# 1) Variables storage in process segments

global variable-> .DATA

static variale-> .BSS if initialized as 0, .DATA else

local variable -> Stack

constant type -> code or .DATA

variables in main() -> Stack

pointers -> .DATA or stack

dynamically allocated variable -> Heap

# 2) Usage of volatile

During executing a program, the compiler seems tentative to apply aggressive optimization upon some paragraphs of code. If you change variable or overload/ override a function from somewhere else, the compiler wouldn’t recognize the difference, and perform the optimization anyway. So to prevent such actions from happening, you can add keyword ***volatile*** ahead of a variable to tell the compiler not to execute normal optimization.

# 3) Difference between mutex and semaphore

**Atomic operation:**

Atomic operation can be interpreted as an unbreakable or an uninterrupted operation.

**Critical section:**

When executing concurrent programs, multi-threaded operations may acquire shared data simultaneously. So it’s very critical for us to provide synchronization of shared data so that the result will be as predicted.

**Mutex:**

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, **only one thread can work with the entire buffer**. The concept can be generalized using semaphore.

**Semaphore:**

A **semaphore is a generalized mutex**. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The **consumer and producer can work on different buffers at the same time**.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

# 4) Difference between Thread and process

**Processes**

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

**Threads**

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

1. Processes are independent while thread is within a process.

2. Processes have separate address spaces while threads share their address spaces.

3. Processes communicate each other through inter-process communication.

4. Processes carry considerable state (e.g., ready, running, waiting, or stopped) information, whereas multiple threads within a process share state as well as memory and other resources.

5. [Context switching](http://www.bogotobogo.com/cplusplus/multithreaded.php" \l "ContextSwitch" \t "_blank) between threads in the same process is typically faster than context switching between processes.

**6. Multithreading** has some advantages over **multiple processes**. Threads require less overhead to manage than processes, and intraprocess thread communication is less expensive than interprocess communication.

**7. Multiple process** concurrent programs do have one advantage: Each process can execute on a different machine (**distribute program**). Examples of distributed programs are file servers (NFS), file transfer clients and servers (FTP), remote log-in clients and servers (Telnet), groupware programs, and Web browsers and servers.

# 6) Thread inter communication mechanisms and synchronization

There are three major types of inter-thread communication.

First –flag: Raise a flag during execution of a thread by ISR (Interrupt service routines), while main program will continuously check if flag is on. If the flag is on, the program will pause, and execute the program in that thread, otherwise, the program will run without interruption.

Second-mailbox. A flag accompanied by a data form. The data flow could be both directions, either main thread check flag, and execute data in background thread, or vice versa.

Third—FIFO. Once run into a ISR, the FIFO will be created by a function called put(data). At the same time, we will create another get() function in main(). Instead of sending just on data, we can now send a data stream following the order of FIFO.

# 7) WHY use multi-thread programming?

Application that has more than one thread of execution within the application itself is called multhreaded application.

For example, if we want to create a server that can serve as many concurrent connections as the the server can cope with, we can achieve such a task relatively easily if we devote a new thread to each connection. In some cases, instead of creating a separate socket to handle incoming connections, a multithreaded server creates a new thread for each incoming connection, and creates a new socket with each new thread.

Another typical example of the case when we need multithreading is GUI application. GUI applications have one thread of execution (Main Thread) and do one operation at a time. This thread is either waiting for an event or processing an event. So, if the user triggers a time-consuming operation from the user interface, the interface freezes while the operation is in progress. In this case, we need to implement multithreading.

The main thread can start new threads by creating objects of a Thread subclass. If multithreaded, the GUI runs in its own thread and additional processing takes place in other threads, and the application will have responsive GUIs even during intensive processing. In other words, we pass on the heavy-duty processing to separate secondary threads and leave the primary GUI thread free to respond to the user.

These new threads need to communicate among themselves so that application can keep the user informed of progress (keep the user interface up-to-date regarding the progress), allow the user to intervene (provide the user with some control over the secondary thread), and let the primary thread know when processing is complete. Typically, they use **shared variables** together by using a resource protection mechanism such as **mutextes**, **semaphores**, or **wait conditions**, etc.

# 8) Processes vs Kernel

In Linux systems, a **kernel** knows and controls everything of a running system but a **process** does not. A process is an instance of an executing program. When a program is executed, the kernel loads the code of the program into virtual memory, allocates space for program variables, and sets up kernel bookkeeping data structures to record various information (such as process ID, termination status, user IDs, and group IDs) about the process.

**Process**

A running system typically has numerous processes.

For a process, many things happen asynchronously.

An executing process doesn't know when it will next time out, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. The delivery of signals and the occurrence of interprocess communication events are mediated by the kernel, and can occur at any time for a process.

Many things happen transparently for a process. A process doesn't know where it is located in RAM or, in general, whether a particular part of its memory space is currently resident in memory or held in the swap area (a reserved area of disk space used to supplement the computer's RAM).

Similarly, a process doesn't know where on the disk drive the files it accesses are being held; it simply refers to the files by name.

A process operates in isolation; it can't directly communicate with another process.

A process can't itself create a new process or even end its own existence. Actually, a process can create a new process using the [fork()](http://www.bogotobogo.com/Linux/linux_process_and_signals.php) system call. The process that calls **fork()** is referred to as the parent process, and the new process is referred to as the child process. The **kernel**creates the child process by making a duplicate of the parent process.

Finally, a process can't communicate directly with the input and output devices attached to the computer.

**Kernel**

The kernel facilitates the running of all processes on the system.

The kernel decides which process will next obtain access to the CPU, when it will do so, and for how long.

The kernel maintains data structures containing information about all running processes and updates these structures as processes are created, change state, and terminate.

The kernel maintains all of the low-level data structures that enable the filenames used by programs to be translated into physical locations on the disk.

The kernel also maintains data structures that map the virtual memory of each process into the physical memory of the computer and the swap area(s) on disk.

All communication between processes is done via mechanisms provided by the kernel. In response to requests from processes, the kernel creates new processes and terminates existing processes.

Lastly, the kernel (in particular, device drivers) performs all direct communication with input and output devices, transferring information to and from user processes as required.

# 9) Interprocess communication and synchronization.

**1. Pipe**

**2. Named pipes (FIFO)**

**3. Semaphores**

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it. Semaphores are often dubbed the most difficult to grasp of the three types of System V IPC objects. In order to fully understand semaphores, we'll discuss them briefly before engaging any system calls and operational theory.

4. Shared memory

# 10) Multi-thread deadlock, livelock and starvation

**Deadlock:**

A **deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.

Resource A and resource B are used by process X and process Y

1. X starts to use A.
2. X and Y try to start using B
3. Y 'wins' and gets B first
4. now Y needs to use A
5. A is locked by X, which is waiting for Y

The best way to avoid deadlocks is to avoid having processes cross over in this way. Reduce the need to lock anything as much as you can.

**Starvation**

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

**Livelock**

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...