of contact has to deal with every request, so it is NOT scalable, and it creates a single point of failure.

Free-for-all Approach**:**

**·Free-for-all** allows any object that wants a name to make up its own name.

**·advantage:** NO single point of failure.

**·disadvantage:** may cause name conflicts(two or more nodes choose the same name)

Delegating Naming Responsibilities Approach**:**

**·**Smaller parts of the system can allocate names, and governed by some rules.

**·**Balance the conflicting issues associated with single points of failure and scalability, but it raises questions as to what rules are appropriate for each system.

MAC Addresses Media Access Control Address **:**

**·**The MAC Address is a unique identifier given to each network device in a system: this means that every ethernet or wifi card in a computer has one MAC address.

**·**A MAC address is a 48 bit number. The first 24 bits identify OUI(manufacturer of NIC). The last 24 bits identify NIC itself.

**·**NOT tell the location of the device.

IP Addresses Internet Protocol Address **:**

**·**The IP Addresses are uniquely assigned to devices by ISP(Internet Service Providers).

**·**Tell some information about the location of the device, but NOT the exact location.

Domain Names**:**

**·Domain Names** are created because IP addresses are hard to read.

**·DNS(Domain Name System)** is a distributed system built on top of the Internet. **DNS** is used to create associations between readable names and IP addresses. A device’s Domain Name can be mapped into the device’s IP address.

**·**Unlike MAC and IP addresses, DNS NOT allocate ‘batches’ of names upfront, and needs to respond in real-time to requests to translate names into IP addresses. It achieves this by being in itself a

distributed system, consisting of a hierarchy of servers with the most authoritative server at the ‘top’ of the hierarchy, dealing with requests from users.

Protocol**:**

**·**Protocols define sets of rules governing how two or more objects should interact with one another.

**·**An example of an relevant protocol is Hyper-Text Transport Protocol (HTTP), which provides an specification (i.e., a vocabulary) that allows client applications to request resources from Web servers, and Web servers to respond to these requests.

Statelessness of HTTP Protocol**:**

**·**each request made by a client to a server is treated as an independent transaction, with no memory or knowledge of previous requests. In other words, each request is processed without any knowledge of the previous requests made by the client.

Process**:**

**·**A process is an executing instance of a program.

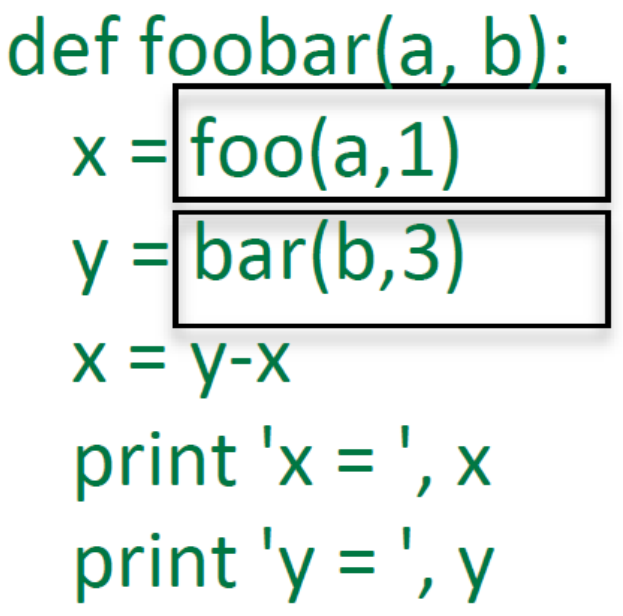
**·**resource usage is typically controlled by an operating system (OS).

**·**Since resources are scarce and differ in their capabilities, an **OS** aims to make the most efficient use possible of those resources.

**·**The **OS** assigns a unique identity to each process and then controls how a process is granted access to computing resources.

**·**The **OS** also controls how a process is granted access to storage resources by assigning an address space to that process.

**·**When the **OS** ensures that each process P has a single address space A that is exclusive to P , we are allowed a sequential reading of the steps that comprise the process.

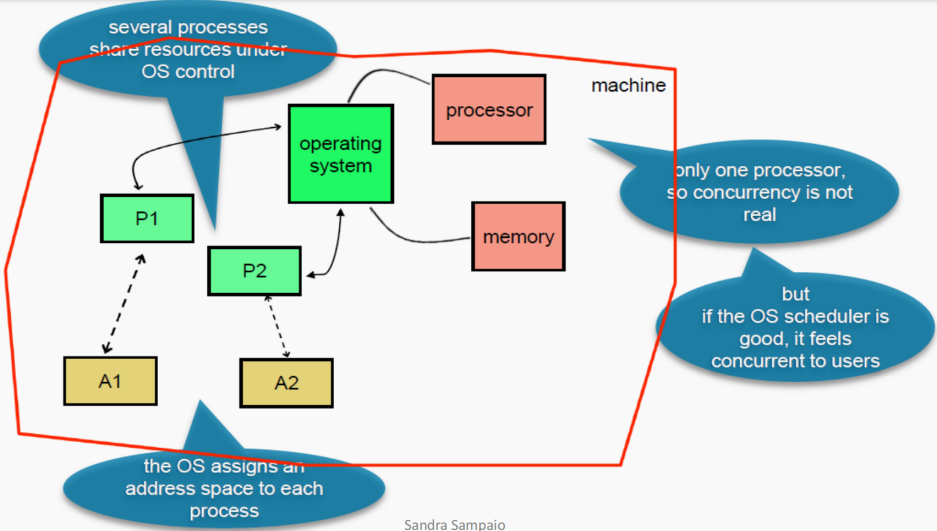


If functions “foo” and “bar” take long time to run, we can run them concurrently, in different **processes**; even better in parallel, in different **processors**.

Multi-Tasking**:**

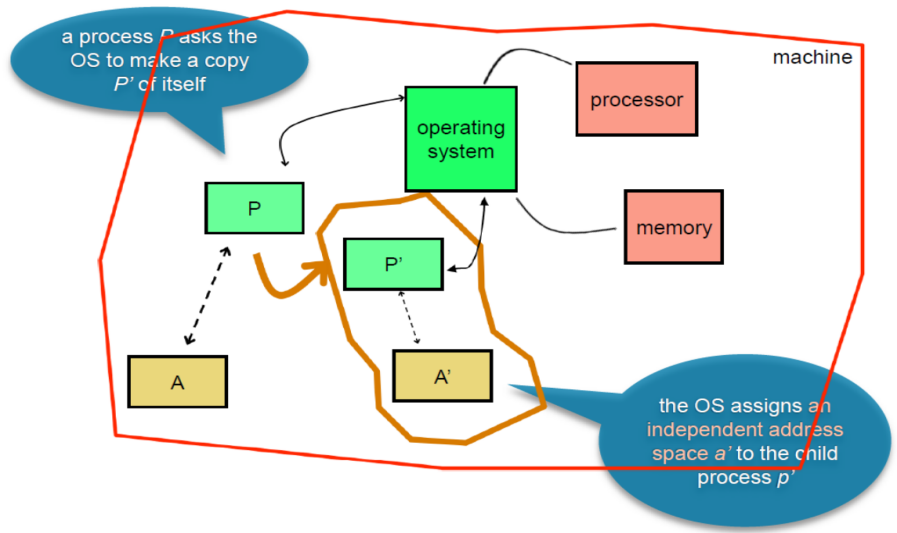
**·**the ability of a single processor or core to execute multiple tasks or processes simultaneously, by dividing its processing time between them. This is typically achieved through a technique called time-sharing, where the processor switches rapidly between different tasks, giving the appearance that they are running simultaneously (processes are NOT executing concurrently).

Multi-Processing**:**



**·**the use of multiple processors or cores within a single computer system to perform multiple tasks or processes simultaneously. This increases throughput and performance. Each processor or core can be assigned to a different task or process, allowing them to run independently without interfering with each other.

Multi-Processing by Forking**:**



**·**creating a copy of an existing process, known as the parent process, to create a new process, known as the child process. The child process is an exact copy of the parent process, including its memory space, data, and execution context. However, the child process has its own unique address space and process ID (PID), allowing it to run independently of the parent process.

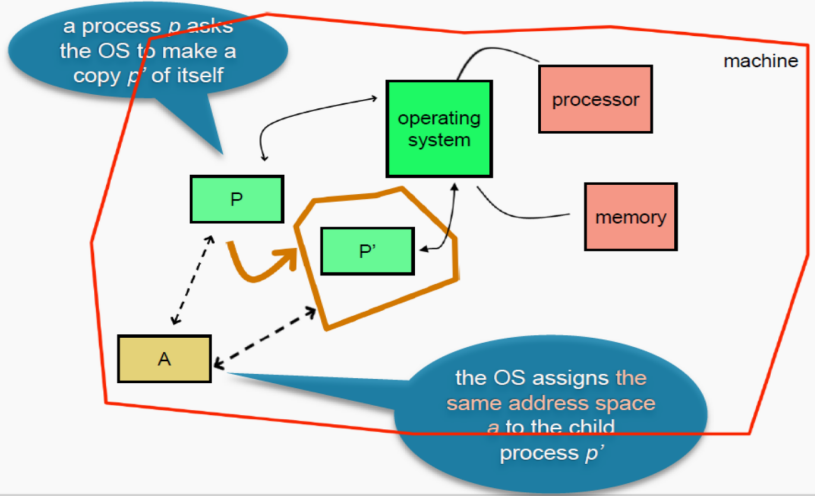
**·**if one process modifies a variable in its address space, the other process NOT see the change.

**·**The child process starts executing after the OS call; The parent process can continue or wait for the child process to execute; Finally, the parent process must know how and when the child process completes the execution.

**·***Advantage:* safe due to independent address space)

*Disadvantage:* expensive due to copy

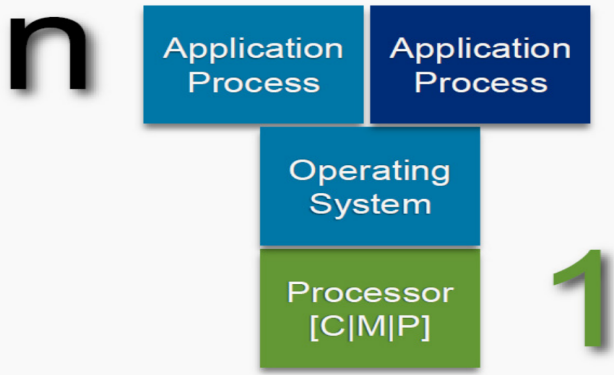
Multi-Processing by Threading**:**



**·**parent processes and children processes can interact and share.

**·**the address space is shared instead of copied. If one process modifies a variable, all other processes see it.

Concurrent Computing**:**



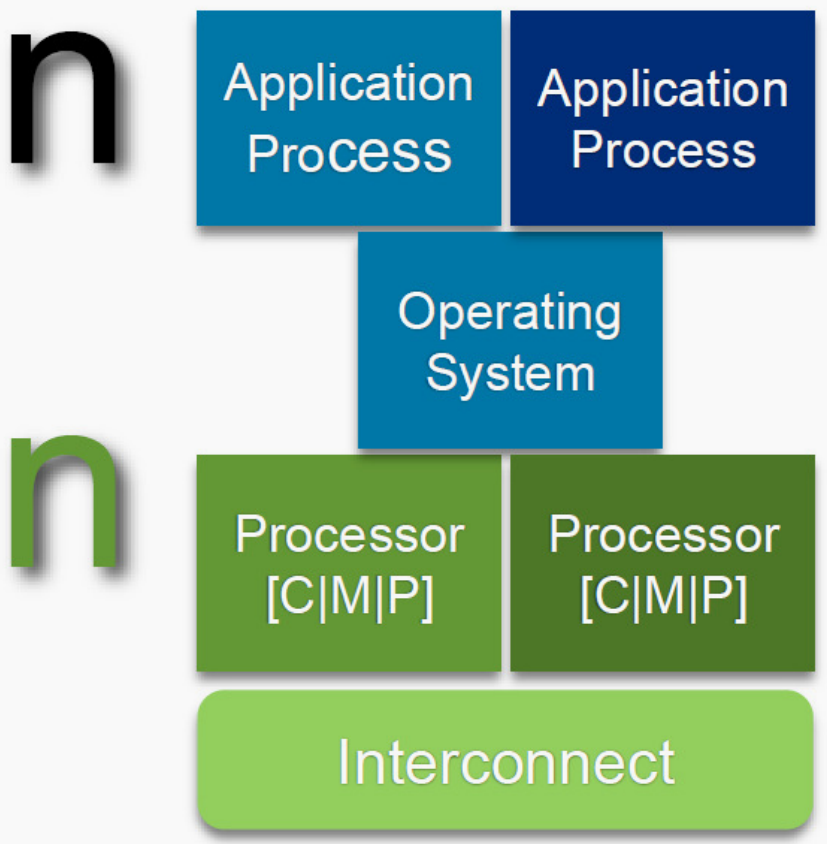
**·**One Processor & Multiple Processes.

**·**Processes are often threads.

**·**The OS schedules the execution of n copies of a process Pi,

1<= i<= n, to run in the same processor, typically sharing a single address space.

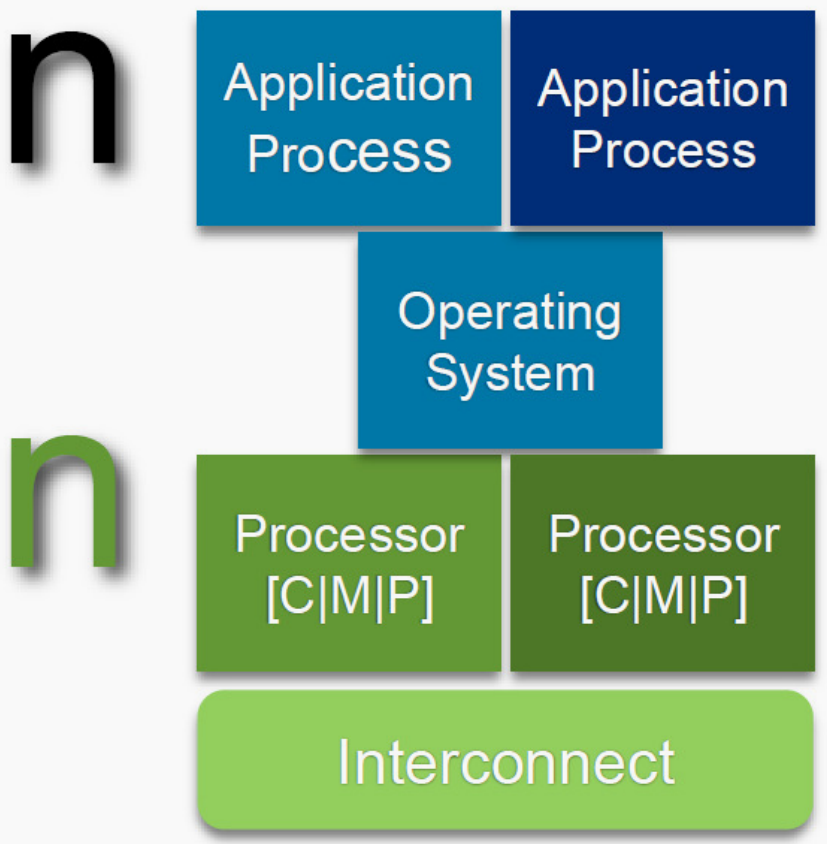
Parallel Computing**:**



**·**Multiple Processors & Multiple Processes & One Machine.

**·**n copies of a process Pi, 1<=i<=n, each run in one of m, 1<=j<=m, different processors Cj, possibly (but not necessarily) sharing a single address space A.

Distributed Computing**:**



**·**Multiple Processors & Multiple Processes & Multiple Machines.

**·**n (not necessarily identical) processes Pi, 1 =< i =< n, each run in one of m, 1<=j<=m, different machines Mj, that cannot share a single address space A (and therefore must communicate).