System Programming Basic

On Linux

Prayas Mohanty (Red Hat Certified Instructor)

Red Hat Certification ID: 100-005-594

Objective

- What is Kernel Mode
- What is User mode
- Difference between Kernel Mode & User Mode
- What is System Calls?
- Why System calls?
- Type of System Calls
- How to execute System Call
- Debug system call using strace command.

Prerequisite of Participants

- Having Knowledge on Linux Platform
- Having familiarity with Linux Command line
- Basic Knowledge on vi Editor
- Proper Knowledge on C programming
- Basic Knowledge on File Handling in C

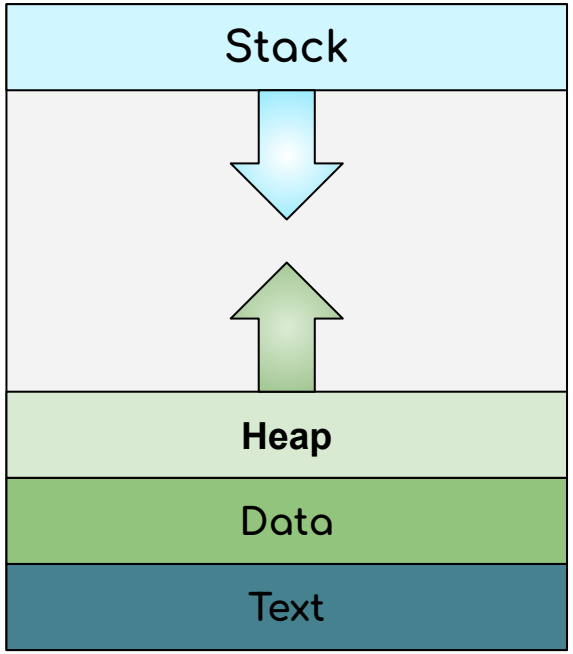
What is system Programming

- Systems programming, or system programming, is the activity of programming computer system software.
- The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software.
- Systems programming requires a great degree of hardware awareness. Its goal is to achieve efficient use of available resources, either because the software itself is performance critical or because even small efficiency improvements directly transform into significant savings of time or money.

Before we Start

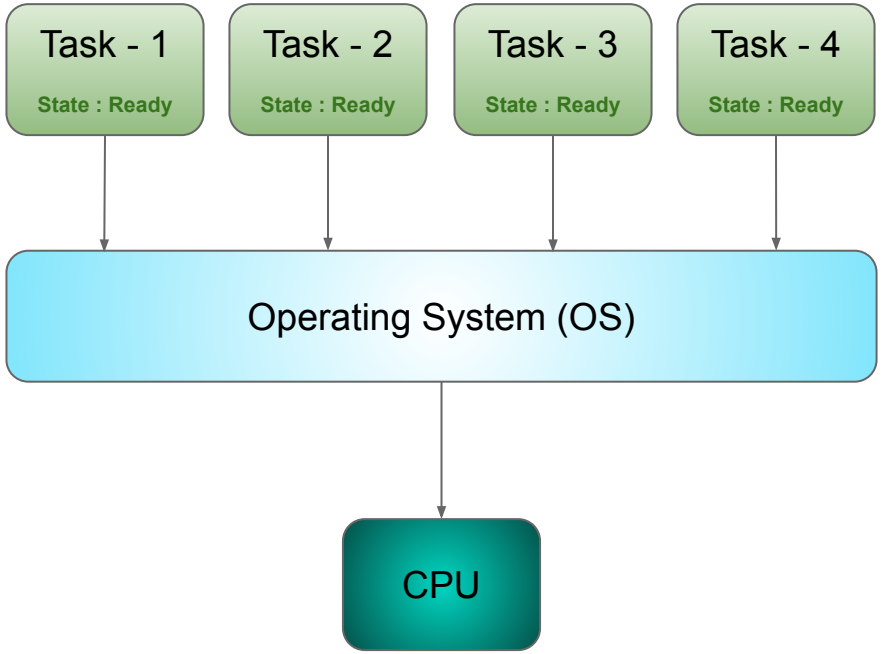
- Program
  - A set of instructions which is in human readable format. A passive entity stored on secondary storage.
- Executable
  - A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage.
- Process
  - A program loaded into memory and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting). A process is an active entity and needs resources such as CPU time, memory etc to execute.
- CPU
  - The central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.
- Register
  - A processor register is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage. A CPU only has a small number of registers.
- Memory
  - Memory refers to the computer hardware integrated circuits that store information for immediate use in a computer; it is synonymous with the term "primary storage". The memory is much slower than the CPU register but much larger in size.

A process in memory



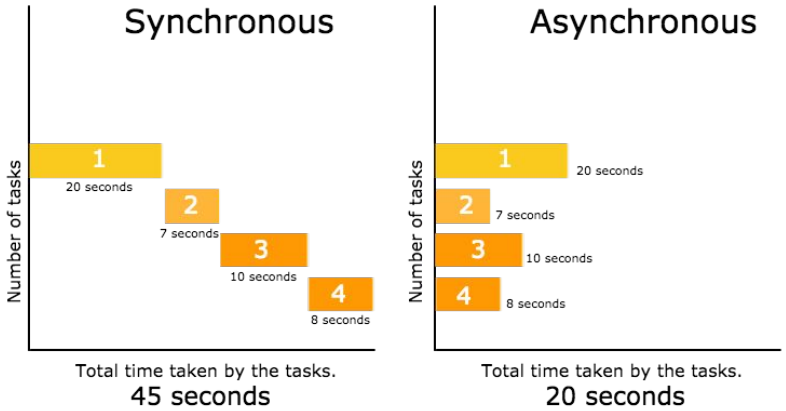
- Stack:
  - Temporary data such as function parameters, local variables and return addresses.
  - The stack grows from high addresses towards lower address.
- Heap:
  - Dynamically allocated (malloc) by the program during runtime.
  - The heap grows from low addresses towards higher addresses.
- Data:
  - Statically (known at compile time) global variables and data structures.
- Text:
  - The program executable machine instructions.

Multitasking (many process in memory)



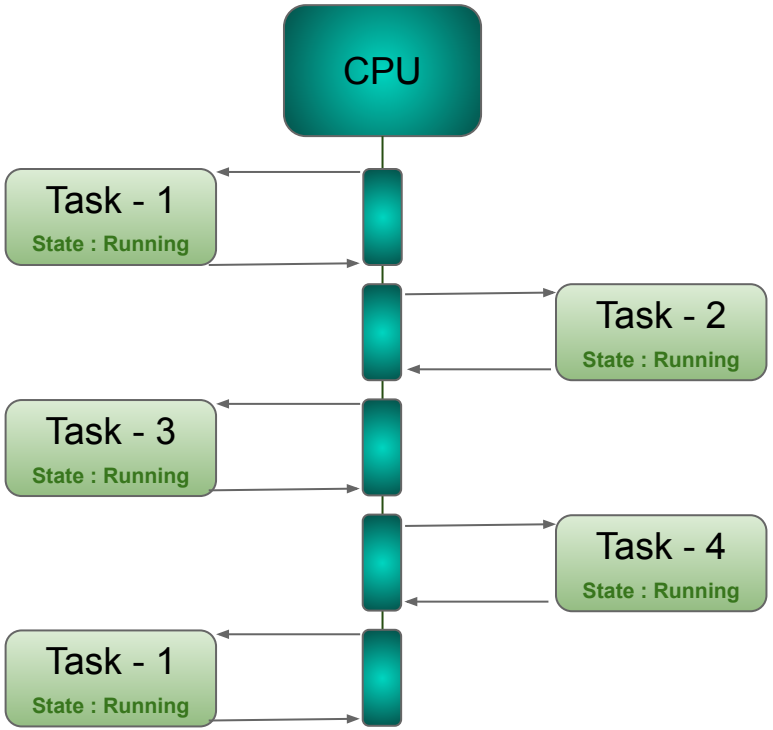
- Multitasking (aka time sharing) is a logical extension of multiprogramming where a timer is set to cause an interrupt at a regular time interval.
- At beginning, several jobs are kept in memory at the same time. Initially, all jobs are in the ready state.
- One of the ready jobs is selected to execute on the CPU and changes state from ready to running.
- The running job is replaced if the job requests I/O or if the job is interrupted by the timer. This way, the running job is given a time slice of execution than cannot be exceeded.

Synchronous and asynchronous events



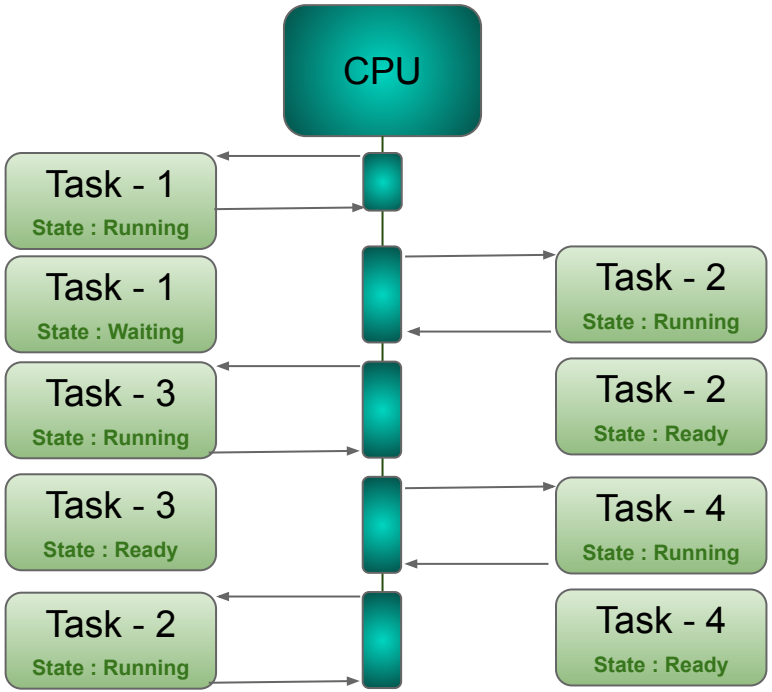
- Synchronous simply means that all events are occurring in a certain time order that can be predicted.
  - A certain event would always follow another and they can't be interchanged.
- Asynchronous is the opposite of synchronous.
  - In asynchronous processes, there is no time order.

Context Switching



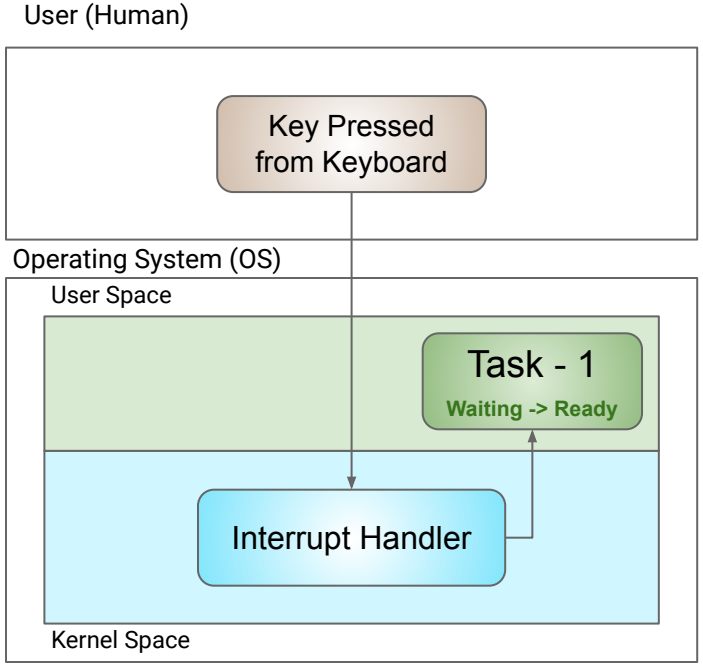
- Context switching switches the current process with the new process requiring the CPU to finish its tasks.
- At any point in time, the values of all the registers in the CPU defines the CPU context.
- When a new process takeover the processor time, the values in all registers must be saved to memory before the new process can use the registers.
- Similarly before resuming execution of the new process, the values of all registers must be restored using the values saved to memory earlier.

What about a Task waiting for keyboard Input



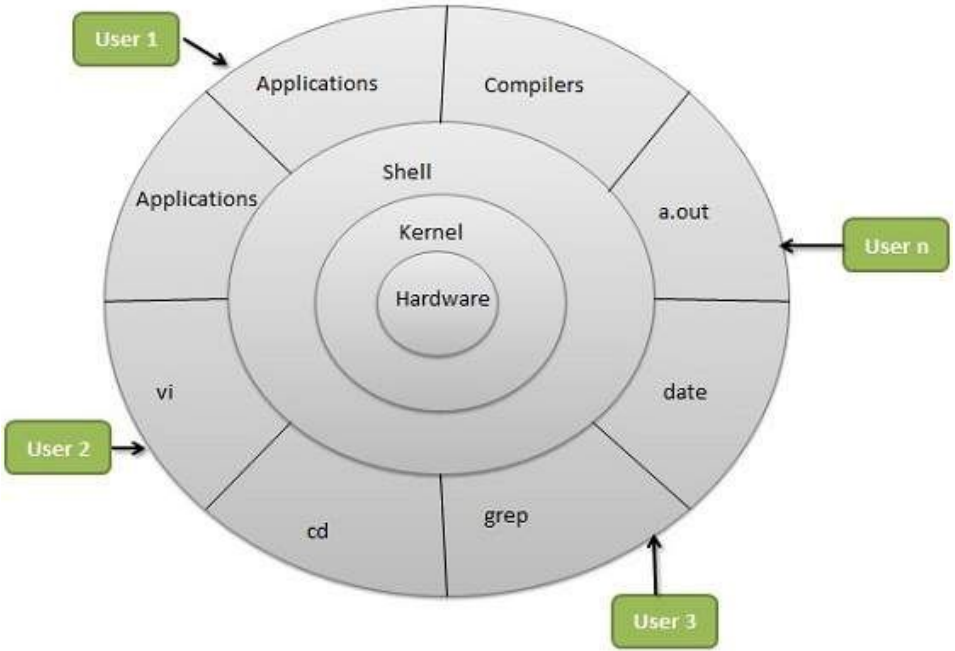
- Humans are very slow compared to the CPU.
  - No matter how fast you are, the CPU will be able to execute a huge number of instructions between every key-press you make.
- Is it possible to make the CPU do something useful while waiting for user input?
  - CPU switch context to next task and
  - The current task change it state to waiting instated of Ready.
  - When the next time slot come for the current task, it got ignored as the state is waiting but the slots are divided among ready task only.
- To bring back the Waiting task to ready state after the input got collected interrupt is used.

Interrupt driven input



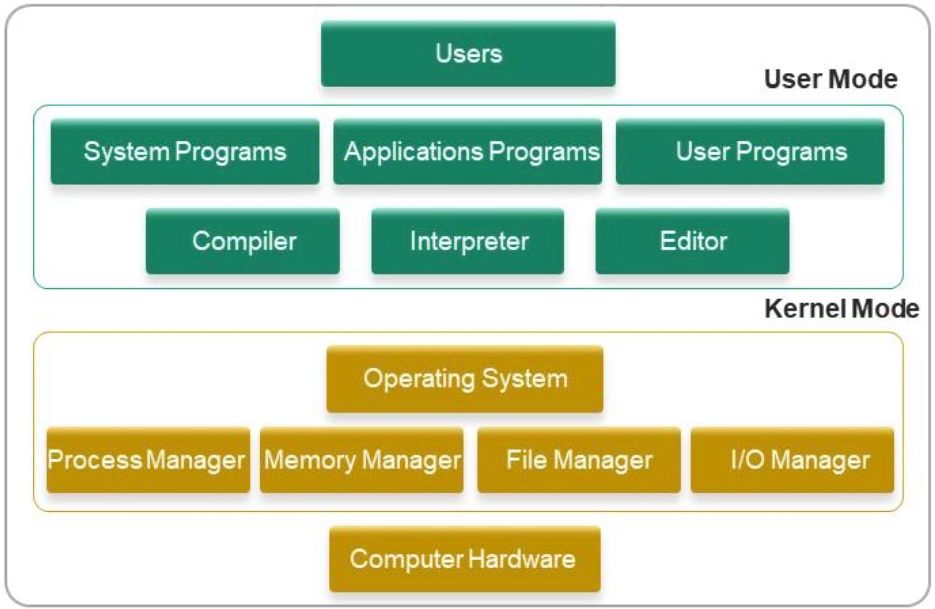
- Interrupts are used to notify the system when an I/O request is completed.
- Interrupts are generated by hardware devices outside the CPU at arbitrary times with respect to the CPU clock signals and are therefore considered to be asynchronous.
- Key-presses on a keyboard might happen at any time.
  - Even if a program is run multiple times with the same input data, the timing of the key presses will most likely vary.
- Read and write requests to disk is similar to key presses.
  - The disk controller is external to the executing process and the timing of a disk operation might vary even if the same program is executed several times.

Kernel



- The kernel is a computer program at the core of a computer's operating system and generally has complete control over everything in the system.
- It is the portion of the operating system code that is always resident in memory, and facilitates interactions between hardware and software components.
- The critical code of the kernel is usually loaded into a separate area of memory (Kernel Space), which is protected from access by application software or other less critical parts of the operating system.

User & The Kernel Mode

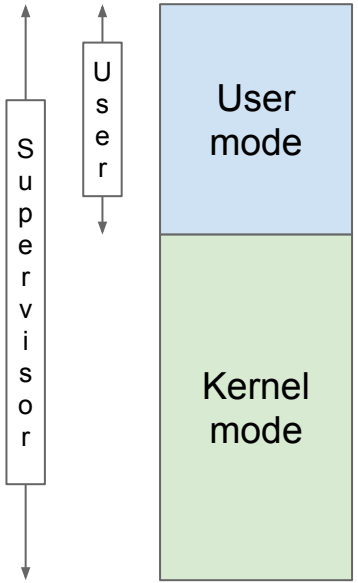


- Many decades ago, when computers were the size of a room, users were running their programs with huge struggles, and sometimes their programs even crashed the computer.
    - To protect against programs that continuously crashed computers, we developed newer operating systems with two distinct operative modes.
- Kernel mode, also known as system mode, is one of the central processing unit (CPU) operating modes. While processes run in kernel mode, they have unrestricted access to the hardware.
- The other mode is user mode, which is a non-privileged mode for user programs. Therefore, when a process runs in user mode, it has limited access to the CPU and the memory.

What is User Mode?

- It is a non-privileged mode in which each process starts out.
- Processes in this mode are forbidden to access those portions of memory that have been allocated to the kernel or to other programs.
  - The kernel is not a process, but rather a controller of processes, and it alone has access to all resources on the system.
- The system is in user mode when the operating system is running a user application such as handling a text editor.

What is Kernel Mode?

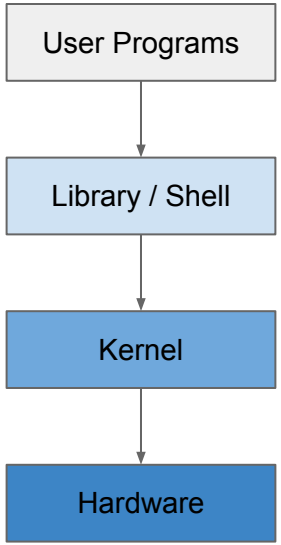


Process has their own memory space allocated & restricted within that space

As a supervisor Kernel have access to full memory space including the user Memory space

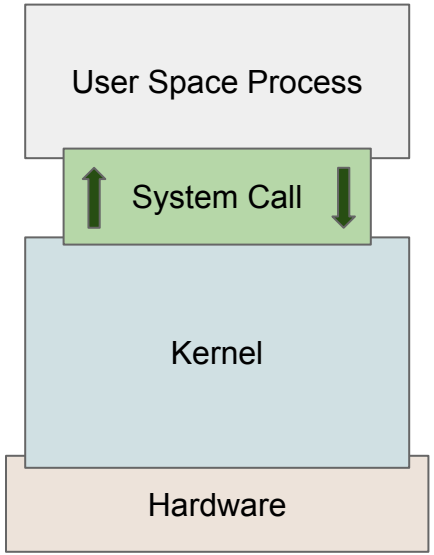
- Kernel mode, also known as system mode, is one of the central processing unit (CPU) operating modes. While processes run in kernel mode, they have unrestricted access to the hardware.
- It can execute any CPU instruction and reference any memory address.
- Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system.
- Crashes in kernel mode will halt the entire PC.

Necessity for User Mode and Kernel Mode



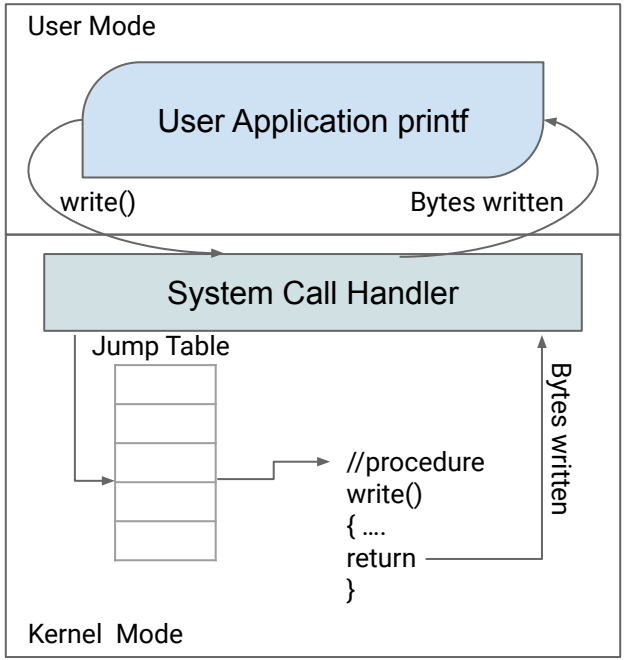
- The OS must accomplish two major objectives:
  - Interact with the hardware that services all of the low-level programmable elements
  - Maintain an environment in which the computer system's applications can run. These are also called user programs.
- Some operating systems, such as MS-DOS, allow user programs to directly interact with the hardware. On the other hand, Unix-like operating systems protect all low-level information about the physical organization of the machine from programs run by the user.
- A program must request the OS in order to access a hardware resource. The kernel reviews the request, and if it decides to give permission, interacts with the relevant hardware components on behalf of the user program.
- The hardware implements it in two different CPU execution modes: a non-privileged mode for user programs, and a privileged mode for kernel programs. Unix calls these user mode and kernel mode.

Dual mode operation



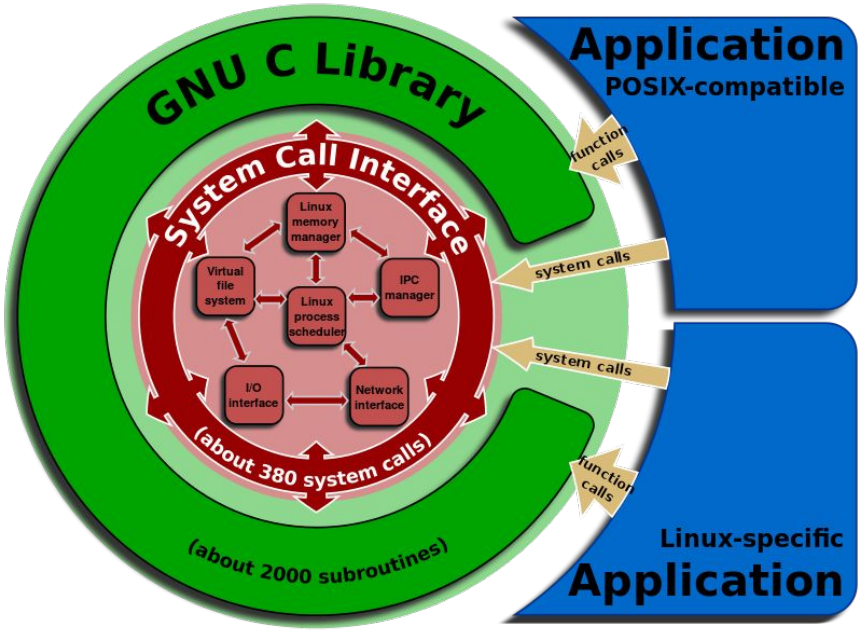
- There are times when a user program needs to read from disk, or get a line from the keyboard in user mode.
- Kernel is the only software that can do such operations.
- At that point, the program must ask the Kernel to access the hardware on its behalf and get some input for the program.
- So we need a mechanism where a user-mode program can switch into kernel mode.
- When the kernel has satisfied the process's request, it restores the process to user mode.
- This mechanism is implemented by all major OS through System Calls

What is system calls



- A system call is a mechanism that provides the interface between a process and the operating system.
  - It is a programmatic method in which a computer program requests a service from the kernel of the OS on which it is executed.
  - System calls are the only entry points for the kernel system.
- This may include hardware-related services (for example, accessing a hard disk drive or accessing the device's camera), creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- System calls provide an essential interface between a process and the operating system.

System Call Interface



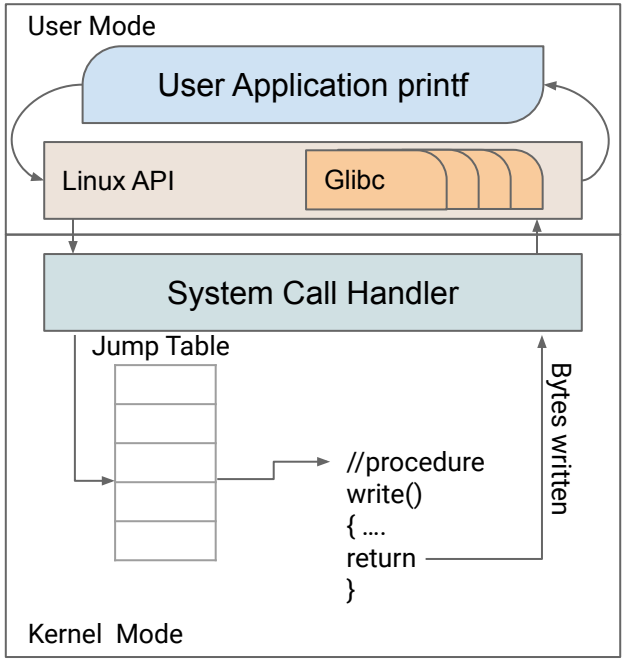
- The Linux kernel provides several interfaces to user-space applications that are used for different purposes and that have different properties by design.
- A system call interface is a set of functions for requesting a service from the kernel on the operating system they are executed on.
- System Call Interface is the denomination for the entirety of all implemented and available system calls in a kernel.

System Call Interface Example

```
#include <stdio.h>
int main(void) {
  printf("hello world!\n");
  return 0;
}
```

```
#include <unistd.h>
#include <sys/syscall.h>
int main(void) {
  syscall(1, 1, "hello world!\n",
14);
  return 0;
}
```

Linux API



- The Linux API is the kernel–user space API, which allows programs in user space to access system resources and services of the Linux kernel.
- It is composed out of the System Call Interface of the Linux kernel and the subroutines in the GNU C Library (glibc).
- The focus of the development of the Linux API has been to provide the usable features of the specifications defined in POSIX in a way which is reasonably compatible with other Unix like OS.
- The combination of the Linux kernel System Call Interface and a C standard library that wrap around the system calls is what builds the Linux API.

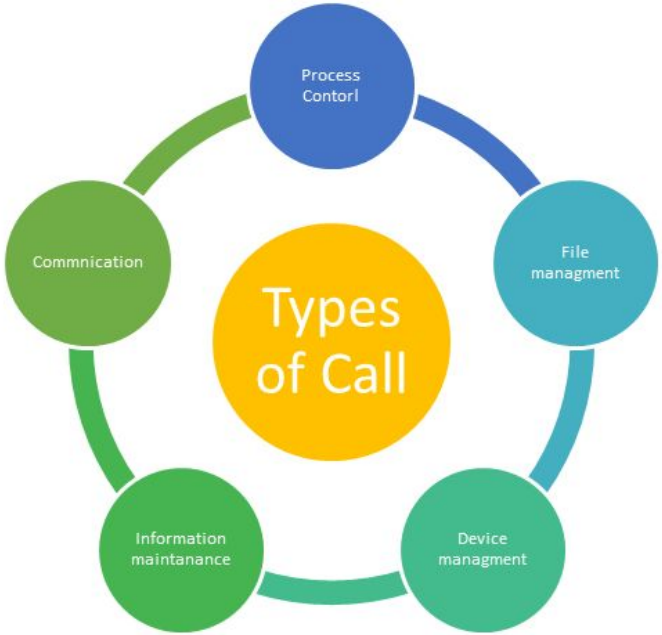
Linux API example

```
#include <unistd.h>
int main(void) {
  write(1, "hello, world!\n", 14);
  return 0;
}
```

- Generally, systems provide a library or API that sits between normal programs and the operating system.
- On Unix-like systems, that API is usually part of an implementation of the C library (libc), such as glibc, that provides wrapper functions for the system calls, often named the same as the system calls they invoke.
- Some popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.
- The primary function of the wrapper is to place all the arguments to be passed to the system call and also setting a unique system call number for the kernel to call.
- In this way the library, which exists between the OS and the application, increases portability.

Types of System calls

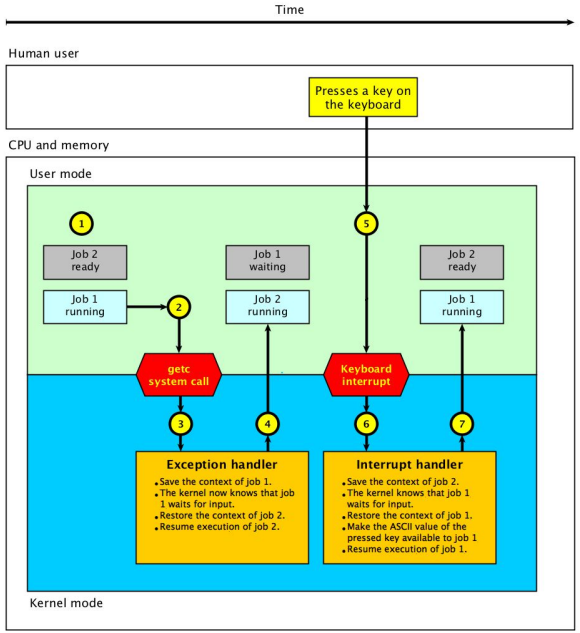
- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications



Role of System Call in each types

- Process Control
  - End and Abort
  - Load and Execute
  - Create Process and Terminate Process
  - Wait and Signal Event
  - Allocate and free memory
- File Management
  - Create a file
  - Delete file
  - Open and close file
  - Read, write, and reposition
  - Get and set file attributes
- Device Management
  - Request and release device
  - Logically attach/ detach devices
  - Get and Set device attributes
- Information Maintenance
  - Get or set time and date
  - Get process and device attributes
- Communication:
  - Create, delete communications connections
  - Send, receive message
  - Help OS to transfer status information
  - Attach or detach remote devices

System Call implementation (Exceptions/trap)



- Implementing system calls requires a transfer of control from user space to kernel space, which involves some sort of architecture-specific feature.
- System calls are implemented using a special system call exception called trap (not Error).
- CPU Handle trap exception similar to interrupt mechanism.
    - The only difference being exception is internal so synchronous where as interrupts are external and asynchronous.
- An Trap exception consist of 3 routine
    - Trap Vector : Which System call to revoke (Index of trap table)
    - Get the starting address of routine from the trap table & place on PC.
    - Save the address of the nest instruction (current PC) to stack.

Using strace

- The strace command traces the execution of another program, listing any system calls the program makes and any signals it receives.
- To watch the system calls and signals in a program, simply invoke strace , followed by the program and its command-line arguments.
  - For example, to watch the system calls that are invoked by the hostname command, use this command:

```
$ strace hostname
```

- This produces a couple screens of output. Each line corresponds to a single system call.

An example Program with system call

```
#include<fcntl.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
int main (int argc, char* argv[]) {
  char* file = argv[1];int fd;
  struct flock lock;
  printf ("opening %s\n", file);
  /* Open a file descriptor to the file.*/
  fd = open (file, O WRONLY);
  printf ("locking\n");
  /* Initialize the flock structure. */
  memset (&lock, 0, sizeof(lock));
  lock.l type = F WRLCK;
  /* Place a write lock on the file. */
  fcntl (fd, F SETLKW, &lock);
  printf ("locked; hit Enter to unlock... ");
  /* Wait for the user to hit Enter. */
  getchar ();
  printf ("unlocking\n");
  /* Release the lock. */
  lock.l type = F UNLCK;
  fcntl (fd, F_SETLKW, &lock);
  close (fd);
  return 0;
}
```

- The fcntl system call is the access point for several advanced operations on file descriptors (not file pointer).
- We'll describe here one of the most useful fcntl operations, file locking.
- The fcntl system call allows a program to place a read lock or a write lock on a file.
  - A read lock is placed on a readable file descriptor.
  - Write lock is placed on a writable file descriptor.
- More than one process may hold a read lock on the same file at the same time, but only one process may hold a write lock.
- placing a lock does not actually prevent other processes from accessing the file, unless they acquire locks with fcntl as well.

Summary

- What is Kernel Mode
- What is User mode
- Difference between Kernel Mode & User Mode
- What is System Calls?
- Why System calls?
- Type of System Calls
- How to execute System Call
- Debug system call using strace command.

Any Questions ?

Thank you!