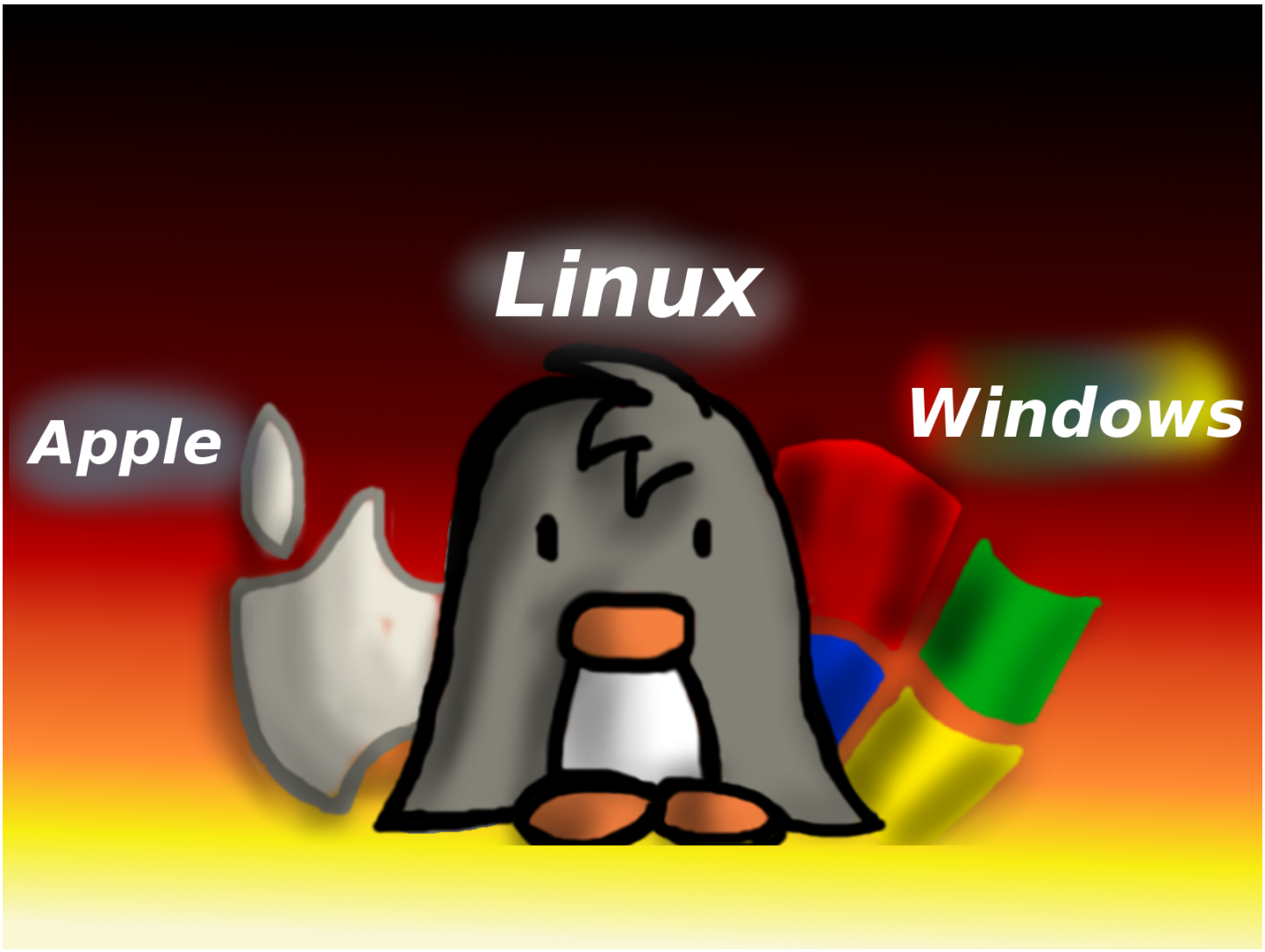


The 3 main types of OS: Mac, Windows, and Linux

The 10 Operating System Concepts Software Developers Need to Remember

James Le Follow
Dec 28, 2017 · 22 min read

Do you speak binary? Can you comprehend machine code? If I gave you a sheet full of 1s and 0s could you tell me what it means/does? If you were to go to a country you've never been to that speaks a language you've never heard, or maybe your heard of it but don't actually speak it, what would you need while there to help you communicate with the locals?

You would need a translator. Your **operating system** functions as that translator in your PC. It converts those 1s and 0s, yes/no, on/off values into a readable language that you will understand. It does all of this in a streamlined graphical user interface, or GUI, that you can move around with a mouse click things, move them, see them happening before your eyes.

Knowing how operating systems work is a fundamental and critical to anyone who is a serious software developer. There should be no attempt to get around it and anyone telling you it's not necessary should be ignored. While the extend and depth of knowledge can be questioned, knowing more than the fundamentals can be critical to how well your program runs and even its structure and flow.

Why? When you write a program and it runs too slow, but you see nothing wrong with your code, where else will you look for a solution. How will you be able to debug the problem if you don't know how the operating system works? Are you accessing too many files? Running out of memory and swap is in high usage? But you don't even know what swap is! Or is I/O blocking?

And you want to communicate with another machine. How do you do that locally or over the internet? And what's the difference? Why do some programmers prefer one OS over another?

In an attempt to be a serious developer, I recently took Georgia Tech's course "**Introduction to Operating Systems**." It teaches the basic OS abstractions, mechanisms, and their implementations. The core of the course contains concurrent programming (threads and synchronization), inter-process communication, and an introduction to distributed OSs. I want to use this post to share my takeaways from the course, that is the **10 critical operating system concepts** that you need to learn if you want to get good at developing software.

But first, let's define what an operating system is. An Operating System (OS) is a collection of software that manages computer hardware and provides services for programs. Specifically, it hides hardware complexity, manages computational resources, and provides isolation and protection. Most importantly, it directly has privilege access to the underlying hardware. Major components of an OS are file system, scheduler, and device driver. You probably have used both Desktop (Windows, Mac, Linux) and Embedded (Android, iOS) operating systems before.

There are 3 key elements of an operating system, which are: (1) **Abstractions** (process, thread, file, socket, memory), (2) **Mechanisms** (create, schedule, open, write, allocate), and (3) **Policies** (LRU, EDF)

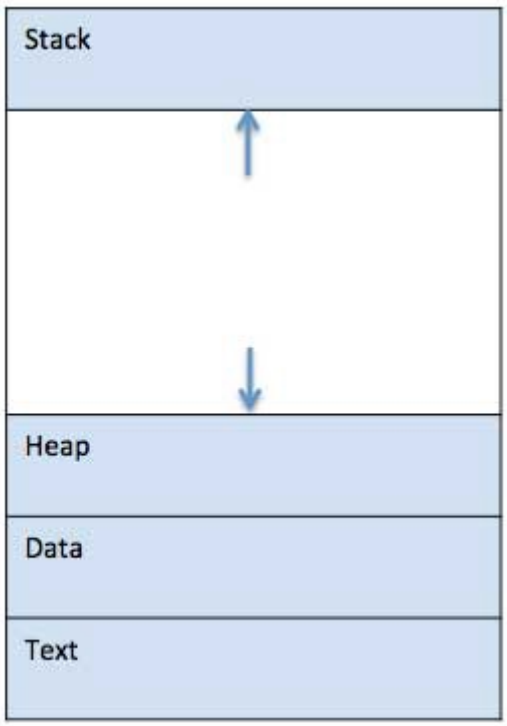
There are 2 operating system design principles, which are: (1) **Separation of mechanism and policy** by implementing flexible mechanisms to support policies, and (2) **Optimize for common case:** Where will the OS be used? What will the user want to execute on that machine? What are the workload requirements?

The 3 types of Operating Systems commonly used nowadays are: (1) **Monolithic OS**, where the entire OS is working in kernel space and is alone in supervisor mode; (2) **Modular OS**, in which some part of the system core will be located in independent files called modules that can be added to the system at run time; and (3) **Micro OS**, where the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space.

1—Processes and Process Management

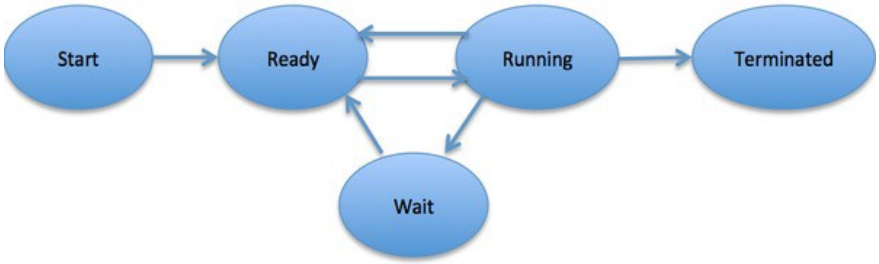
A process is basically a program in execution. The execution of a process must progress in a sequential fashion. To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



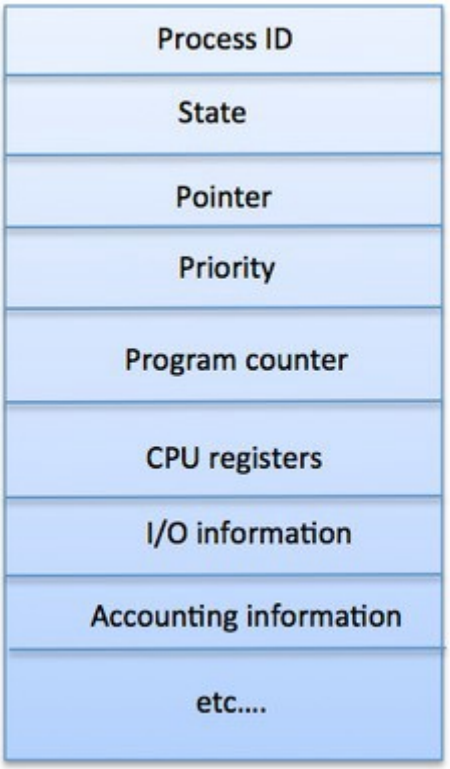
- **Stack:** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- **Heap:** This is dynamically allocated memory to a process during its run time.
- **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data:** This section contains the global and static variables.

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. In general, a process can have one of the following five states at a time:



- **Start:** This is the initial state when a process is first started/created.
- **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
- **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below:



- **Process State:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Process Privileges:** This is required to allow/disallow access to system resources.
- **Process ID:** Unique identification for each of the process in the operating system.
- **Pointer:** A pointer to parent process.
- **Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU Registers:** Various CPU registers where process need to be stored for execution for running state.
- **CPU Scheduling Information:** Process priority and other scheduling information which is required to schedule the process.
- **Memory Management Information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

- **Accounting Information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO Status Information:** This includes a list of I/O devices allocated to the process.

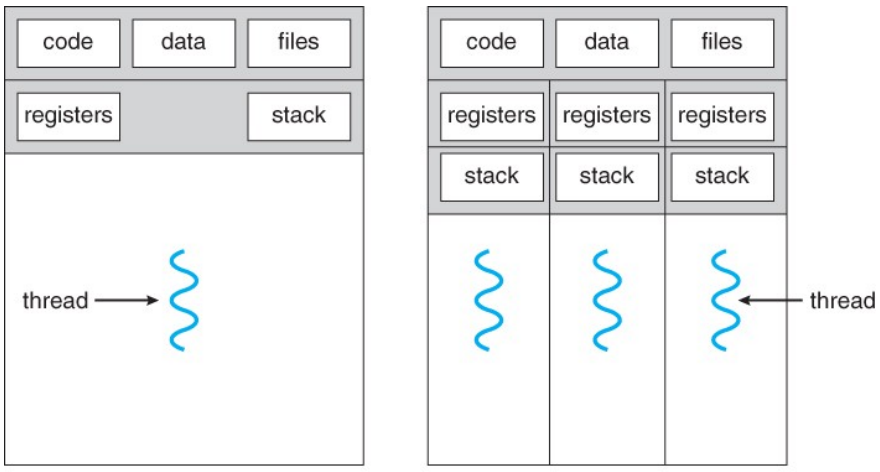
2—Threads and Concurrency

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.



single-threaded process multithreaded process

Advantages of Thread:

- Threads minimize the context switching time.

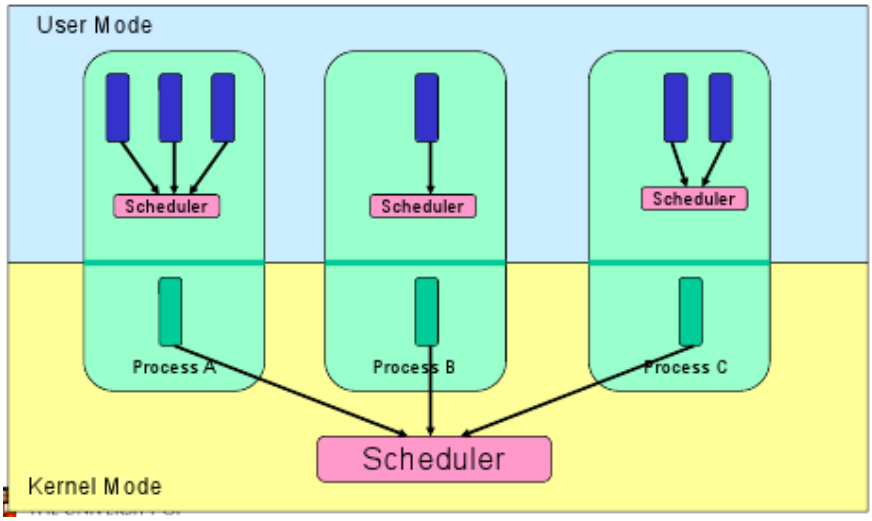
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Threads are implemented in the following 2 ways:

- **User Level Threads:** User managed threads.
- **Kernel Level Threads:** Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.

- User level threads are fast to create and manage.

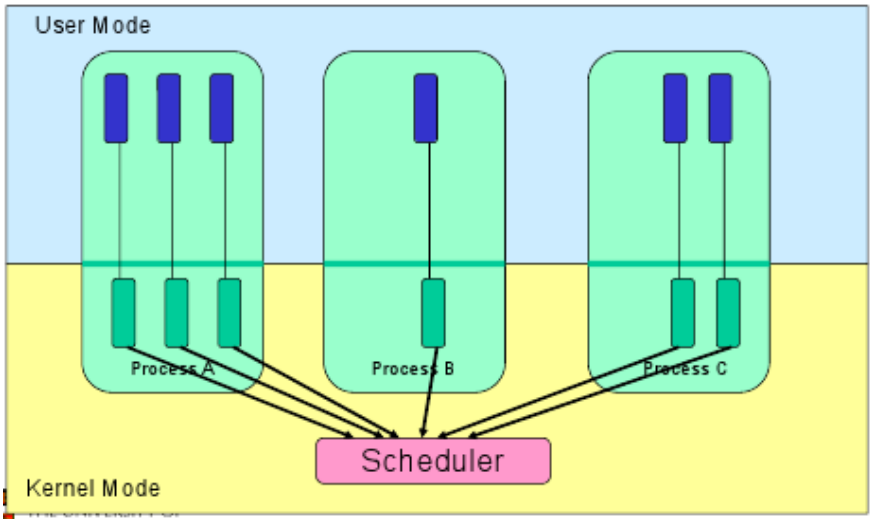
Disadvantages:

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.



Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

3—Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

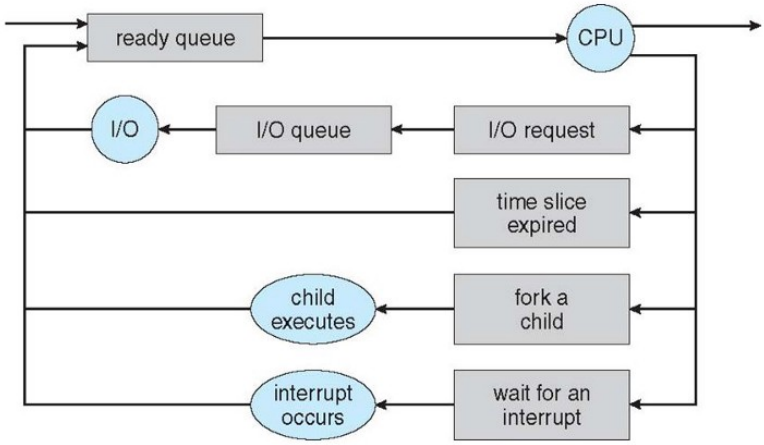
The OS maintains all Process Control Blocks (PCBs) in **Process Scheduling Queues**. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues:

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

Process Scheduling Queues

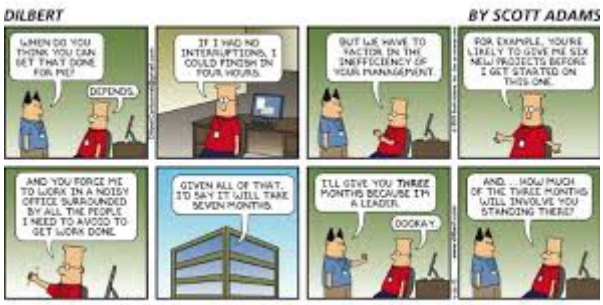
A process migrates among the queues throughout its life:



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-state process model refers to running and non-running states:

- **Running:** When a new process is created, it enters into the system as in the running state.
- **Not Running:** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.



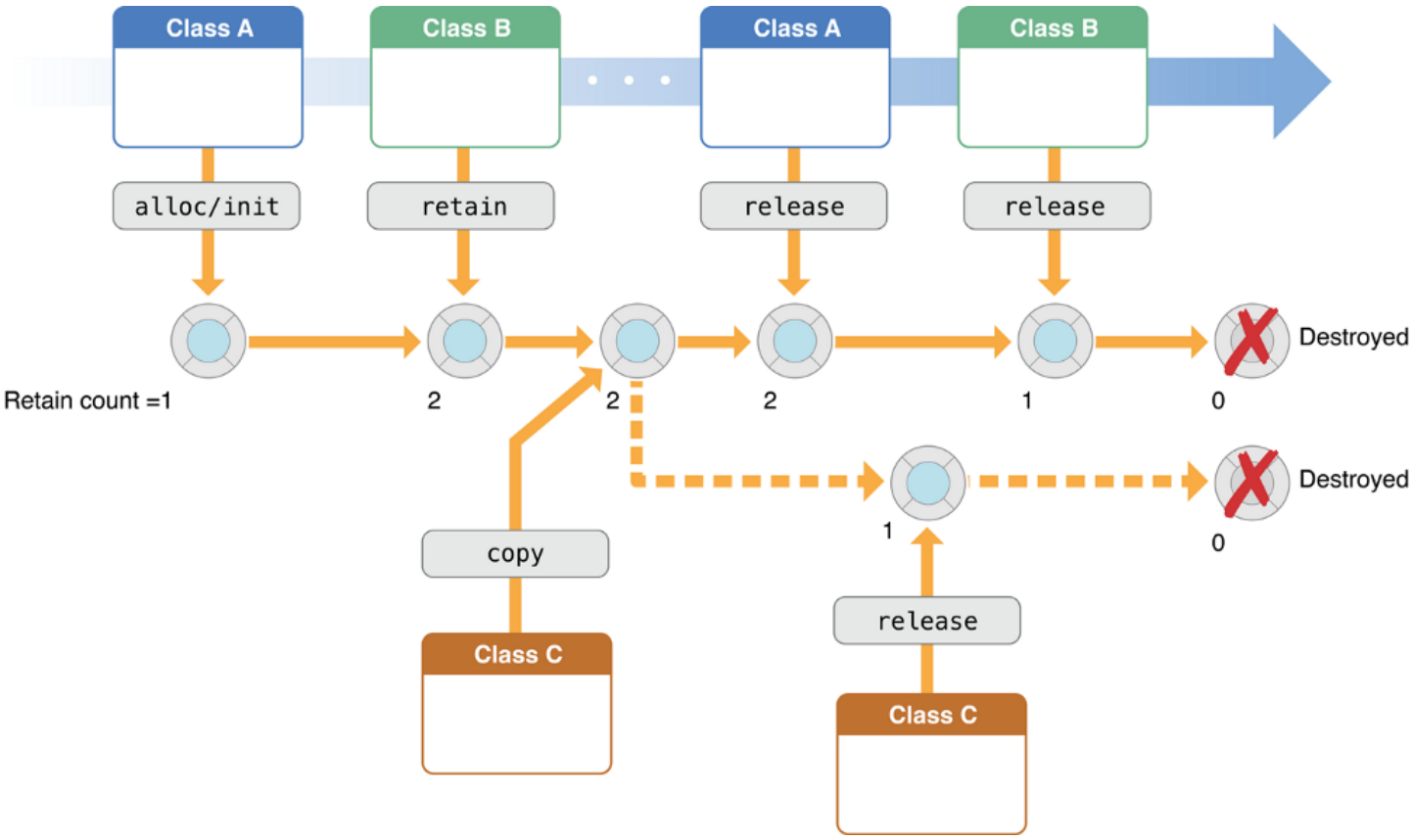
A **context switch** is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use: Program Counter, Scheduling Information, Base and Limit Register Value, Currently Used Register, Changed State, I/O State Information, and Accounting Information.

4—Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.



The **process address space** is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^{31} possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated:

- **Symbolic addresses**: The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
- **Relative addresses**: At the time of compilation, a compiler converts symbolic addresses into relative addresses.
- **Physical addresses**: The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

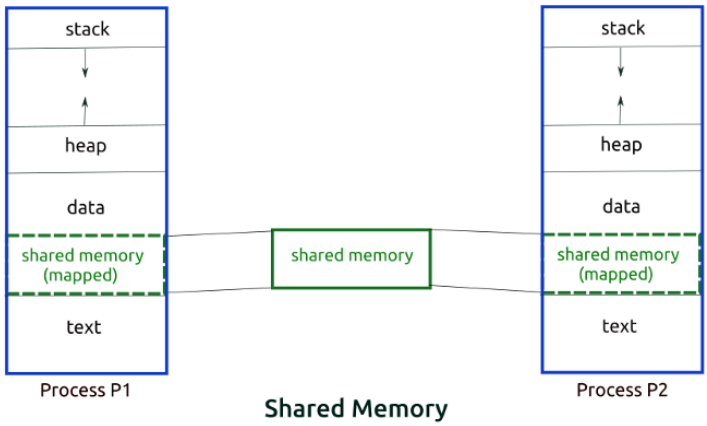
The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

5—Inter-Process Communication

A process can be of 2 types: Independent process and Co-operating process. An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. **Inter-process communication (IPC)** is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways: Shared Memory and Message Parsing.

Shared Memory Method

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it.

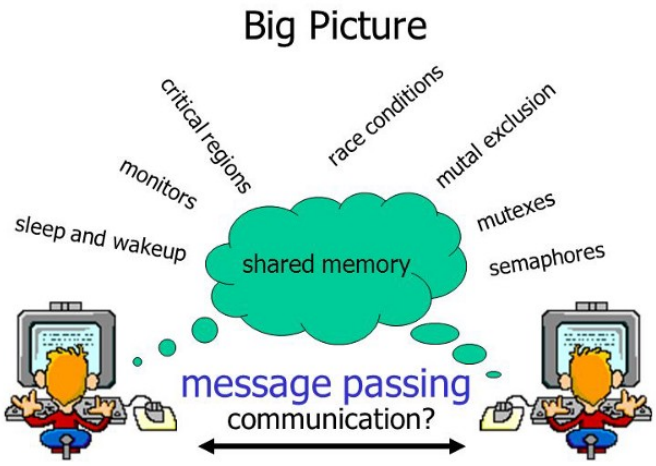


In the bounded buffer problem: First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

Message Parsing Method

In this method, processes communicate with each other without using any kind of of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives. We need at least two primitives: **send**(message, destination) or **send**(message) and **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

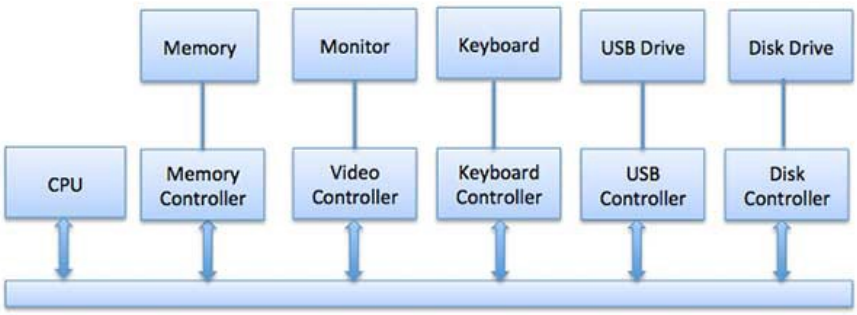
The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

6—I/O Management

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An **I/O system** is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories:

- **Block devices**—A block device is one with which the driver communicates by sending entire blocks of data. For example, hard disks, USB cameras, Disk-On-Key etc.
- **Character Devices**—A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc.



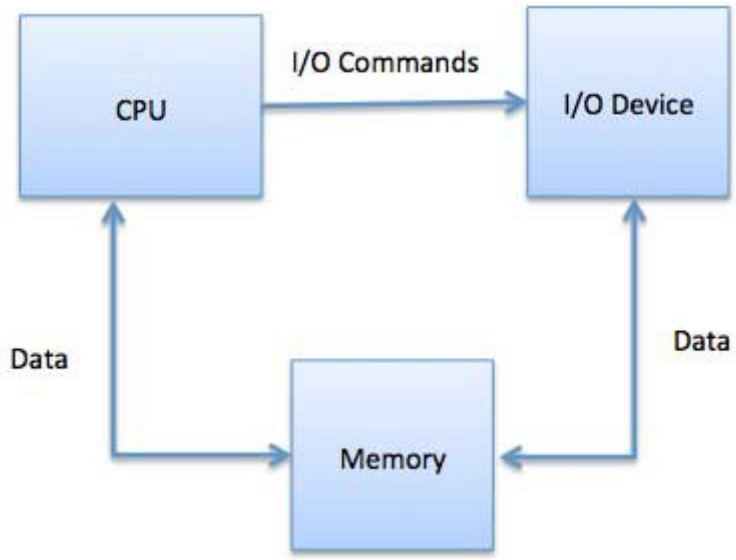
The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

1> Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

2> Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

3> Direct memory access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O

device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

7—Virtualization

Virtualization is technology that allows you to create multiple simulated environments or dedicated resources from a single, physical hardware system. Software called a **hypervisor** connects directly to that hardware and allows you to split 1 system into separate, distinct, and secure environments known as **virtual machines** (VMs). These VMs rely on the hypervisor's ability to separate the machine's resources from the hardware and distribute them appropriately.

The original, physical machine equipped with the hypervisor is called **the host**, while the many VMs that use its resources are called **guests**. These guests treat computing resources—like CPU, memory, and storage—as a hangar of resources that can easily be relocated. Operators can control virtual instances of CPU, memory, storage, and other resources, so guests receive the resources they need when they need them.

Ideally, all related VMs are managed through a single web-based virtualization management console, which speeds things up. Virtualization lets you dictate how much processing power, storage, and memory to give VMs, and environments are better protected since VMs are separated from their supporting hardware and each other. Simply put, virtualization creates the environments and resources you need from underused hardware.

Types of Virtualization:

1. **Data Virtualization:** Data that's spread all over can be consolidated into a single source. Data virtualization allows companies to treat data as a dynamic supply—providing processing capabilities that can bring together data from multiple sources, easily accommodate new data sources, and transform data according to user needs. Data virtualization tools sits in front of multiple data sources and allows them to be treated as single source, delivering the needed data—in the required form—at the right time to any application or user.

2. **Desktop Virtualization:** Easily confused with operating system virtualization—which allows you to deploy multiple operating systems on a single machine—desktop virtualization allows a central administrator (or automated administration tool) to deploy simulated desktop environments to hundreds of physical machines at once. Unlike traditional desktop environments that

are physically installed, configured, and updated on each machine, desktop virtualization allows admins to perform mass configurations, updates, and security checks on all virtual desktops.

3. **Server Virtualization:** Servers are computers designed to process a high volume of specific tasks really well so other computers—like laptops and desktops—can do a variety of other tasks. Virtualizing a server lets it to do more of those specific functions and involves partitioning it so that the components can be used to serve multiple functions.

4. **Operating System Virtualization:** Operating system virtualization happens at the kernel—the central task managers of operating systems. It's a useful way to run Linux and Windows environments side-by-side. Enterprises can also push virtual operating systems to computers, which: (1) Reduces bulk hardware costs, since the computers don't require such high out-of-the-box capabilities, (2) Increases security, since all virtual instances can be monitored and isolated, and (3) Limits time spent on IT services like software updates.

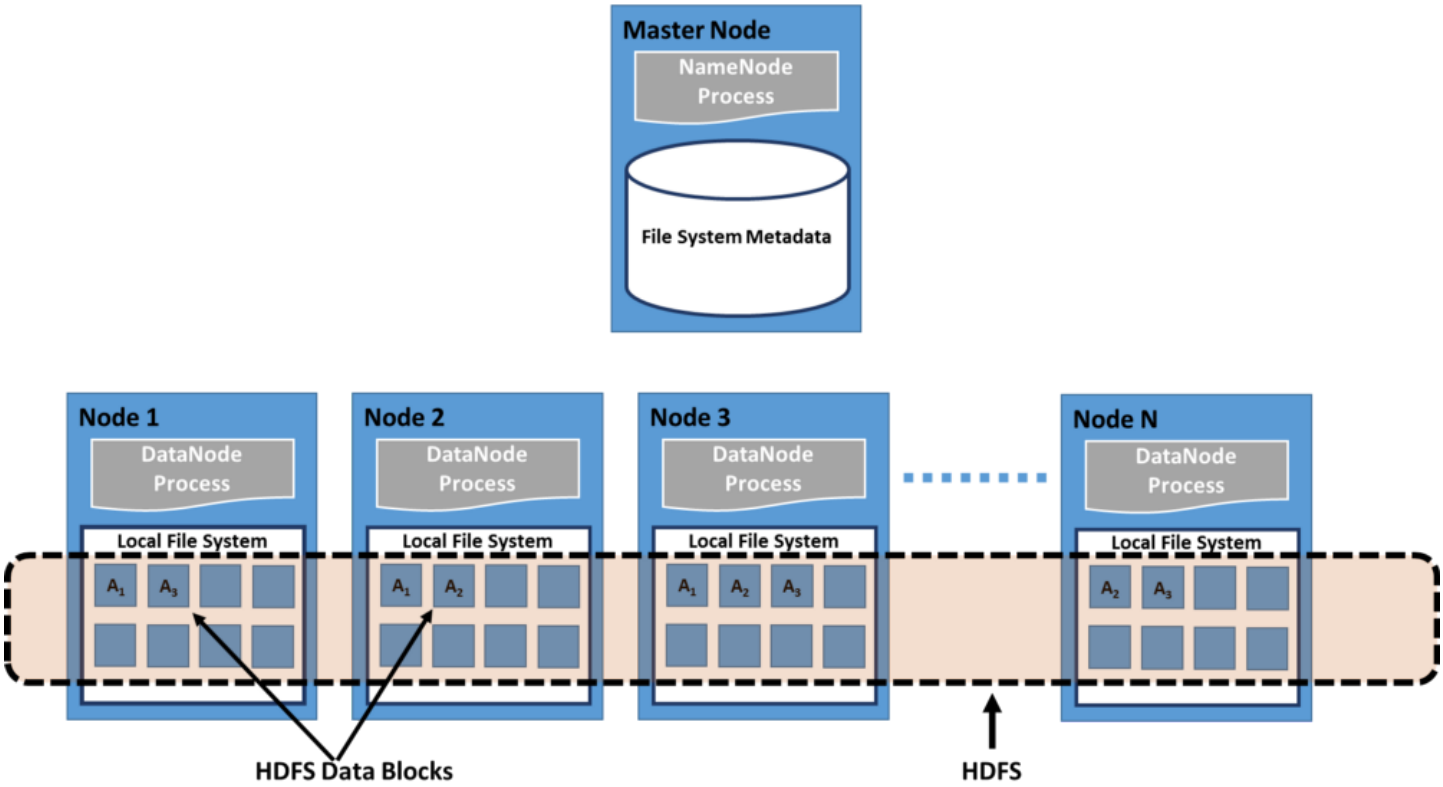
5. **Network Functions Virtualization:** Network functions virtualization (NFV) separates a network's key functions (like directory services, file sharing, and IP configuration) so they can be distributed among environments. Once software functions are independent of the physical machines they once lived on, specific functions can be packaged together into a new network and assigned to an environment. Virtualizing networks reduces the number of physical components—like switches, routers, servers, cables, and hubs—that are needed to create multiple, independent networks, and it's particularly popular in the telecommunications industry.

8—Distributed File Systems

A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer. When a user accesses a file on the server, the server sends the user a copy of the file, which is cached on the user's computer while the data is being processed and is then returned to the server.

Ideally, a distributed file system organizes file and directory services of individual servers into a global directory in such a way that remote data access is not location-specific but is identical from any client. All files

are accessible to all users of the global file system and organization is hierarchical and directory-based.



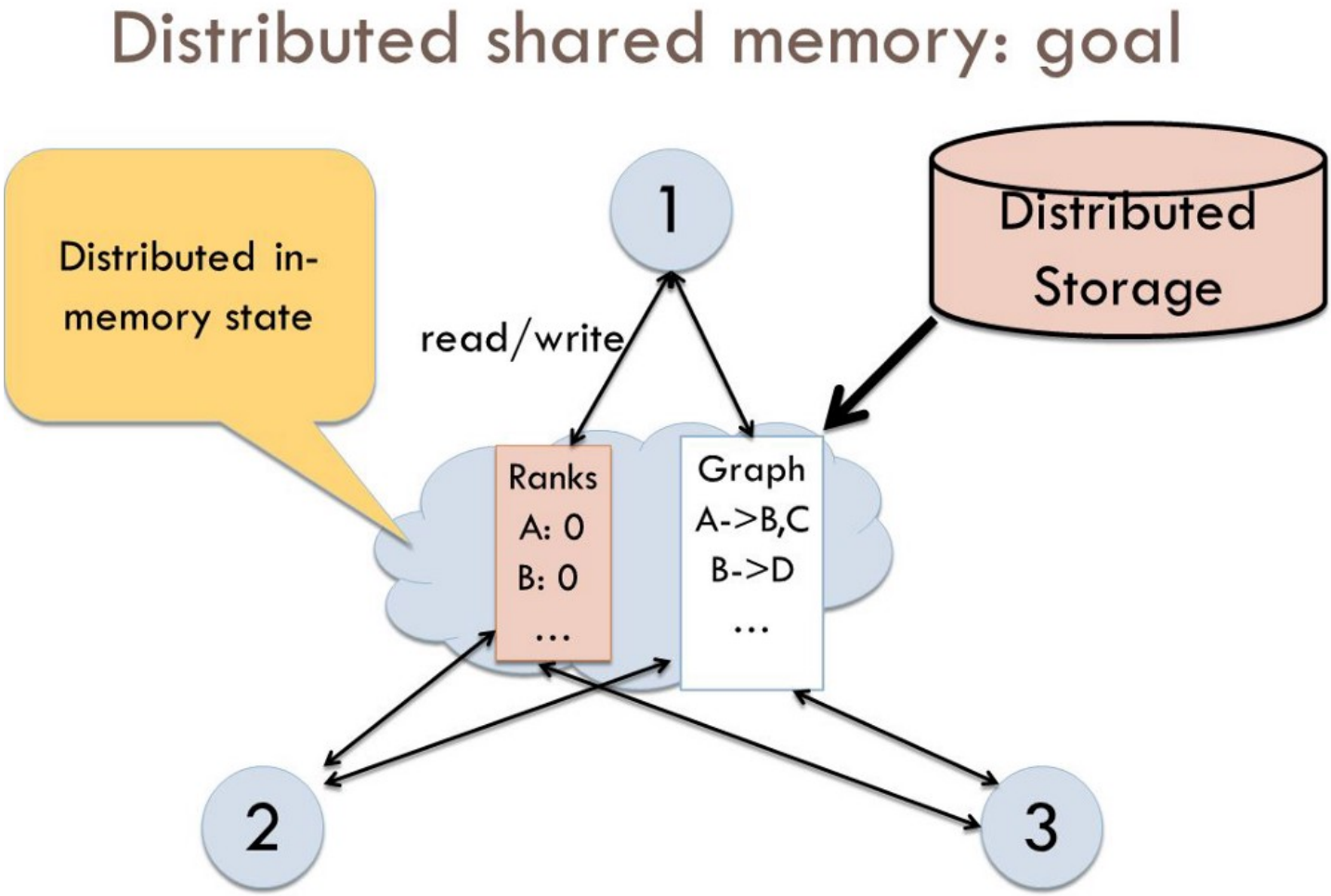
Since more than one client may access the same data simultaneously, the server must have a mechanism in place (such as maintaining information about the times of access) to organize updates so that the client always receives the most current version of data and that data conflicts do not arise. Distributed file systems typically use file or database replication (distributing copies of data on multiple servers) to protect against data access failures.

Sun Microsystems' Network File System (NFS), Novell NetWare, Microsoft's Distributed File System, and IBM's DFS are some examples of distributed file systems.

9—Distributed Shared Memory

Distributed Shared Memory (DSM) is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory provides a virtual address space that is shared among all computers in a distributed system.

In DSM, data is accessed from a shared space similar to the way that virtual memory is accessed. Data moves between secondary and main memory, as well as, between the distributed main memories of different nodes. Ownership of pages in memory starts out in some pre-defined state but changes during the course of normal operation. Ownership changes take place when data moves from one node to another due to an access by a particular process.



Advantages of Distributed Shared Memory:

- Hide data movement and provide a simpler abstraction for sharing data. Programmers don't need to worry about memory transfers between machines like when using the message passing model.
- Allows the passing of complex structures by reference, simplifying algorithm development for distributed applications.

- Takes advantage of “locality of reference” by moving the entire page containing the data referenced rather than just the piece of data.
- Cheaper to build than multiprocessor systems. Ideas can be implemented using normal hardware and do not require anything complex to connect the shared memory to the processors.
- Larger memory sizes are available to programs, by combining all physical memory of all nodes. This large memory will not incur disk latency due to swapping like in traditional distributed systems.
- Unlimited number of nodes can be used. Unlike multiprocessor systems where main memory is accessed via a common bus, thus limiting the size of the multiprocessor system.
- Programs written for shared memory multiprocessors can be run on DSM systems.

There are two different ways that nodes can be informed of who owns what page: invalidation and broadcast. Invalidation is a method that invalidates a page when some process asks for write access to that page and becomes its new owner. This way the next time some other process tries to read or write to a copy of the page it thought it had, the page will not be available and the process will have to re-request access to that page. Broadcasting will automatically update all copies of a memory page when a process writes to it. This is also called write-update. This method is a lot less efficient more difficult to implement because a new value has to sent instead of an invalidation message.

10—Cloud Computing

More and more, we are seeing technology moving to the cloud. It’s not just a fad—the shift from traditional software models to the Internet has steadily gained momentum over the last 10 years. Looking ahead, the next decade of cloud computing promises new ways to collaborate everywhere, through mobile devices.

So what is cloud computing? Essentially, cloud computing is a kind of outsourcing of computer programs. Using cloud computing, users are able to access software and applications from wherever they need, while it is being hosted by an outside party—in “the cloud.” This means that they do not have to worry about things such as storage and power, they can simply enjoy the end result.

Traditional business applications have always been very complicated and expensive. The amount and variety of hardware and software required to run them are daunting. You need a whole team of experts to install, configure, test, run, secure, and update them. When you multiply this effort across dozens or hundreds of apps, it isn't easy to see why the biggest companies with the best IT departments aren't getting the apps they need. Small and mid-sized businesses don't stand a chance.

With cloud computing, you eliminate those headaches that come with storing your own data, because you're not managing hardware and software—that becomes the responsibility of an experienced vendor like Salesforce and AWS. The shared infrastructure means it works like a utility: you only pay for what you need, upgrades are automatic, and scaling up or down is easy.

Cloud-based apps can be up and running in days or weeks, and they cost less. With a cloud app, you just open a browser, log in, customize the app, and start using it. Businesses are running all kinds of apps in the cloud, like customer relationship management (CRM), HR, accounting, and much more.

As cloud computing grows in popularity, thousands of companies are simply rebranding their non-cloud products and services as “cloud computing.” Always dig deeper when evaluating cloud offerings and keep in mind that if you have to buy and manage hardware and software, what you’re looking at isn’t really cloud computing but a false cloud.

Last Takeaway

As a software engineer, you will be part of a larger body of computer science, which encompasses hardware, operating systems, networking, data management and mining, and many other disciplines. The more engineers in each of these disciplines understand about the other disciplines, the better they will be able to interact with those other disciplines efficiently.

As the operating system is the “brain” that manages input, processing, and output, all other disciplines interact with the operating system. An understanding of how the operating system works will provide valuable insight into how the other disciplines work, as your interaction with those disciplines is managed by the operating system.