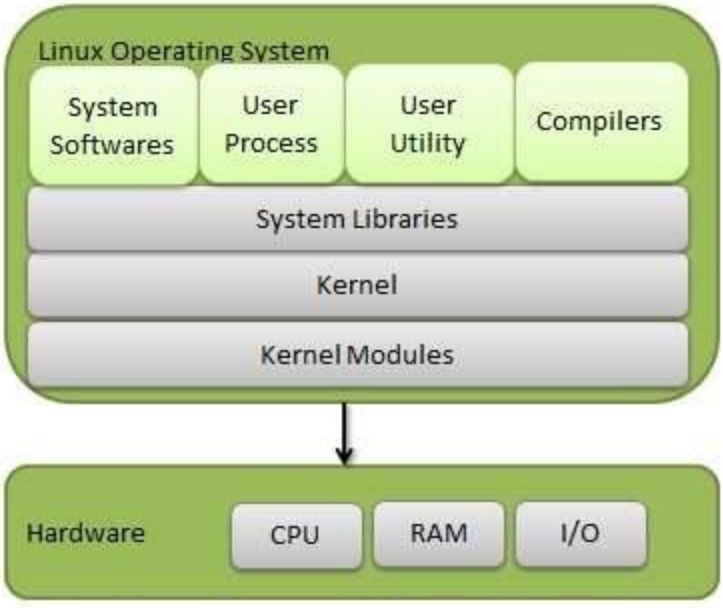
Topic 1: Components of Linux System:

Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

Components of Linux System

Linux Operating System has primarily three components

- **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.



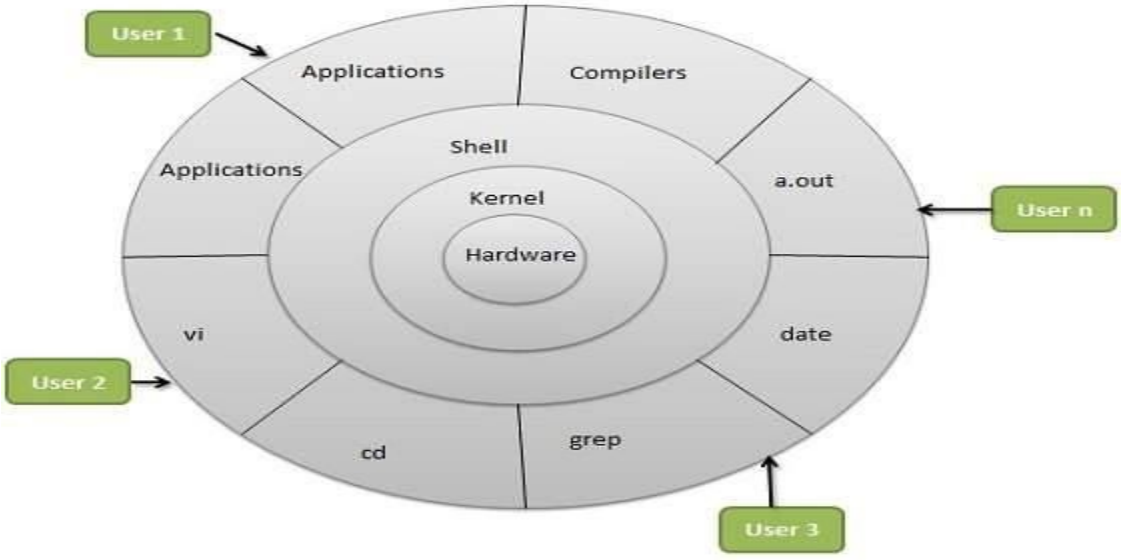
Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** – Portability means software can works on different types of hardware in same way.
- **Open Source** – Linux source code is freely available and it is community based development project.
- **Multi-User** – Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** – Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** – Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** – Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** – Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Architecture

The following illustration shows the architecture of a Linux system –



The architecture of a Linux System consists of the following layers –

- **Hardware layer** – Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).

- **Kernel** – It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** – An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.
- **Utilities** – Utility programs that provide the user most of the functionalities of an operating systems.

Topic 2: Interprocess Communication:

Inter-Process-Communication (or IPC for short) are mechanisms provided by the kernel to allow processes to communicate with each other. On modern systems, IPCs form the web that bind together each process within a large scale software architecture.

The Linux kernel provides the following IPC mechanisms:

1. Signals
2. Named Pipes or FIFOs
3. SysV Message Queues
4. POSIX Message Queues
5. SysV Shared memory
6. POSIX Shared memory
7. SysV semaphores
8. POSIX semaphores
9. Etc……

While the above list seems quite a lot, each IPC mechanism from the list describe above , is tailored to work better for a particular use-case scenario.

SIGNALS

Signals are the cheapest forms of IPC provided by Linux. Their primary use is to notify processes of change in states or events that occur within the kernel or other processes. We use signals in real world to convey messages with least overhead - think of hand and body gestures. For example, in a crowded gathering, we raise a hand to gain attention, wave hand at a friend to greet and so on.

On Linux, the kernel notifies a process when an event or state change occurs by interrupting the process's normal flow of execution and invoking one of the signal handler functinos registered by the process or by the invoking one of the default signal dispositions supplied by the kernel, for the said event.

NAMED PIPES OR FIFO

Named pipes (or FIFO) are variants of pipe that allow communication between processes that are not related to each other. The processes communicate using named pipes by opening a special file known as a FIFO file. One process opens the FIFO file from writing while the other process opens the same file for reading. Thus any data written by the former process gets streamed through a pipe to the latter process. The FIFO file on disk acts as the contract between the two processes that wish to communicate.

MESSAGE QUEUES

Message Queues are synonymous to mailboxes. One process writes a message packet on the message queue and exits. Another process can access the message packet from the same message queue at a latter point in time. The advantage of message queues over pipes/FIFOs are that the sender (or writer) processes do not have to wait for the receiver (or reader) processes to connect. Think of communication using pipes as similar to two people communicating over phone, while message queues are similar to two people communicating using mail or other messaging services.

There are two standard specifications for message queues.

1. **SysV message queues**.

   The AT&T SysV message queues support message channeling. Each message packet sent by senders carry a message number. The receivers can either choose to receive message that match a particular message number, or receive all other messages excluding a particular message number or all messages.
2. **POSIX message queues**.

   The POSIX message queues support message priorities. Each message packet sent by the senders carry a priority number along with the message payload. The messages get ordered based on the priority number in the message queue. When the receiver tries to read a message at a later point in time, the messages with higher priority numbers get delivered first. POSIX message queues also support asynchronous message delivery using threads or signal based notification.

Linux support both of the above standards for message queues.

SHARED MEMORY

As the name implies, this IPC mechanism allows one process to share a region of memory in its address space with another. This allows two or more processes to communicate data more efficiently amongst themselves with minimal kernel intervention.

There are two standard specifications for Shared memory.

1. **SysV Shared memory**. Many applications even today use this mechanism for historical reasons. It follows some of the artifacts of SysV IPC semantics.
2. **POSIX Shared memory**. The POSIX specifications provide a more elegant approach towards implementing shared memory interface. On Linux, POSIX Shared memory is actually implemented by using files backed by RAM-based filesystem. I recommend using this mechanism over the SysV semantics due to a more elegant file based semantics.

SEMAPHORES

Semaphores are locking and synchronization mechanism used most widely when processes share resources. Linux supports both SysV semaphores and POSIX semaphores. POSIX semaphores provide a more simpler and elegant implementation and thus is most widely used when compared to SysV semaphores on Linux.

Topic 3: Synchronization

Linux kernel provide couple of synchronization mechanism.

1. **Atomic operation:** This is the very simple approach to avoid race condition or deadlock. Atomic operators are operations, like add and subtract, which perform in one clock cycle (uninterruptible operation). The atomic integer methods operate on a special data type, *atomic_t.* **A common use of the atomic integer operations is to implement counters which is updated by multiple threads.** The kernel provides two sets of interfaces for atomic operations, one that operates on integers and another that operates on individual bits. All atomic functions are inline functions.

2. **Semaphore:** This is another kind of synchronization mechanism which will be provided by the Linux kernel. When some process is trying to access semaphore which is not available, semaphore puts process on wait queue(FIFO) and puts task on sleep. That's why semaphore is known as a sleeping lock. As soon as semaphore get available, one of task from wait queue in invoked.

There two flavors of semaphore is present.

- Basic semaphore
- Reader-Writter Semaphore

When a multiple threads/process wants to share data, in the case where read operation on data is more frequent and write operation is rare. In this scenario Reader-Writter Semaphore is used. Multiple thread can read a data by same time. The data will be only locked(all other read thread should wait) when one thread write/update data. On the other side writers has to wait until all the readers release the read lock. When writer process release lock the reader from wait-queue(FIFO) will get invoked.

Couple of observations about nature of semaphore :

1. Semaphore puts a task on sleep. So the semaphore can be only used in process context. Interrupt context can not sleep.
2. Operation to put task on sleep is time consuming(overhead) for CPU. So semaphore is suitable for lock which is holding for long term. Sleeping and invoking task over kills CPU if semaphore is locked and unlocked for short time via multiple tasks.
3. Semaphore wait list is FIFO in nature. So the task which tried to acquire semaphore first will be waken up from wait list first.
4. Semaphore can be acquired/release from any process/thread.

3. **Spin-lock:** This is special type of synchronization mechanism which is preferable to use in multi-processor(SMP) system. Basically its a busy-wait locking mechanism until the lock is available. In case of unavailability of lock, it keeps thread in light loop and keep checking the availability of lock. Spin-lock is not recommended to use in single processor system. If some procesq_1 has acquired a lock and other process_2 is trying to acquire lock, in this case process 2 will spins around and keep processor core busy until it acquires lock. process_2 will create a deadlock, it dosent allow any other process to execute because CPU core is busy in light loop by semaphore.

There two flavors of spin-lock is present.

- Basic spin-lock
- Reader-Writter Spin-lock

With increasing the level of concurrency in Linux kernel read-write variant of spin-lock is introduces. This lock is used in the scenario where many readers and few writers are present. Read-write spin-lock can have multiple readers at a time but only one writer and there can be no readers while there is a writer. Any reader will not get lock until writer finishes it.

4. **Sequence Lock:** This is very useful synchronization mechanism to provide a lightweight and scalable lock for the scenario where many readers and a few writers are present. Sequence lock maintains a counter for sequence. When

the shared data is written, a lock is obtained and a sequence counter is incremented by 1. Write operation makes the sequence counter value to odd and releasing it makes even. In case of reading, sequence counter is read before and after reading the data. If the values are the same which indicates that a write did not begin in the middle of the read. In addition to that, if the values are even, a write operation is not going on. Sequence lock gives the high priority to writers compared to readers.

Topic 4: Interrupt, Exception and System Call

See another attachment of this mail with name

“Interrupts, Exceptions and System Calls”