Operating Systems

Unit-I : Introduction

- **Introduction to Operating System:** Virtualizing the CPU, Virtualizing Memory, Concurrency, Persistence, Design Goals, Some History

- **The Process:** Process abstraction, System calls for Process management, Process Creation: A Little More Detail, Process States, Data Structures, Process execution mechanisms Process API, Process Control and Users, Useful Tools.

Unit-II: Scheduling

- Scheduling Workload Assumptions, Scheduling Metrics, First In, First Out (FIFO), Shortest Job First (SJF), Shortest Time-to-Completion First (STCF),

- A New Metric: Response Time, Round Robin, Incorporating I/O, The Multi-Level Feedback Queue, The Priority Boost, Attempt, Better Accounting, Multiprocessor Scheduling,

- Synchronization, Cache Affinity, Single-Queue Scheduling Multi-Queue Scheduling, Linux Multiprocessor Schedulers

Unit-III: Address Spaces

- Early Systems, Multiprogramming and Time Sharing, The Address Space, Memory API: Types of Memory, The malloc() Call, The free() Call, Common Errors, Underlying OS Support, Segmentation, Fine-grained vs. Coarse-grained Segmentation, Free-Space Management,

- Paging, A Memory Trace, Faster Translations (TLBs), TLB Basic Algorithm, Example: Accessing An Array, Who Handles The TLB Miss?, TLB Issue: Context Switches, Replacement Policy,

- Hybrid Approach: Paging and Segments, Beyond Physical Memory: Mechanisms, Swap Space, The Present Bit, The Page Fault, What If Memory Is Full?, Page Fault Control Flow, When Replacements Really Occur, The Linux Virtual Memory System

Unit-IV : Concurrency

- Shared Data, Uncontrolled Scheduling, The Wish For Atomicity, Waiting For Another, Thread API : Why Use Threads?, Thread Creation, Thread Completion Locks : The Basic Idea, Pthread Locks, Building A Lock, Evaluating Locks, Controlling Interrupts, Failed Attempt: Just Using Loads/Stores, Building Working Spin Locks with Test-And-Set, Compare-And-Swap, Load Linked and Store-Conditional, Fetch-And-Add,

- Different OS, Different Support, Semaphores: A Definition, Binary Semaphores (Locks) Semaphores For Ordering, The Producer/Consumer (Bounded Buffer) Problem, Reader-Writer Locks The Dining Philosophers, How To Implement Semaphores, Common Concurrency Problems.

Unit V: I/O Devices

- System Architecture, A Canonical Device, The Canonical Protocol, Lowering CPU Overhead With Interrupts, More Efficient Data Movement With DMA, Methods Of Device Interaction, Fitting Into The OS: The Device Driver,

- Case Study: A Simple IDE Disk Driver, Hard Disk Drives, Redundant Arrays of Inexpensive Disks (RAIDs),Files and Directories, Locality and The Fast File System, File System Implementation, Flash-based SSDs.

Unit-VI: Advanced topics in OS

- Data Integrity and Protection: Disk Failure Modes, Handling Latent Sector Error,
- Detecting Corruption: The Checksum, Using Checksums, Performance evaluation of computer systems, load testing, Little's law,
- Distributed Systems, Sun’s Network File System (NFS), The Andrew File System (AFS),
- Case Studies of: The xv6 operating system, The Linux Operating Systems.

Text Books

- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau “Operating Systems: Three Easy Pieces”, Arpaci-Dusseau Books, March , 2015

- Stallings William., "Operating Systems", Fourth Edition, Prentice Hall of India,2001

Three Pieces

- Virtualization
- Concurrency
- Persistence

- Virtualization: CPU
  - Assume there is one physical CPU in a system (though now there are often two or four or more).
  - What virtualization does is **take that single CPU and make it look like many virtual CPUs to the applications running on the system**.
  - Thus, while each application thinks it has its own CPU to use, there is really only one.
  - And thus the OS has created a illusion: it has virtualized the CPU

- Virtualization: Memory
  - OS will give each program the view that **it has a large contiguous address space to put its code and data into**;
  - thus, as a programmer, you never have to worry about things like “where should I store this variable?”
  - because the virtual address space of the program is large and has lots of room for that sort of thing.
  - Life, for a programmer, becomes much more tricky if you have to worry about fitting all of your code data into a small, crowded memory.

- **Concurrency:**
  - There are certain types of programs that we call multi-threaded applications.
  - Each thread is kind of like an independent agent running around in this program, doing things on the program's behalf.
  - But these threads access memory, If we don't coordinate access to memory between threads, the program won't work as expected.
  - **Concurrency is the execution of several instruction sequences at the same time.**
  - **In an operating system, this happens when there are several process threads running in parallel.**
  - These threads may communicate with each other through either shared memory or message passing.

- Persistence :
  - Making information persist, despite computer crashes, disk failures, or power outages is a tough and interesting challenge.

Unit-I

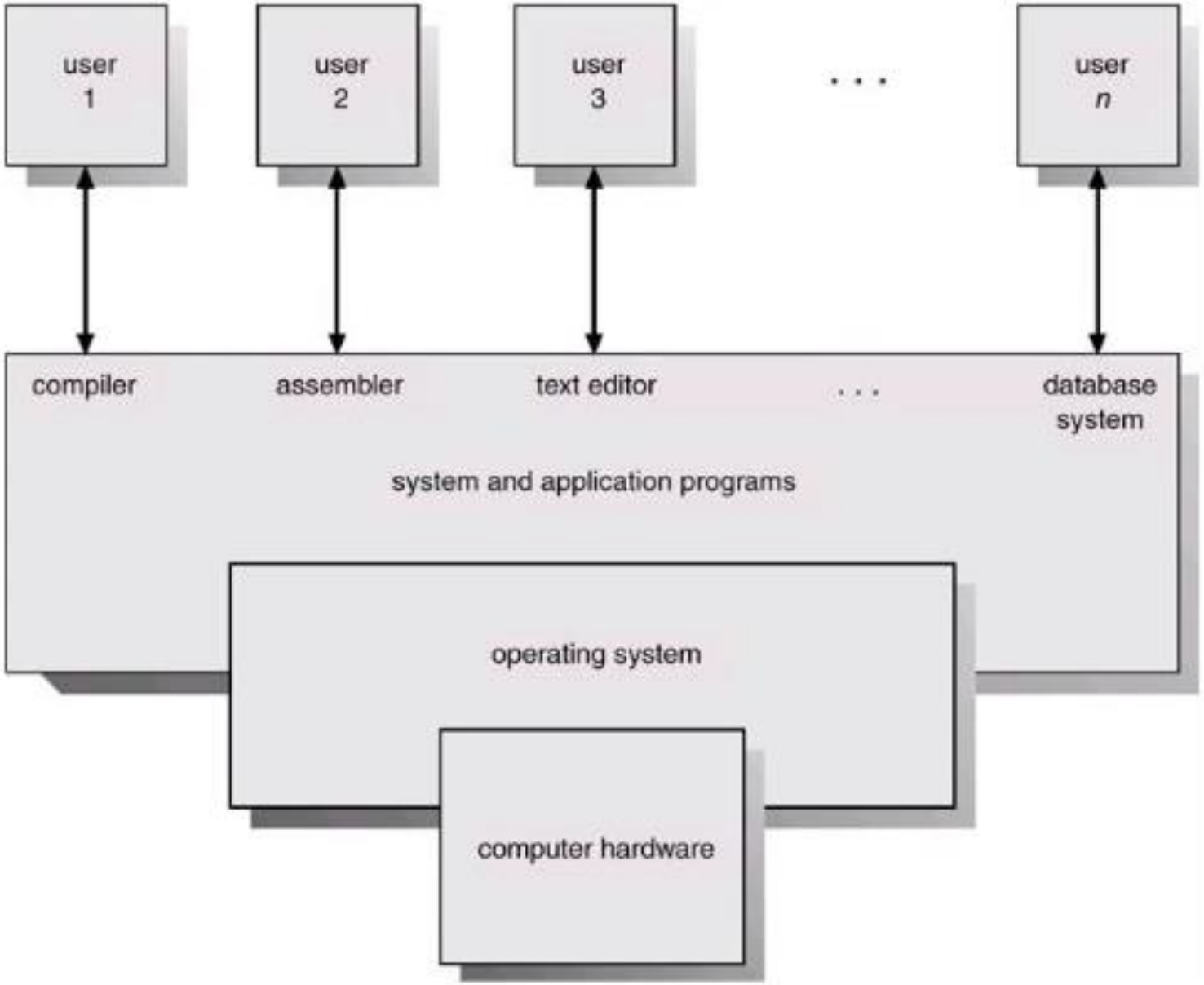
- **Introduction to Operating System:** Virtualizing the CPU, Virtualizing Memory, Concurrency, Persistence, Design Goals, Some History

- **The Process:** Process abstraction, System calls for Process management, Process Creation: A Little More Detail, Process States, Data Structures, Process execution mechanisms Process API, Process Control and Users, Useful Tools.

What is Operating System?

- It is a middleware between user program and System Hardware.
  - Users use computers to run various programs like browser, word, or any others.
  - These programs run on the underlying hardware (CPU, MM, Disk, IO Devices etc)
- Resource Manager-It manages Hardware, CPU, Main Memory, IO Devices(Disks, Network Card, Keyboard etc)
- Provides a platform on which other application programs installed

Abstract view of System Components



What happens when you run the program? (Background)

- A compiler translates high level program into an executable (“.c” to “a.out”)
- **The exe contains the instructions** that the CPU can understand**, and the data** (variables) of the program (all numbered with addresses)
- These instructions run on CPU:
  - The CPU runs instructions corresponding to user or OS code.
  - Every CPU has a certain instruction set architecture.
  - **Compiler translates the program in to instruction set** which can be executed on particular CPU.
- CPU also consists of few registers.
  - Example:
    - Pointer to current instruction (Program counter or PC)
    - Operands of instructions, Memory Address

What happens when you run the program?

- To run an exe, CPU
  - Fetches instructions pointed at by PC from memory
  - Loads data required by the instruction into registers
  - Decodes and executes the instructions
  - Stores results to memory

- Most recently used instructions and data are in CPU caches for faster access
  - Instruction Cache
  - Data Cache

So, What does the OS do?

- OS manages program memory (It sets up the memory for execution)
  - Loads program executable (Code, Data) from disk to memory
  - Ex: fetching a.out from disk to memory

- OS manages CPU (it sets up the CPU for execution)
  - Initialize program counter (PC) and other registers to begin execution.
  - OS sets correct values to PC and registers. PC pointing to current instruction.

- OS manages external devices
  - Read/write files from disk

OS manages CPU

- OS provides the process abstraction
  - Process: a running program (Ex: a.out becomes the process)
  - OS creates and manages processes.

- Each process has the illusion of having the complete CPU ie, OS virtualizes CPU. (but many processes are running at time ex: we are listening to music while browsing the web)

- Timeshares CPU between processes.

- Enables coordination between processes

OS manages memory

- OS manages the memory of the process:
  - Memory of the process contains code, data, stack, heap etc.
  - Stack : function call , arguments to the functions, return address from function
  - Heap: ex. malloc

- Each process thinks it has a dedicated memory space for itself,
  - It gives addresses to code, data, stack, heap starting from 0 (virtual addresses)
  - But in fact all of these memory images could be placed in memory at different (random) pieces.
  - RAM could be shared among multiple processes.

- OS abstracts out the details of the actual placement in memory, translates from virtual addresses to actual physical addresses (OS virtualizes memory)

- CPU registers to MM to Disk, as we go down the list, access delay increases, Size increases, and cost decreases.

OS manages devices

- OS has code to manage disk, network card, and other external devices: device drivers.

- OS has device drivers.

- Device driver talks the language of the hardware devices
  - Issues instructions to devices(fetch data from a file)
  - Responds to interrupt events from devices(user has pressed a key on keyboard)

- OS manages file system:
  - Persistent data organized as a filesystem on disk

- When running a process,
  - The PC of the CPU points to some address in the memory image of the process.
  - The CPU also generates requests to read/write data in the memory image.

- How does CPU knows the address of the memory image of a process in physical memory?
  - CPU does not need to know the actual physical address in the memory.
  - Instead, every process has a virtual address space.
  - These virtual addresses requested by the CPU are converted to actual physical address(where the program is stored) before the addresses are seen by the physical memory hardware.
  - Actual address translation is offloaded to a hardware MMU.

Design goals of an Operating System

- Convenience, abstraction of hardware resources for user programs.
- Efficiency of usage of CPU, memory etc.
- Isolation between multiple processes

History of OS

- Started out as a library to provide common functionality across programs.
- Later, evolved from procedure call to system call.
- When a system call is made to run OS code, the CPU executes at a higher privilege level.
- Evolved from running a single program to multiple processes concurrently.

In starting mainframe computers…

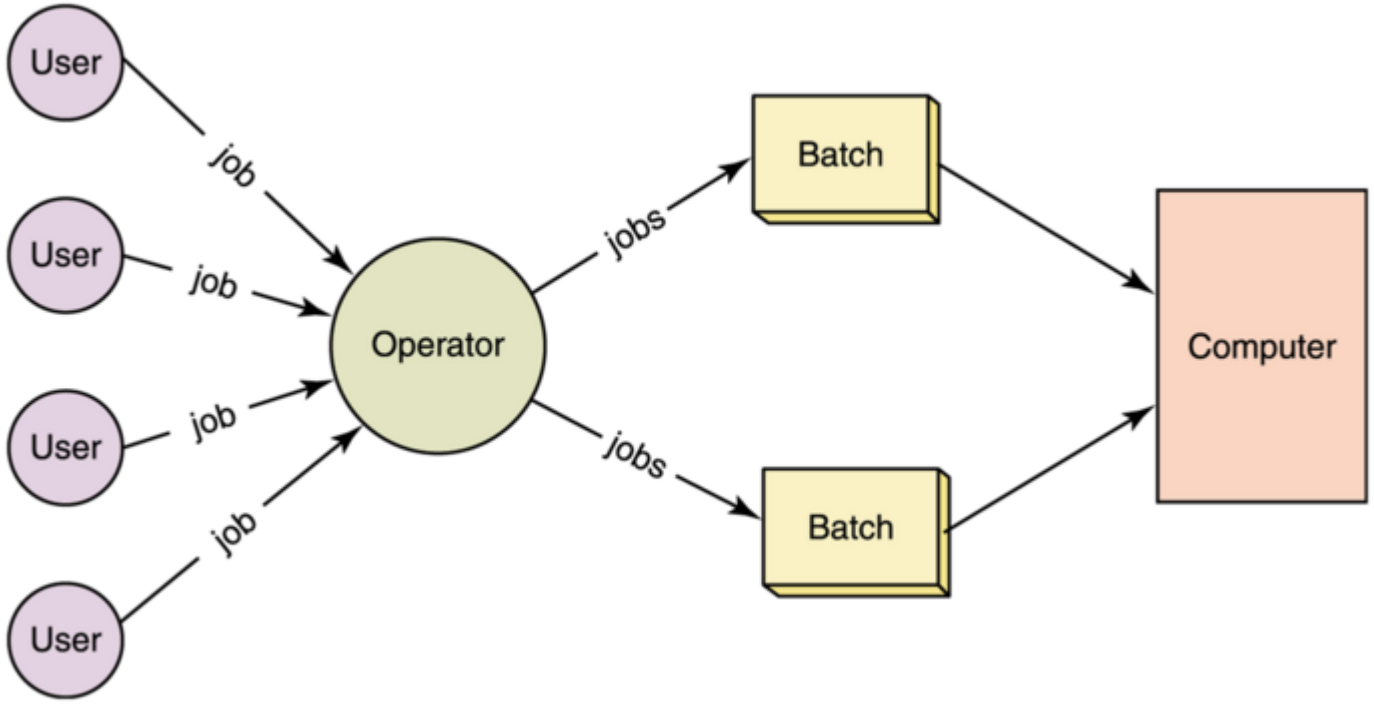
- Common input output devices were card readers and tape drives.
- User prepare a job which consists of program , i/o data and control instructions.
- Input job is given in the form of punch cards and results also appear in the form of punch card
- So, OS was very simple, always present in memory and major task is to transfer the control form one job to another

Batch Processing

- Jobs with similar needs are batched together and executed through the processor as a group.

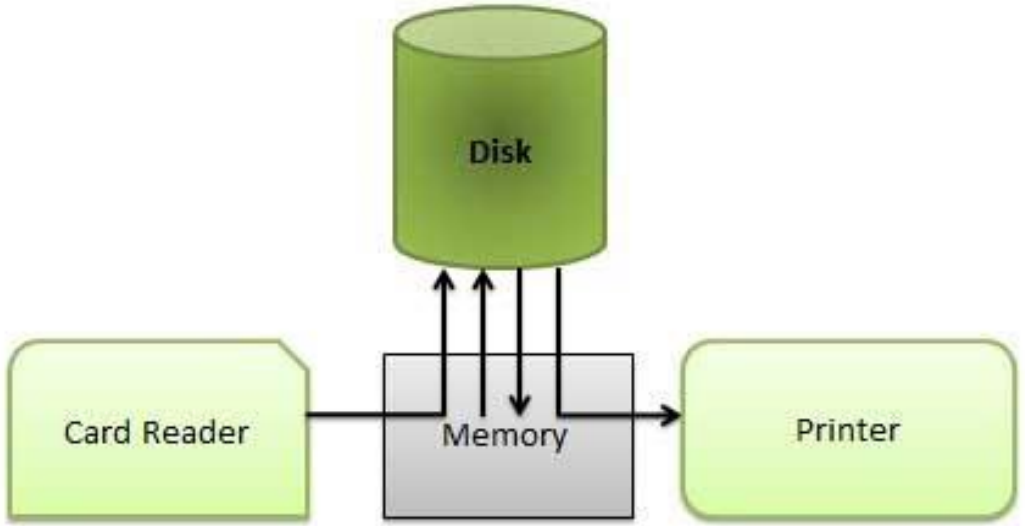
- Operator sort jobs as a deck of punch cards into batch with similar needs.
  - Ex. COBOL batch

- Advantages:
  - In a batch, job executes one after another saving time form activities like loading complier.
  - During batch execution no manual intervention is needed
- Disadvantages:
  - memory limitation
  - interaction of input and output devices directly with CPU.



SPOOLING

- Simultaneous peripheral operations online
- Input and output devices are relatively slow compared to CPU.
- In spooling, data is stored first onto the disk and then CPU interact with disk via MM.



- Spooling Advantages:
  - No interaction with input and output devices with CPU
  - CPU utilization is more.

- Disadvantages:
  - In starting, spooling was uniprogramming

Multiprogramming OS

- **Multiprogramming means more than one process in main memory which are ready to execute.**
- Maximize CPU utilization
- Process generally require CPU time and I/O time.
  - So, if running process perform I/O or some other event which do not require CPU,
  - then instead of sitting idle,
  - CPU make a context switch and picks some other process and its idea will continue.
- CPU never idle unless, there is no process ready to execute.

Multiprogramming OS

- Advantages:
  - High CPU utilization.
  - Less response time for the process.
  - May be extended to multiple users

- Disadvantages:
  - Difficult scheduling
  - MM management is required

Multitasking OS/Time Sharing/ multiprogramming with RR

- Multitasking is **multiprogramming with time sharing.**
- **Only one CPU but switches between processes so quickly** that it gives illusion that all executing at same time.
- The task in multitasking may refer to multiple threads of the same program.
- Main idea is **better response time and executing multiple process together.**

Multiprocessing OS

- Two or more CPU with a single computer in close communication sharing the system bus, memory an other I/O devices.
- Different process may run of different CPU, true parallel execution.
- Types of multiprocessing
  - Symmetric: One OS control all CPU, each CPU has equal rights.
  - Asymmetric: Master slave architecture, system has take on one processor and application on other as one CPU.
    - One processor is dedicated for : I/O devices and hardware interrupts
    - Remaining processor : used for increasing computations

- Advantages
  - Increased throughput (Throughput: per unit time completed processes)
  - Increased reliability (graceful degradation: Even if one processor is not working, still your work will get executed)
  - Cost saving (Memory and other devices are not increased)
  - Battery efficient (when there is less load, only one core will be working)
  - True parallel processing

- Disadvantages:
  - More complex
  - Overhead or coupling reduce throughput (if we try to use other devices parallel, may reduce the speed)
  - Large main memory.

The Process Abstraction

OS provides process abstraction

- When you run an exe file, the OS creates a process = a running program.
- **OS timeshares CPU across multiple processes: virtualizes CPU.**
- **OS has a CPU scheduler that picks one of the many active processes to execute on a CPU.**
- Scheduler has two parts:
  - Policy: which process to run, among active processes.
  - Mechanism: how to “context switch” between processes

What constitutes a process?

- Every process created in the system has a unique identifier (PID)

- Process has Memory image
  - Code and data (static)
  - Stack and heap(dynamic)

- Process has CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer

- File descriptors :
  - Pointers to open files and devices
  - process is writing something on the screen or taking input from keyboard (STDOUT, STDIN)

How does OS create a process?

- Disk has a.out, OS takes the chunk of memory and allocates it to the process.

- Allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap

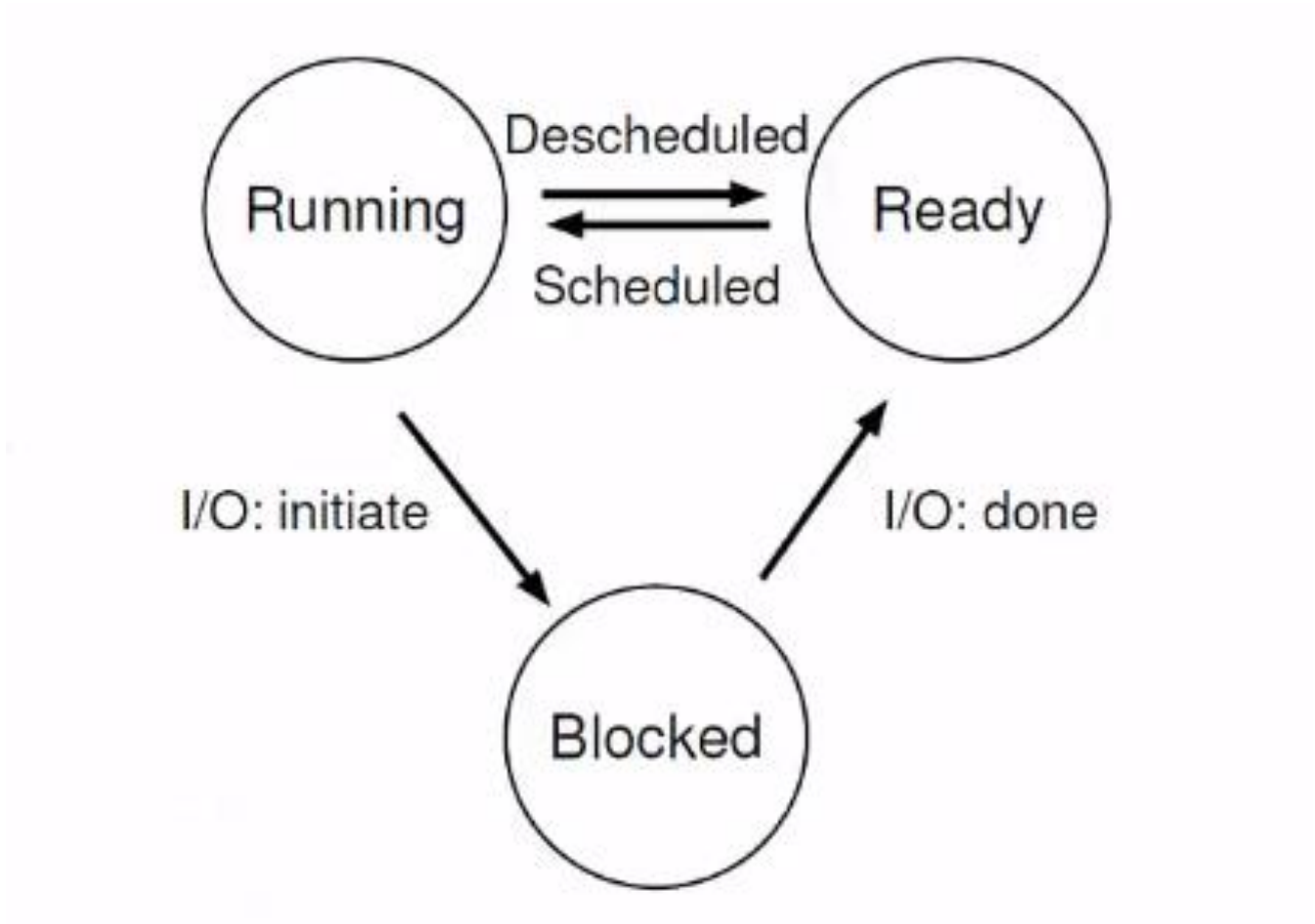
- OS opens basic files
  - STD IN, OUT, ERR

- Initialize CPU registers
  - PC points to first instruction

States of Process

- Process exists in different states:
- **Running**: currently executing on CPU
- **Ready:** Waiting to be scheduled, ready to run but not yet run/ the process is waiting to be assigned to the processor.
- **Blocked:** suspended, not ready to run
  - Process is waiting for some event to occur (such as I/O completion)
- New: being created, yet to run
- Dead: terminated : The process has finished execution.

Process State Transition



Example: Process States

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Tracing Process State: CPU and I/O

Note: Once your I/O operation is done, it jumps from blocked state to ready state, not directly jump to cpu.

How OS keeps track of all of these processes?

OS Data structures

- How OS keeps track of all of these processes?
- OS maintains a data structures (eg. List) of all active processes.
- Each element of the list is called the PCB.
- Each process in the OS is represented by PCB or Task control Block
- List of PCB is maintained to track processes in the system
- **Information about each process is stored in a process control block(PCB)**
  - Process identifier
  - Process state
  - Pointers to other related processes(parent)
  - CPU context of the process (saved when the process is suspended)
  - Pointers to memory locations
  - Pointers to open files
  - Accounting information (resources used by particular process)

PROCESS API

What API does the OS provide the user program?

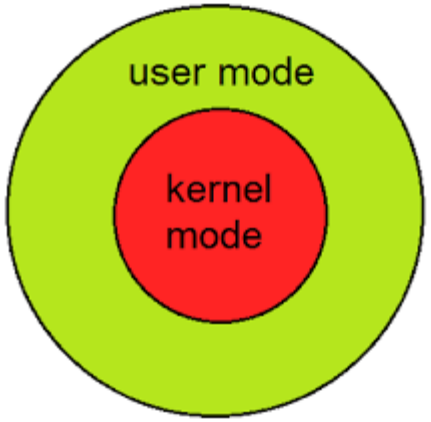
- **Operating system provides API to user programs in order to create and mange processes.**

- **API=Application Programing interface**

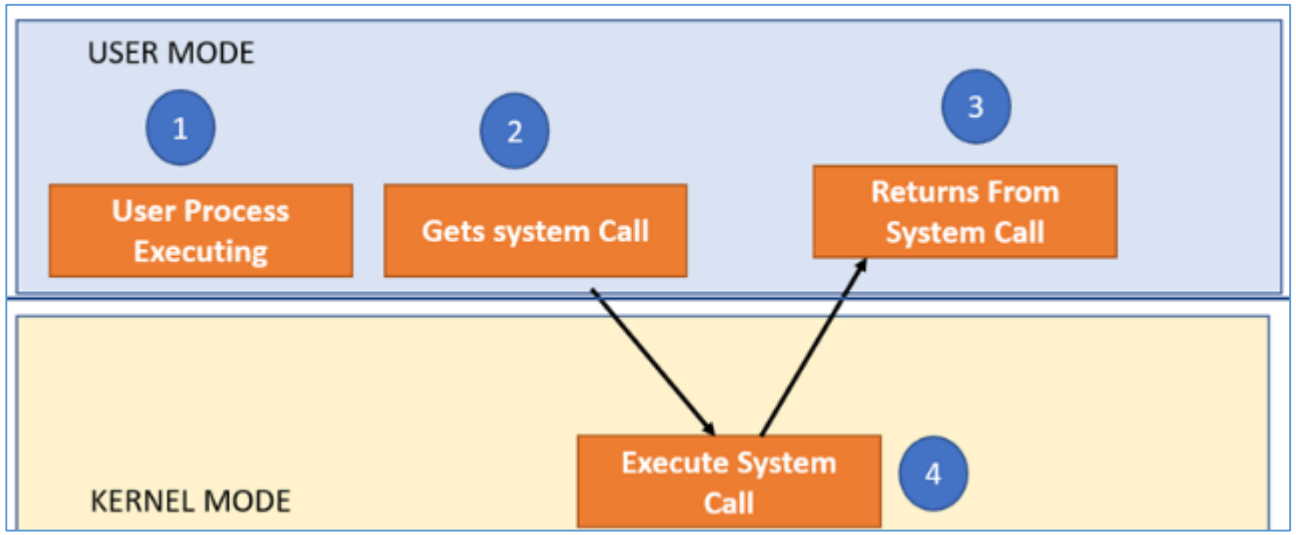
  Function available to users to write user programs

- **API provided by OS is basically a set of “system calls”**
  - System call is function call into OS code **that runs at a higher privilege level of the CPU**
  - Sensitive operations (eg, access to hardware) are allowed only at a higher privilege level.
    - Reading form disk
    - Accessing the memory
  - Some “blocking” system calls cause the process to be blocked and descheduled (eg. Read from disk)

System Calls



- **Program can execute in two modes of operations**
  - User mode and Kernel mode.
  - If the program is running in user mode, then that program does not have access to hardware.
  - If the program is running in kernel mode, then that program is having access to memory, hardware and resources
  - **Kernel mode is privileged mode**.
- When program is running in user mode, and if it needs to access the hardware resource, then it has to run in kernel mode, this switching is called context switching.
- **System call is the programmatic way, in which a computer program requests a service from the kernel of the OS.**
- These calls are generally available as routines written in C and c++.



- **Step 1)** The processes executed in the user mode till the time a system call interrupts it.
- **Step 2)** After that, the system call is executed in the kernel-mode on a priority basis.
- **Step 3)** Once system call execution is over, control returns to the user mode.,
- **Step 4)** The execution of user processes resumed in Kernel mode.

Example: System Calls

- Copying contents of one file (source file) to the other (Destination file).

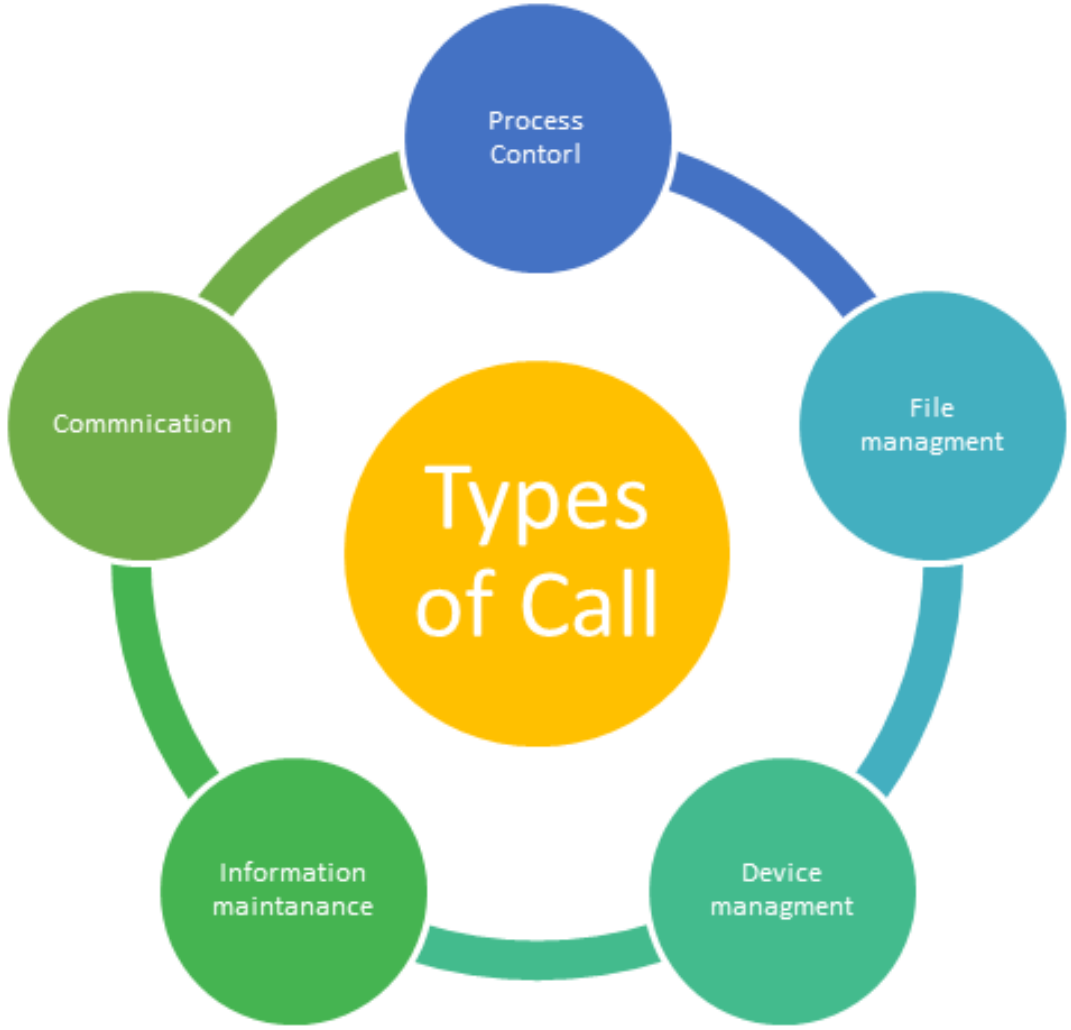
Acquire input filename
Write prompt to screen
Accept input
Acquire output filename
Write prompt to screen
Accept input
Open input file
If file doesn't exist, ABORT
Create Output file
If file exists, ABORT
Read form Input file
Write to Output file
Close output file
Write completion message to screen
Terminate normally

Why do you need System Calls in OS?

- Following are situations which need system calls in OS:
  - Reading and writing from files demand system calls.
  - If a file system wants to create or delete files, system calls are required.
  - System calls are used for the creation and management of new processes.
  - Network connections need system calls for sending and receiving packets.
  - Access to hardware devices like scanner, printer, need a system call.

Types of System Calls

- Process Control
- File Management
- Device Management
- Information maintenance
- Communication



Process Control

- This system calls perform the task of process creation, process termination, etc.

- Functions:
  - End and Abort
  - Load and Execute
  - Create Process and Terminate Process
  - Wait and Signed Event
  - Allocate and free memory

File Management

- File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

- Functions:
  - Create a file
  - Delete file
  - Open and close file
  - Read, write, and reposition
  - Get and set file attributes

Device Management

- Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

- Functions
  - Request and release device
  - Logically attach/ detach devices
  - Get and Set device attributes

Information Maintenance

- It handles information and its transfer between the OS and the user program.

- Functions:
  - Get or set time and date
  - Get process and device attributes

Communication:

- These types of system calls are specially used for interprocess communications.

- Functions:
  - Create, delete communications connections
  - Send, receive message
  - Help OS to transfer status information
  - Attach or detach remote devices

So, Should we rewrite programs for each OS?

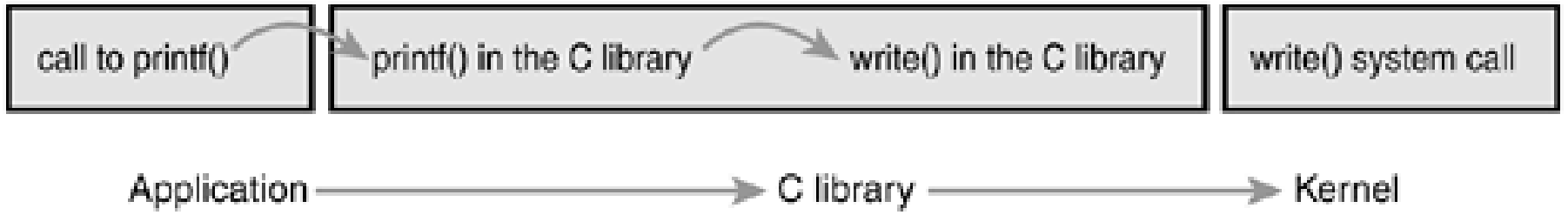
- Do we rewrite the program when we go from Linux to Windows?

- POSIX API: a standard set of system calls that an OS must implement
  - Portable Operating System Interface (**POSIX**)
  - Programs written to the POSIX API can run on any POSIX compliant OS (POSIX comprises a series of standards from the IEEE)
  - Most modern OSes are POSIX compliant
  - Ensures program portability

- **User never invokes the system calls directly.**

- Program language libraries hide the details of invoking system calls
  - The *printf* function in the C library, calls the write system call, to write to screen
  - User programs usually do not need to worry about invoking system calls

The relationship between applications, the C library, and the kernel with a call to `printf()`.



Process related system calls (in UNIX)

- **What are the system calls related to create and managing processes:**

- **fork()** creates a new child process
  - All processes are created by forking from a parent (every process is created from another process)
  - When OS starts, it creates **init** process. It is a ancestor of all processes.

- **exec()** makes a process execute a given executable.

- **exit()** terminates a process

- **wait()** causes a parent to block until child terminates

- Many variants exist of the above system calls with different arguments.

Categories	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	loctl() read() write()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() write() close!)
Information maintanence	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup ()	Chmod() Umask() Chown()

What happens during a fork?

- A new process is created by making a copy of parent's memory image. Ex: P is a parent process and C is a child process.
- The new process is added to the OS process list and scheduled, So that OS can keep track of these processes.
- Parent and child start executing just after fork (with return different values). Sometimes P gets the CPU, sometimes C gets the CPU.
- Parent and child execute and modify the memory data independently.

Waiting for children to die..

- When a process terminates what happens?

- Process termination scenarios:
  - By calling exit() (exit is called automatically when end of main is reached)
  - OS terminates a misbehaving process.

- Terminated processes are not removed instantly from list. Terminated process **exists as zombie.**

- **When these zombie process cleared out.**
  - When a parent calls wait(), zombie child is cleaned up or “reaped”
  - The parent process calls wait() to delay its execution until the child finishes executing.

- What if parent terminates before child? init process adopts orphans and reaps them.

- All the processes whose parents are not around are cleaned by init process.

Example: fork

```
int main()
{

   // make two process which
run same program after this
instruction
   fork();

   printf("Hello world!\n");
   return 0;
}
```

```
Hello world!
Hello world!
```

```
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, $2^3 = 8$

- So there are total eight processes (new child processes and one original process).
- If we want to represent the relationship between the processes as a tree hierarchy it would be the following:
- The main process: P0
  Processes created by the 1st fork: P1
  Processes created by the 2nd fork: P2, P3
  Processes created by the 3rd fork: P4, P5, P6, P7

```
void forkexample()
{
   // child process because return value zero
   if (fork() == 0)
      printf("Hello from Child!\n");

   // parent process because return value non-zero.
   else
      printf("Hello from Parent!\n");
}
int main()
{
   forkexample();
   return 0;
}
```

Hello from Child!

Hello from Parent!

OR

Hello from Parent!

Hello from Child!

fork() returns 0 in the child process and positive integer in the parent process.

Here, two outputs are possible because the parent process and child process are running concurrently. So we don’t know whether the OS will first give control to the parent process or the child process.

```
void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

```
void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Parent has x=0

Child has x=2

OR

Child has x=2

Parent has x=0

Parent process and child process are running the same program, but it does not mean they are identical.

OS allocate different data and states for these two processes, and the control flow of these processes can be different.

Fork() and process id: example

```
main()
{
printf (“Hello world (pid: %d)” getpid());
int rc=fork();
if(rc<0)  //fork failed
{
        printf (“fork failed”);
        exit(1)
}
else if (rc==0)
{
        printf(“I am child (pid: %d)” getpid());
}
else
{
        printf (“I am parent of %d (pid: %d)”, rc, getpid())
}
```

Hello world (pid: 6)

I am parent of 7 (pid:6) // parent receive the PID of newly-created child

I am child (pid: 7)

OR (output is non-deterministic)

Hello world (pid: 6)

I am child (pid: 7)

I am parent of 7 (pid:6)

Wait() example

```
main()
{
printf (“Hello world (pid: %d)” getpid());
int rc=fork();
if(rc<0)  //fork failed
{
        printf ( “fork failed”);
        exit(1)
}
else if (rc==0)
{
        printf(“I am child (pid: %d)” getpid());
}
else
{
        int rc_w =wait()
        printf (“I am parent of %d (pid: %d)”, rc, getpid())
}
```

- Parent calls wait to delay its execution until the child finishes executing
- When the child is done, wait() returns to its parent.
- Even parent runs first, it politely waits for the child to finish running, then wait() returns, then parent prints its message
- Adding wait() call, makes the output deterministic
- Output:

  Hello world (pid: 6)
  I am child (pid: 7)
  I am parent of 7 (pid:6)

What happens during exec?

- After fork, parent and child are running same code
  - Not too useful!

- What if the parents want that its child do something else, apart from its code
  - **A process can run exec() to load another executable to its memory image**
  - So, child can run a different program from parent.
  - Given the name of an executable and some arguments to exec(),
    - it loads from that executable and overwrites its current code segment with it,
    - the heap and stack and other parts of the memory space of the program are re-initialized.

Process Control

- The kill() system call sends a specified signal to a specified process, including directives to pause, die, and other useful imperatives.

- In UNIX shells, certain keystrokes combinations are configured to deliver a specific signal to the currently running process
  - Control-c sends SIGINT(interrupt) to the process
    - normally terminating it
  - Control-z sends SIGTSTP (stop) signal
    - thus pausing the process in mid execution (you can resume it later with command ex. fg is a built-in command fond in many shells)

Process Control and Users

- Which processes can be controlled by particular person is encapsulated in the notion of user
- The OS allows multiple users onto the system
- It ensures users can only control their own processes

Useful Tools

- Command line tools.
- Ps: allows you to see which processes are running
- Top: displays the processes of the system and how much CPU and other resources they are using.
- Kill: send arbitrary signal to processes

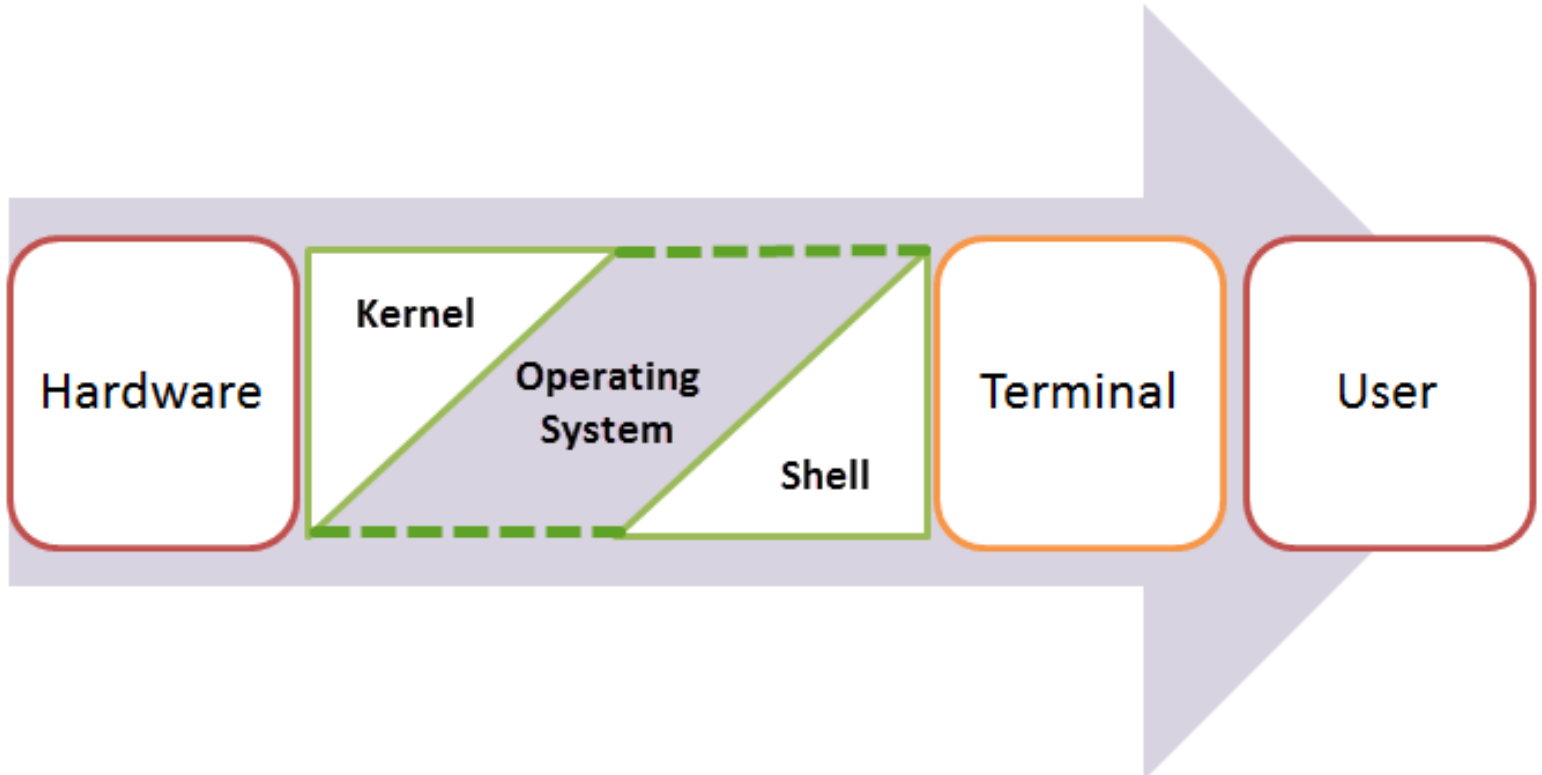
What is Kernel

- Kernel is a computer program, that is core of computers OS.
  - with complete control over everything in the system.

- It manages following recourses of the Linux system
  - File Management
  - Process Management
  - I/O management
  - Memory Management etc.

What is Shell

- A *shell* is a special user program which provide an interface to user to use OS services.
- *Software* living in kernel space can execute *privileged instructions*, such as dealing directly with hardware.
- Shell accept human readable commands form user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.
- The way the shell talks to the kernel is by system calls. These system calls allows the user to do things like *open files* and *create processes*.
- The shell gets started when the user logs in or start the terminal


Hardware
Kernel
Operating System
Shell
Terminal
User

Command line shell

- Shell accessed accessed by user using a command line interface.
- The special program called Terminal in linux or command prompt in windows OS is provides to type in the human readable commands and then it is being executed.
- The result is then displayed on the terminal.
- It is bit difficult for beginners, as it is hard to memorize so many commands.
- It is powerful, it allows user to store commands in a file and execute them together, using this repetitive task can be easily automated.

Graphical Shells

- It provides means for manipulating programs based on GUI,
  - by allowing for operations such as opening, closing, moving and resizing windows.

- User do not need to type in command for every action

Shells available in Linux System

- BASH(Bourne Again Shell):
  - Most widely used shell.
  - It is used as default login shell in Linux and macOS.
  - It can be installed on Windows OS.
- CSH (C Shell)
- KSH (Korn Shell)
- Each shell does the same job but understand different commands and provide different built in functions.

How does shell works?

- In a basic OS, the *init* process is created after initialization of hardware.
- The *init* process spawns a shell like bash
- Shell reads user command,
  - forks a child,
  - execs the command executable,
  - waits for it to finish, and
  - reads next command.
- Common commands like *ls* are all executables that are simply executed by the shell

prompt>ls

a.txt b.txt

…….You have your shell process, user has typed *ls*,

shell will *fork* the child process, then *exec* will be called on the child process, exec(ls),

once the child process finishes running *ls* it prints out the output (a.txt b.txt) and child process terminates

(shell calls wait until child process finishes) and it comes back to user for next command.

More about Shell

- Shell can manipulate the child in strange way.
- Suppose you want to redirect output from a command to a file
- Prompt>ls>abc.txt
  - Shell spawns a child, rewrites its standard output to a file, then calls exec on the child
  - It manipulates child for storing the output in abc.txt
  - And then it runs *exec*

Shell Script

- It executes bunch of commands.

- An shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid repetitive work.

These files are called shell scripts or shell programs.

- Shell scripts are similar to batch file in MS-DOS.

- Shell script is saved with .sh extension.

Shell script comprises

- Shell script comprises following elements.
- Shell kewords –if, else, break etc
- Shell commands- cd, ls, echo, touch, pwd etc
- Functions
- Control flow – if.. Then.. Else , case and shell loops etc

Why do we need shell script

- To avoid repetitive work and automation.
- System admins use shell scripting for routine backups
- System monitoring

Practical examples

- Data backups
- Find out what processes are eating up your system resources.
- Find out available and free memory.
- Find out all logged in users and what they are doing.
- Find out all failed login attempt.
- Find out information about local or remote servers.

Comments and Variables

- echo is used to print comments
  - Example: echo “Hello World”
- Variables: System Variables and User defined variables
- System Variables: $BASH, $BASH_VERSION, $PWD
- User defined variables
  - name= “Akshay”
  - echo name
  - value=10
  - echo $value

- Reading multiple variables
  - echo “Enter name”
  - read name1 name2 name3
- Enter input on same line
  - read –p
- Make input silent
  - Read -sp

- Saving input in array
  - echo “enter names”
  - Read –a names
  - echo “names ${names[0], $names[1]}”

- Reading without variable
  - Input goes into inbuilt variable REPLY
  - echo $REPLY

IF Statement

```
If [condition]
then
        Statement
fi
```

```
count =10
if [$count –eq 9]
then
echo “condition is true”
fi
```

int comparisons

- -eq is equal to
- -ne is not equal to
- -gt
- -ge
- -lt
- -le

Example:
[“$a” –eq “$b”]

- <, >, =, >= etc
- Example:
  ((“$a” < “$b”))

String Comparisons

- =, <, >, != etc
- Angle bracket with strings

```
word =a
If[[$word <“b”]]
Then
Echo “true”
```

If-elif-else

```
word=a
If[[$word ==“b”]]
        then
                echo “condition b is true”
elif [[$word ==“a”]]
        then
                echo “condition a is true”
else
                echo “condition is false”
fi
```

Looping Statements

- There are total 2 looping statements which can be used in bash programming
  - while statement
  - for statement

- To alter the flow of loop statements, two commands are used they are,
  - break
  - continue

While Statement

- command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated.

- Syntax:

```
While command
Do
    Statement to be executed
done
```

Example:

```
a=0
# -lt is less than operator

#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
   # Print the values
   echo $a

   # increment the value
   a=`expr $a + 1`
done
```

For statement

- Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words).

- Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

```
for var in word1 word2 ...wordn
do
  Statement to be executed
done
```

Example

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
   # if a is equal to 5 break the loop
   if [ $a == 5 ]
   then
      break
   fi
   # Print the value
   echo "Iteration no $a"
done
```

```
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
```

```
for a in 1 2 3 4 5 6 7 8 9 10
do
   # if a = 5 then continue the loop and
   # don't move to line 8
   if [ $a == 5 ]
   then
      continue
   fi
   echo "Iteration no $a"
done
```

```
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
Iteration no 6
Iteration no 7
Iteration no 8
Iteration no 9
Iteration no 10
```

Case Statement

- case statement works as a switch statement if specified value match with the pattern.
- It will execute a block of that particular pattern.
- When a match is found all of the associated statements until the double semicolon (;;) is executed.
- A case will be terminated when the last command is executed.
  If there is no match, the exit status of the case is zero.

Case statement

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
    #case 1
    "mercedes") echo "Headquarters - Germany" ;;

    #case 2
    "audi") echo "Headquarters - US" ;;

    #case 3
    "bmw") echo "Headquarters – India" ;;
esac
```

Grep command

- The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern.

- Syntax: grep [option] pattern [files]

- **Options Description**
  - **-c** : This prints only a count of the lines that match a pattern
  - **-h :** Display the matched lines, but do not display the filenames.
  - **-i :** Ignores, case for matching
  - **-l :** Displays list of a filenames only.
  - **-n :** Display the matched lines and their line numbers.
  - **-v :** This prints out all the lines that do not matches the pattern
  - **-e exp :** Specifies expression with this option. Can use multiple times.
  - **-f file :** Takes patterns from file, one per line.
  - **-E :** Treats pattern as an extended regular expression (ERE)
  - **-w :** Match whole word
  - **-o :** Print only the matched parts of a matching line,
  - with each such part on a separate output line.

```
$cat > geekfile.txt
```

```
unix is great os.
unix is opensource.
unix is free os.learn operating system.
Unix linux which one you choose.
uNix is easy to learn.unix is a multiuser os. Learn unix . unix is a powerful.
```

Case insensitive search:

- The –i option enables to search for a string case insensitive in the given file.
- $ grep –i “UNix” file.txt

```
unix is great os.
unix is open source.
unix is free os.
learn operating system.
Unix linux which one you choose.
uNix is easy to learn. unix is a multiuser os. Learn unix . unix is a powerful.
```

```
unix is great os.
unix is open source.
unix is free os.
Unix linux which one you choose.
uNix is easy to learn. unix is a multiuser os.Learn unix .unix is a powerful.
```