Table of Contents

Technical Questions

Diff between embedded systems and general computers system

Technically both are computers by definition, they have processors, RAM, ROM, and other various peripherals, but :

Embedded System, as it appears from its name, is a part of a bigger system, a computer restricted to **one function** (or a finite set of functions) that controls, monitors or integrate with larger systems like automotive, robotics, home appliances and military applications.

They have very tight constraints regarding size, performance, memory, price and durability, also an embedded system is required in most cases to respond in real time, you don't want your brakes to work after pressing the pedal by a couple of seconds tho..

General Computer System is a computer that is built to be customizable in software, like desktop PCs and laptops, you can make it do many thing, sometimes together, with low or no constraints on power, performance or cost, also a general computing system is contained in itself, it's not a part of a larger system.. it is the system itself.

What is Thread?

A thread of execution is the **smallest sequence of programmed instructions** that can be managed independently by a scheduler, which is typically a part of the operating system.[1] The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

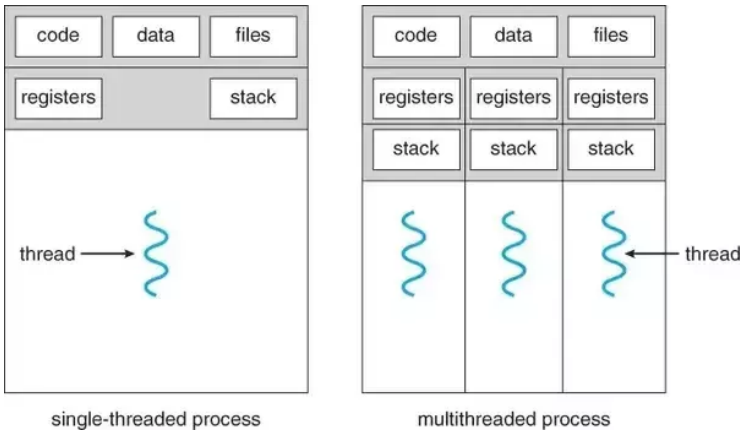
Systems with a single processor generally implement multithreading by time slicing: the central processing unit (CPU) switches between different software threads. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel. On a multiprocessor or multi-core system, multiple threads can execute in parallel, with every processor or core executing a separate thread simultaneously; on a processor or core with hardware threads, separate software threads can also be executed concurrently by separate hardware threads.

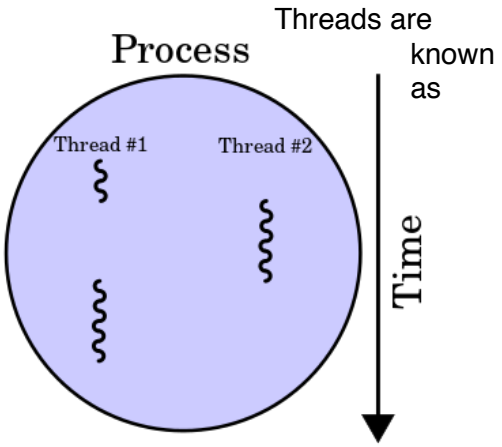
What is concurrency and multithreading?

Multithreading is a widespread programming and execution model that allows **multiple threads to exist within the context of one process**. These threads share the process's resources, but are able to execute independently. **A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.**

In computer architecture, multithreading is the ability of **a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads**

concurrently, appropriately supported by the operating system. This approach differs from multiprocessing, as with multithreading the processes and threads share the resources of a single or multiple cores: the computing units, the CPU caches, and the translation lookaside buffer (TLB). **(everything including** head **is shared among threads but only stack is not)**





Threads are known as Lightweight processes. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multiple processes can be executed parallel-y by increasing number of threads.

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.

Advantages:
If a thread gets a lot of cache misses, the other threads can **continue taking advantage of the unused computing resources**, which may lead to **faster overall execution as these resources would have been idle** if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

Where multiprocessing systems include multiple complete processing units, multithreading aims to **increase utilization of a single core by using thread-level as well as instruction-level parallelism**.

Disadvantages:
Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but can be degraded, even when only one thread is executing, due to lower

frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Benefits of multithreaded programming?

Responsiveness. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource sharing. Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default.

Economy. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general, it is much more time consuming to create and manage processes than threads.

Scalability. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multiCPU machine increases parallelism

Better system utilization as an example, a file system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) with neither thread waiting for the other to finish.

Faster execution this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple central processing units (CPUs) or one or more multi-core processors, or across a cluster of machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).

What is dead lock and what cause it?

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock
There are three ways to handle deadlock
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.
2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.
3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Benefits of process over threads?

1. **No data corruption.** When using processes you are forced to deal with communication via messages, for example, this is the way Erlang handles communication. Data is not shared, so there is no risk of data corruption.

2. **Independency.** Another advantage of processes is that they can crash and you can feel relatively safe in the knowledge that you can just restart them (even across network hosts). However, if a thread crashes, it may crash the entire process, which may bring down your entire application. To illustrate: If an Erlang process crashes, you will only lose that phone call, or that webrequest, etc. Not the whole application.

What is volatile keyword?

Volatile is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby.

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

1) Objects declared as **volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time**. The system always reads the current value of **a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested**, even if a previous instruction asked for a value from the same object.

2) **Global variables within a multi-threaded application**: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. **When two threads sharing information via global variable, they need to be qualified with volatile.** Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. **To nullify the effect of compiler optimizations, such global variables to be qualified as volatile**

If we do not use volatile qualifier, the following problems may arise
1) Code may not work as expected when optimization is turned on.
2) Code may not work as expected when **interrupts are enabled and used.**

What is static keyword in C?

- ***Act as global variable only visible to this function/file (static variables in function)***
- ***limit the scope of the variable/function (static class objects/function)***

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

1) **A static int variable remains in memory while the program is running**. A normal or auto variable is destroyed when a function call where the variable was declared is over.
2) Static variables are allocated memory in **data segment, not stack segment**.
3) Static variables (like global variables) are initialized as 0 if not initialized explicitly.
4) In C, **static variables can only be initialized using constant literals.**
5) Static global variables and functions are also possible in C/C++. The purpose of these is to **limit scope of a variable or function to a file**. Access to static functions is restricted to the file where they are declared. Another reason for making functions static can be reuse of the same function name in other files.
6) **Static variable limits the scope of the variable/function within the module it is been declared.**

Difference between declare and defining a variable?

When a variable is defined, the compiler allocates memory for that variable and possibly also initializes its contents to some value. **When a variable is declared, the compiler requires that the variable be defined elsewhere. The declaration informs the compiler that a variable by that name and type exists**, but the compiler does not need to allocate memory for it since it is allocated elsewhere.

What is extern keyword?

In the C programming language, an external variable is a variable defined outside any function block. On the other hand, a local (automatic) variable is a variable defined inside a function block. **An external variable can be accessed by all the functions in all the modules of a program. It is a global variable.**

The extern keyword means "declare without defining". In other words, it is a way to explicitly declare a variable, or to force a declaration without a definition. As an alternative to automatic variables, it is possible to **define variables that are external to all functions**, that is, variables that can be accessed by name by any function. It is also possible to explicitly define a variable, i.e. to force a definition. It is done by assigning an initialization value to a variable. If neither the extern keyword nor an initialization value are present, the statement can be either a declaration or a definition. It is up to the compiler to analyse the modules of the program and decide.

A variable must be defined exactly once in one of the modules of the program. If there is no definition or more than one, an error is produced, possibly in the linking stage. A variable may be declared many times, as long as the declarations are consistent with each other and with the definition (something which header files facilitate greatly). It may be declared in many modules,

including the module where it was defined, and even many times in the same module. But it is usually pointless to declare it more than once in a module.

An external variable may also be declared inside a function. In this case the extern keyword must be used, **otherwise the compiler will consider it a definition of a local (automatic) variable**, which has a different scope, lifetime and initial value. **This declaration** will only be visible inside the function instead of throughout the function's module.

The extern keyword applied to **a function prototype** does absolutely **nothing** (the extern keyword applied to a function definition is, of course, non-sensical). A function prototype is always a declaration and never a definition. Also, in standard C, **a function is always external,** but some compiler extensions allow a function to be defined inside a function.

Extern vs. Static keyword

The static storage class is used to declare an identifier that is a local variable either to a function or a file and that exists and retains its value after control passes from where it was declared. This storage class has a duration that is permanent. A variable declared of this class retains its value from one call of the function to the next. The scope is local. **A variable is known only by the function it is declared within or if declared globally in a file**, it is known or seen only by the functions within that file. This storage class guarantees that declaration of the variable also initializes the variable to zero or all bits off.

The extern storage class is used to declare a global variable that will be known to the functions in a file and capable of being known to all functions in a program. This storage class has a duration that is permanent. Any variable of this class retains its value until changed by another assignment. The scope is global. A variable can be known or seen by all functions within a program.

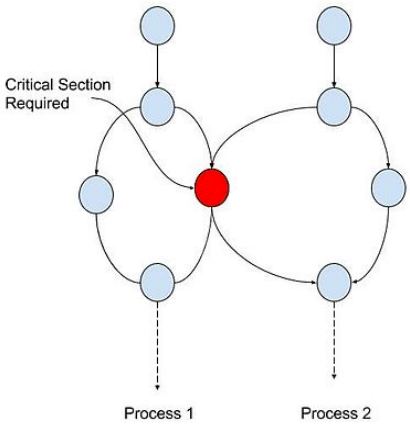
Extern and static are mutually exclusive.

Static means **Internal Linkage**, extern means **External Linkage**.
Internal Linkage refers to **everything only in scope** of a Translation unit.
External Linkage refers to things that **exist beyond** a particular translation unit. In other words, accessible through the whole program.

So both are mutually exclusive.

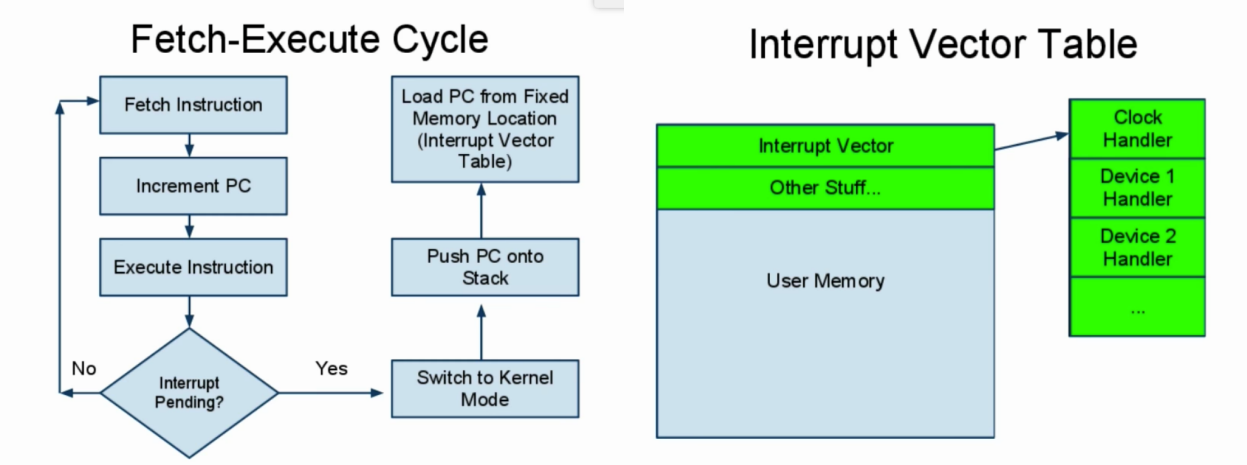
Critical Section

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so **parts of the program where the shared resource is accessed protected. This protected section is the critical section or critical region**. It cannot be executed by more than one process. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.



Interrupt Handling

- CPU checks for HW interrupts after each user mode instruction is executed.
- If an interrupt signal is present, a kernel routine is called to handle the interrupt.
- Different routine or interrupt handler are chosen from the Interrupt vector table to handle different interrupt. These routine are pre defined code that stored in a fixed memory position in the kernel memory space.
- Interrupt Descriptor Table is reserved RAM memory block that intel and AMD CPUs provide special instructions and structures for fast interrupt handling.
- Two types of interrupt handing: slow and fast (all happen in kernel).
- Fast handler is the piece of code invoked directly from IVT (vector table) whenever an interrupt occurs.
- Slow handlers are invoked at a later time and lower priority (can be preempt by fast handlers). It will be put in the OS task queue. CPU check this task queue after executing fast handlers.
- Fast handlers are atomic (un-interruptable), achieving by disabling interrupt while executing fast handlers.
- There will be trouble if FIH (fast interrupt handler) takes too long because it wont take in other interrupts in the process.
- An interrupt storm happens when another interrupt is always waiting to be processed whenever FIH finishes execution (FIH takes too long).
- If OS perpetually handling interrupts, it never run app code and no respond to users input -> result in **live lock -** system running but appear freezing. (use power button to solve this)
- Thats why we use two types of handlers. FIH must be fast and atomic!!! If slow, then treat it as slow IH which can be preempted.



Upper half and bottom half of ISR

The generic problem of any ISR is it's **latency**. Since most often the corresponding interrupt is disabled during the execution of a ISR, the ISR is expected to be short.But, what if you have to do a lot of data processing, memory allocation in a ISR. Linux overcomes this problem by providing an infrastructure in the kernel to split a ISR into

Top-Half: This is the critical section. **The interrupt is disabled** when this is executed.
Bottom-Half: This is the less critical section. **The interrupt is enable when this is executed.**

It is not mandatory to split the ISR's in Linux. It is a device driver developers choice. If the driver developer feels, the ISR is going to very short and can be managed then he may not decide to implement a bottom-half. Similarly, if the driver developer thinks disabling interrupt for a long time is OK for his use case, then he may still have a fat top-half. So, it is a design decision.

The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half -- that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

One thing to keep in mind with bottom-half processing is that all of the restrictions that apply to interrupt handlers also apply to bottom halves. Thus, bottom halves cannot sleep, cannot access user space, and cannot invoke the scheduler.

The Linux kernel has two different mechanisms that may be used to implement bottom-half processing. **Tasklets** were introduced late in the 2.3 development series; they are now the preferred way to do bottom-half processing, but they are not portable to earlier kernel versions. The older **bottom-half (BH) implementation** exists in even very old kernels, though it is

implemented with tasklets in 2.4. We'll look at both mechanisms here. In general, device drivers writing new code should choose tasklets for their bottom-half processing if possible, though portability considerations may determine that the BH mechanism needs to be used instead.

Direct Memory Access (DMA)

Direct memory access (DMA) is a feature of computer systems that a**llows certain hardware subsystems to access main system memory (Random-access memory)**, independent of the central processing unit (CPU).

Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done. This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without DMA channels. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

To carry out an input, output or memory-to-memory operation, the host processor initializes the DMA controller with a count of the number of words to transfer, and the memory address to use. The CPU then sends commands to a peripheral device to initiate transfer of data. The DMA controller then provides addresses and read/write control lines to the system memory. Each time a byte of data is ready to be transferred between the peripheral device and memory, the DMA controller increments its internal address register until the full block of data is transferred.

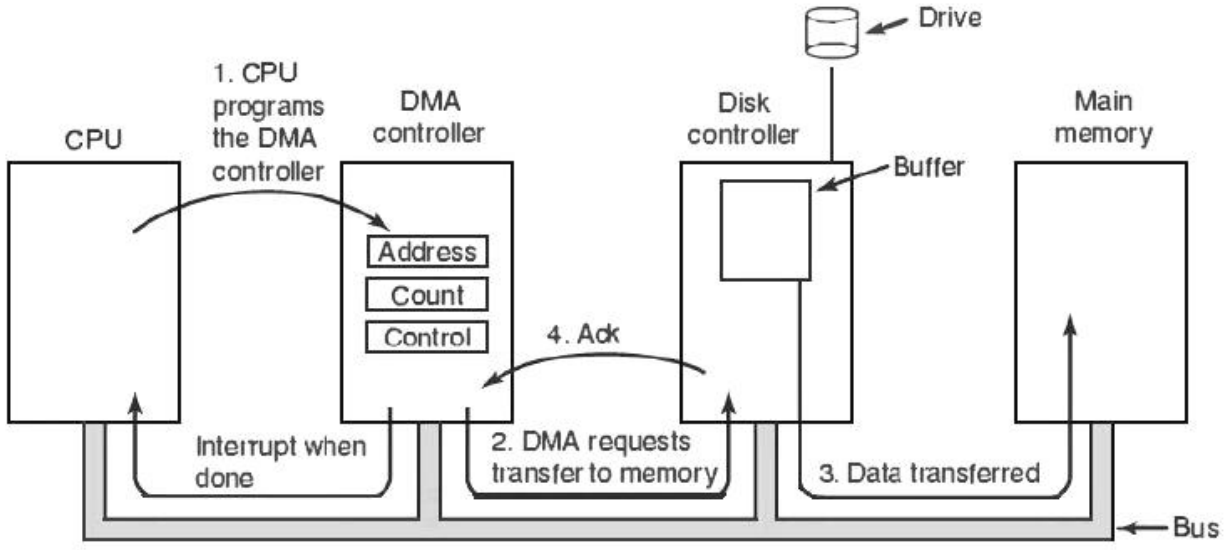


Figure 1. Operation of a DMA transfer.

Priority inversion in a RTOS and its solutions

In computer science, priority inversion is **a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks**. (only happens in priority based scheduling)

Consider two tasks H and L, of high and low priority respectively, either of which can acquire exclusive use of a shared resource R. If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource. Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time. Despite good design, however, it is possible that a third task M of medium priority ($p(L) < p(M) < p(H)$, where $p(x)$ represents the priority for task (x)) becomes runnable during L's use of R. At this point, M being higher in priority than L, preempts L, causing L to not be able to relinquish R promptly, in turn causing H—the highest priority process—to be unable to run. This is called priority inversion where a higher priority task is preempted by a lower priority one.

L is running in CS (critical section); H also needs to run in CS ; H waits for L to come out of CS ; M interrupts L and starts running (while L is in its CS); M runs till completion and relinquishes control ; L resumes and starts running till the end of CS ; H enters CS and starts running.
Note that neither L nor H share CS with M.

The most famous ‘Priority Inversion’ problem was what happened at Mars Pathfinder.

Disabling all interrupts to protect critical sections
When disabling interrupts is used to prevent priority inversion, there are only two priorities: preemptible, and interrupts disabled. With no third priority, inversion is impossible. Since there's only one piece of lock data (the interrupt-enable bit), misordering locking is impossible, and so deadlocks cannot occur. Since the **critical regions always run to completion**, hangs do not occur. Note that this only works if all interrupts are disabled. If only a particular hardware device's interrupt is disabled, priority inversion is reintroduced by the hardware's prioritization of interrupts. In early versions of UNIX, a series of primitives named splx(0) ... splx(7) disabled all interrupts up through the given priority. By properly choosing the highest priority of any interrupt that ever entered the critical section, the priority inversion problem could be solved without locking out all of the interrupts. Ceilings were assigned in rate-monotonic order, i.e. the slower devices had lower priorities.

In multiple CPU systems, a simple variation, "single shared-flag locking" is used. This scheme provides a single flag in shared memory that is used by all CPUs to lock all inter-processor critical sections with a busy-wait. Interprocessor communications are expensive and slow on most multiple CPU systems. Therefore, most such systems are designed to minimize shared resources. As a result, this scheme actually works well on many practical systems. These methods are widely used in simple embedded systems, where they are prized for their reliability, simplicity and low resource use. These schemes also require clever programming to keep the critical sections very brief. Many software engineers consider them impractical in general-purpose computers.

A priority ceiling

With priority ceilings, the shared mutex process (that runs the operating system code) has a characteristic (high) priority of its own, which is assigned to the task locking the mutex. This works well, provided the other high priority task(s) that tries to access the mutex does not have a priority higher than the ceiling priority.

Priority inheritance
Under the policy of priority inheritance, whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low priority task continues at its original priority level.

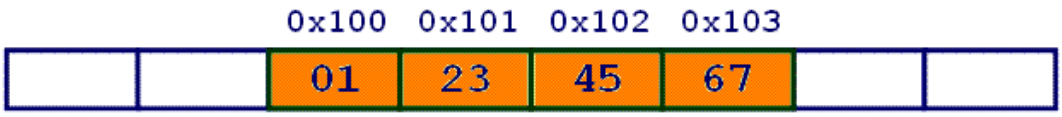
Random boosting
Ready tasks holding locks are randomly boosted in priority until they exit the critical section. This solution is used in Microsoft Windows.

Avoid blocking
Because priority inversion involves a low-priority task blocking a high-priority task, one way to avoid priority inversion is to avoid blocking, for example by using Non-blocking synchronization or Read-copy-update.

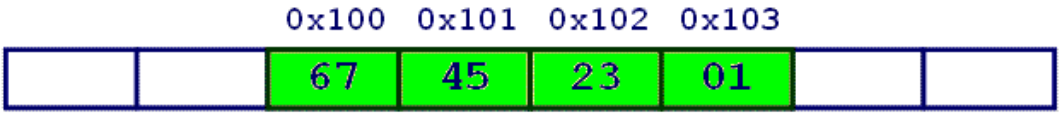
Big & Little endian – definitions, representations, write it down, swap them, etc

Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C ++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Big Endian



Little Endian

Check machine Endianness:

```
#include <stdio.h>
int main()
{
  unsigned int i = 1;
  char *c = (char*)&i;
  if (*c)
     printf("Little endian");
  else
     printf("Big endian");
  getchar();
  return 0;
}
```

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

Semaphore vs mutex

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as synchronization primitives). A Semaphore is an integer variable. There are two types of semaphores : Binary Semaphores (mutex) and Counting Semaphores

Binary Semaphores (Not mutex!!!): They can only be either 0 or 1. They are also known as **mutex locks**, as the locks can provide **mutual exclusion**. All the processes can share the **same mutex semaphore that is initialized to 1**. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section.

When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.

Counting Semaphores : They can have any value and are not restricted over a certain domain. They can be used to control access a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Using Mutex:

A mutex is costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. 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.

Using 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.

Difference:
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 and signals the call processing task to wakeup.

Software method to reduce power consumption of embedded devices

Dynamic change of the processor frequency
The first and most widely-used method for power reduction is the dynamic change of the processor frequency during the execution of the firmware.

Selective peripheral clocking
The trend in microcontrollers is to have individual control over the clocking of each peripheral [1]. This helps to decrease significantly the dynamic power of the whole chip.

Sleep and Deep Sleep modes of operation in Cortex-M core

In sleep mode the clocking to the microprocessor is stopped [3]. The peripheral modules such as UART, SPI, Ethernet and so on continue to operate independently and only when data processing is required the core is awakened from this mode. This is done with the help of interrupts.

How to realize semaphore?

Initialize a semaphore
sem_init(3RT)

Increment a semaphore
sem_post(3RT)

Block on a semaphore count
sem_wait(3RT)

Decrement a semaphore count
sem_trywait(3RT)

Destroy the semaphore state
sem_destroy(3RT)

Prototype:
int sem_init(sem_t *sem, int pshared, unsigned int value);
#include <semaphore.h>

```
sem_t sem;
int pshared;
int ret;
int value;

/* initialize a private semaphore */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

Use sema_init(3THR) to initialize the semaphore variable pointed to by sem to value amount. If the value of pshared is zero, then the semaphore cannot be shared between processes. If the value of pshared is nonzero, then the semaphore can be shared between processes. (For Solaris threads, see sema_init(3THR).)

Multiple threads must not initialize the same semaphore.

A semaphore must not be reinitialized while other threads might be using it.

How to multithread?

#include <pthread.h>

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
...

void *get_result(void *param)  // param is a dummy pointer
{
...
}

int main()
{
...
pthread_t *tid = malloc( ntimes * sizeof(pthread_t) );

for( i=0; i < ntimes; i++ )
   pthread_create( &tid[i], NULL, get_result, NULL );

... // do some tasks unrelated to result

for( i=0; i < ntimes; i++ )
   pthread_join( tid[i], NULL );
...
}
```

Pass by reference vs. Pass by pointer

References are generally implemented using pointers. A reference is same object, just with a different name and reference must refer to an object. Since references can't be NULL, they are safer to use.

A pointer can be re-assigned while reference cannot, and must be assigned at initialization only. Pointer can be assigned NULL directly, whereas reference cannot.
Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.

A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.

A pointer to a class/struct uses '->'(arrow operator) to access it's members whereas a reference uses a '.'(dot operator)

A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly.

Overall, Use references when you can, and pointers when you have to. **But if we want to write C code that compiles with both C and a C++ compiler, you'll have to restrict yourself to using pointers.**

Describe watch-dog timer. (Qualcomm)

A watchdog timer (sometimes called a computer operating properly or COP timer, or simply a watchdog) is **an electronic timer that is used to detect and recover from computer malfunctions**. During normal operation, the **computer regularly resets the watchdog timer to prevent it from elapsing**, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to invoke a reboot if it hangs; it must be self-reliant. For example, remote embedded systems such as space probes are not physically accessible to human operators; these could become permanently disabled if they were unable to autonomously recover from faults. A watchdog timer is usually employed in cases like these. Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service attacks.

Describe virtual memory

Virtual Memory is a storage allocation scheme in which **secondary memory can be addressed as though it were part of main memory**. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps **memory addresses used by a program, called virtual addresses,** into physical addresses in computer memory.

1). All memory references within a process are logical addresses that are **dynamically translated into physical addresses at run time**. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.

2). A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Advantages :

- **More processes may be maintained in the main memory:** Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A **process may be larger than all of main memory:** One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.
- **It allows greater multiprogramming levels by using less of the available (primary) memory for each process.**

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are reference .

Difference between Thread and Process

The typical difference is that threads within the same process run in a **shared memory space**, while processes run **in separate memory spaces**.

Threads are not independent of one other like processes as a result threads shares with other threads their **code section, data section** and **OS resources** like open files and signals. But, like process, a thread has its own **program counter (PC), a register set, and a stack space**.

Advantages of Thread over Process

1). **Responsiveness**: If the process is divided into multiple threads, if one thread completed its execution, then its output can be immediately responded.
2). **Faster context switch**: Context switch time between threads is less compared to process context switch. Process context switch is more overhead for CPU.
3). **Effective Utilization of Multiprocessor system**: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
4). **Resource sharing**: Resources like code, data and file can be shared among all threads within a process.
Note : stack and registers can't be shared among the threads. Each thread have its own stack and registers.
5). **Communication**: Communication between multiple thread is easier as thread shares common address space. while in process we have to follow some specific communication technique for communication between two process.
6). **Enhanced Throughput of the system**: If process is divided into multiple threads and each thread function is considered as one job, then the number of jobs completed per unit time is increased. Thus, increasing the throughput of the system.

What are dangling pointers? where to use them?

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. A dangling pointer is a pointer that points to invalid data or to data which is not valid anymore. (point to unallocated or already freed memory). Need to set pointers to be NULL after free().

Dangling is different from a wild pointer which is basically uninitialized pointer.

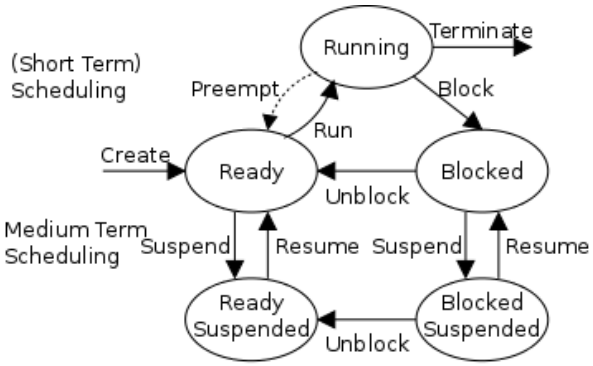
Explain process state Diagram ?

New state : Initially process will be in New state . It means process is under creation or process is being created .

Ready state or Wait: Once the process is created it will be moved on to ready state . In the ready state there will be multiple number of process

Running state : One of the process will be selected from the ready state and dispatched on to the running state .When the process in the running state it occupied the CPU and executing the instruction of the process and performing CPU time .In the running state one process at any point of the time .

Block state : If the running process required any input output then it will come to wait state .In the wait state there will be multiple number of process .It means multiple process will perform input output operation simultaneously .



Block is also known as sleep request, or P.
Unblock (a.k.a wakup, release, V) is done by another task .

Suspended Ready/wait : If the resources are not sufficient to maintain the process in the ready state the some of the process will suspended and they will be moved on to suspended ready state. In systems that support virtual memory, a process may be swapped out, that is, removed from main memory and placed on external storage by the scheduler. From here the process may be swapped back into the waiting state.

Suspend Block state : If the resources are not sufficient to maintain the process in the wait or block state then some of the process (Low priority) will be suspended and they will be moved to the suspended waiting or block state. In this process is reside in the backing store after swap out from main memory.

Terminate:

A process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a zombie process until its parent process calls the **wait system call to read its exit statu**s, at which point the process is removed from the process table, finally ending the process's lifetime. **If the parent fails to call wait, this continues to consume the process table entry (concretely the process identifier or PID), and causes a resource leak.**

Memory Mapping

Memory Mapping is a **CPU-to-device communication methods**, different than DMA which device directly talk to memory, bypass the CPU. Compare to interrupt, memory mapping is CPU-initiated while interrupt is device initiated.

I/O operations can slow memory access if the address and data buses are shared. This is because the peripheral device is usually much slower than main memory. In some architectures, port-mapped I/O operates via a dedicated I/O bus, alleviating the problem.

One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand (loading an operand from a memory location, storing the result to a memory location, or both) can be used with I/O device registers as well.

Memory-mapped file is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource. This resource is typically a file that is physically present on disk, but can also be a device, shared memory object, or other resource that the operating system can reference through **a file descriptor.** Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory.

The primary benefit of memory mapping a file is increasing I/O performance, especially when used on large files. For small files, memory-mapped files can result in a waste of slack space[1] as memory maps are always aligned to the page size, which is mostly 4 KiB. Therefore, a 5 KiB file will allocate 8 KiB and thus 3 KiB are wasted. Accessing memory mapped files is faster than using direct read and write operations for two reasons. Firstly, a system call is orders of magnitude slower than a simple change to a program's local memory. Secondly, in most operating systems the memory region mapped actually is the kernel's page cache (file cache), meaning that no copies need to be created in user space.

Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) (which is also called isolated I/O) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer.

Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation may be permanent or temporary.

Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the in and out instructions found on microprocessors based on the x86 and x86-64 architectures. Different forms of these two instructions can copy one, two or four bytes (outb, outw and outl, respectively) between the EAX register or one of that register's subdivisions on the CPU and a specified I/O port which is assigned to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

Fragmentation

In computer storage, fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both. The exact consequences of fragmentation depend on the specific system of storage allocation in use and the particular form of fragmentation. In many cases, fragmentation leads to storage space being "wasted", and in that case the term also refers to the wasted space itself.

There are three different but related forms of fragmentation: external fragmentation, internal fragmentation, and data fragmentation, which can be present in isolation or conjunction. Fragmentation is often accepted in return for improvements in speed or simplicity.

Internal fragmentation
Due to the rules governing memory allocation, more computer memory is sometimes allocated than is needed. For example, memory can only be provided to programs in chunks divisible by 4, 8 or 16, and as a result if a program requests perhaps 23 bytes, it will actually get a chunk of 32 bytes. When this happens, the excess memory goes to waste. In this scenario, the unusable memory is contained within an allocated region. This arrangement, termed fixed partitions, suffers from inefficient memory use - any process, no matter how small, occupies an entire partition. This waste is called internal fragmentation.

External fragmentation
External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory. It is a weakness of certain storage allocation algorithms, when they fail to order memory used by programs efficiently. The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small individually to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

Data fragmentation
Data fragmentation occurs when a collection of data in memory is broken up into many pieces that are not close together. It is typically the result of attempting to insert a large object into storage that has already suffered external fragmentation.

Floating Point Arithmetic

https://www.youtube.com/watch?v=8afbTaA-gOQ

Fixed point is a representation of floating point number in integer format. So operations can be applied on the number just like on integers. The advantage of using this is that floating point

arithmetic is costlier (processing power). Newer processors have dedicated FPUs (floating point units) for handling that.

So fixed point arithmetic is when processing power is limited, and a little precision loss doesn't cause a havoc.

Inline function vs. Macro

What is inline function :
The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime. NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.

Why to use –
When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. Too much run time overhead.

The C++ inline function provides an alternative. With inline keyword, the compiler replaces the function call statement **with the function code itself** (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.

Pros :-
1. It speeds up your program by avoiding function calling overhead.
2. It save overhead of variables push/pop on the stack, when function calling happens.
3. It save overhead of return call from a function.
4. It increases locality of reference by utilizing instruction cache.
5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)

Cons :-
1. It increases the executable size due to code expansion.
2. C++ inlining is resolved at compile time. Which means if you change the code of the inlined function, you would need to recompile all the code using it to make sure it will be updated
3. When used in a header, it makes your header file larger with information which users don’t care.
4. As mentioned above it increases the executable size, which may cause thrashing in memory. More number of page fault bringing down your program performance.

5. Sometimes not useful for example in embedded system where large executable size is not preferred at all due to memory constraints.

Macro vs. Inline function
Preprocessor macros are just substitution patterns applied to your code. They can be used almost anywhere in your code because they are replaced with their expansions before any compilation starts.

Inline functions are actual functions whose body is directly injected into their call site. They can only be used where a function call is appropriate.

- Macros are not type safe, and can be expanded regardless of whether they are syntatically correct - the compile phase will report errors resulting from macro expansion problems.
- Macros can be used in context where you don't expect, resulting in problems
- Macros are more flexible, in that they can expand other macros - whereas inline functions don't necessarily do this.
- Macros can result in side effects because of their expansion, since the input expressions are copied wherever they appear in the pattern.
- Inline function are not always guaranteed to be inlined - some compilers only do this in release builds, or when they are specifically configured to do so. Also, in some cases inlining may not be possible.
- Inline functions can provide scope for variables (particularly static ones), preprocessor macros can only do this in code blocks {...}, and static variables will not behave exactly the same way.

How to allocate memory in kernel?

Memory allocation in Linux kernel is different from the user space counterpart. The following facts are noteworthy,

Kernel memory is not pageable.
Kernel memory allocation mistakes can cause system oops (system crash) easily.
Kernel memory has limited hard stack size limit.
There’re two ways to allocate memory space for a kernel process, **statically from the stack or dynamically from the heap.**

There’re two functions available to allocate memory from heap in Linux kernel process,

1. **vmalloc**

It’s Linux kernel’s version of malloc() function, which is used in user space. Like malloc, the function allocates virtually contiguous memory that may or may not physically contiguous.

To free the memory space allocated by vmalloc, one simply call vfree().

2. kmalloc
kmalloc allocates a region of physically contiguous (also virtually contiguous) memory and return the pointer to the allocated memory. It returns NULL when the operation fails.

The behavior of kmalloc is dependent on the second parameter flags. Here only the two most popular flags are introduce:

GFP_KERNEL: this flag indicates a normal kernel memory allocation.
GFP_ATOMIC: this flag indicates the kmalloc function is atomic operation.

To free up the memory allocated by kmalloc, one can use kfree defined as below.

vmalloc allocates virtually contiguous memory space (not necessarily physically contiguous), while kmalloc allocates physically contiguous memory (also virtually contiguous). Most of the memory allocations in Linux kernel are done using kmalloc, due to the following reasons:

On many architectures, hardware devices don't understand virtual address. Therefore, their device drivers can only allocate memory using kmalloc.
kmalloc has better performance in most cases because physically contiguous memory region is more efficient than virtually contiguous memory. The reason behind this is not covered here, interested readers can search for Linux memory management articles.

Struct padding and packing in C/C++

In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called **structure padding**. If not aligned, it will require more memory read cycle to process the date type. e.g. a 32-bit machine reads 4 byte (word) each time, it not aligned well, one integer may require two reads.

To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.
Because of this structure padding concept in C, size of the structure is always not same as what we think.

Structure packing suppresses structure padding, padding used when alignment matters most, packing used when space matters most.

Packing, on the other hand prevents compiler from doing padding - this has to be explicitly requested - under GCC it's __attribute__((__packed__)), so the following:

```
struct __attribute__((__packed__)) mystruct_A {
   char a;
   int b;
   char c;
};
```
would produce structure of size 6 on a 32-bit architecture.

How OS detect stack overflow?

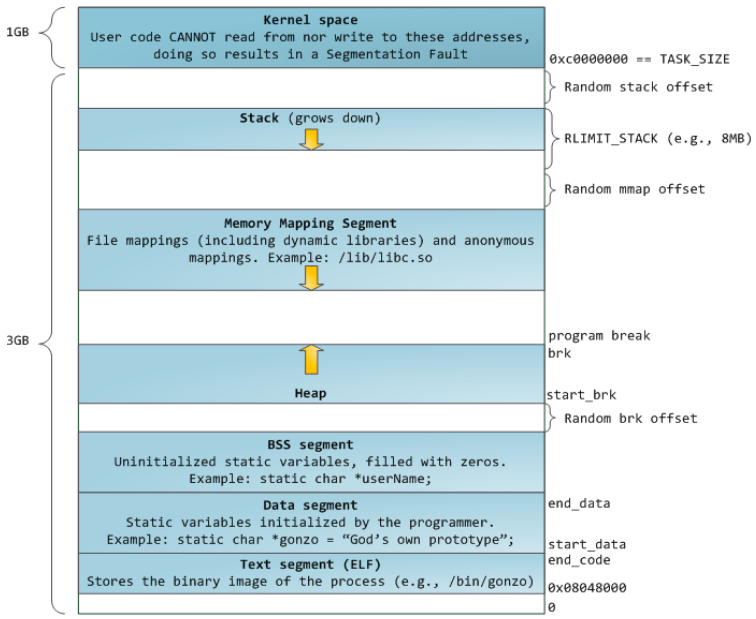
It is possible to exhaust the area mapping the stack by pushing more data than it can fit. This triggers a page fault that is handled in Linux by expand_stack(), which in turn calls acct_stack_growth() to check whether it's appropriate to grow the stack. If the stack size is below RLIMIT_STACK (usually 8MB), then normally the stack grows and the program continues merrily, unaware of what just happened. This is the normal mechanism whereby stack size

adjusts to demand. However, if the maximum stack size has been reached, we have a stack overflow and **the program receives a Segmentation Fault**. While the mapped stack area expands to meet demand, it does not shrink back when the stack gets smaller. Like the federal budget, it only expands.

What is in virtual address space?

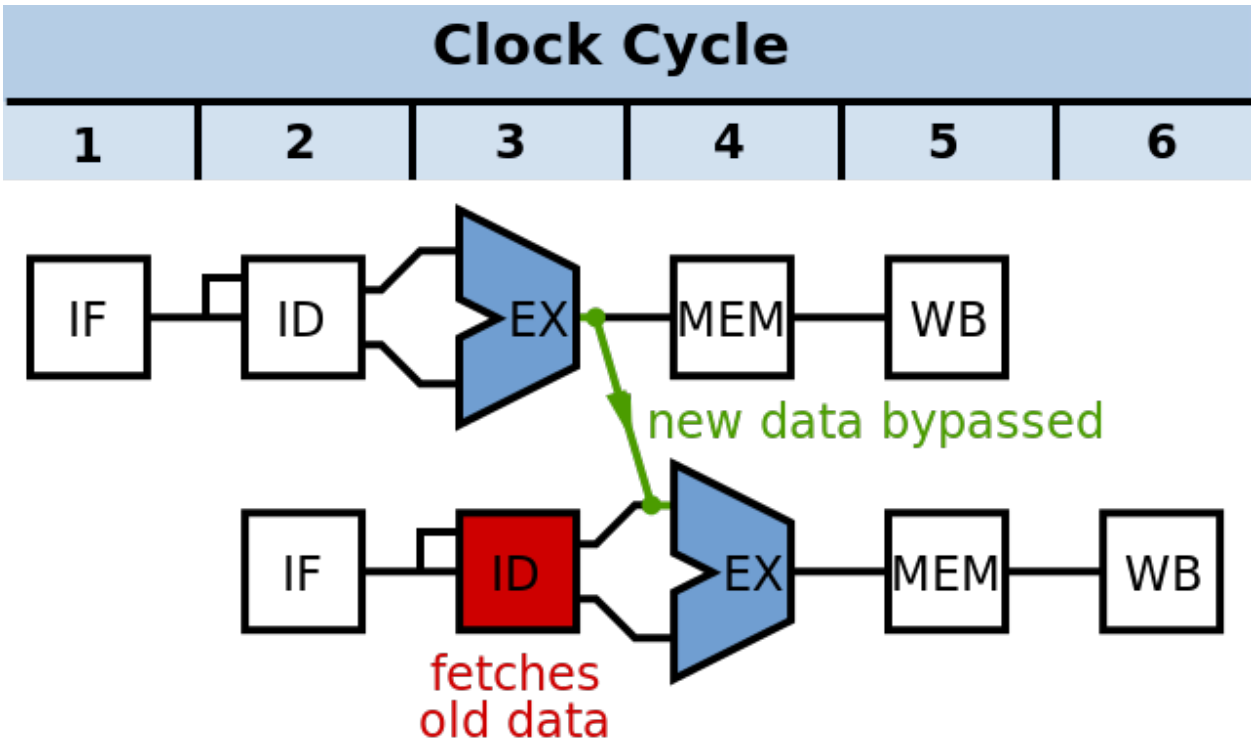
The virtual address space for a process is **the set of virtual memory addresses that it can use.** The address space for each process is private and cannot be accessed by other processes unless it is shared.

Each process in a multi-tasking OS runs in its own memory sandbox. This sandbox is the virtual address space, which in 32-bit mode is always a 4GB block of memory addresses. These virtual addresses are **mapped to physical memory by page tables**, which are maintained by the operating system kernel and consulted by the processor. Each process has its own set of page tables, but there is a catch. Once virtual addresses are enabled, they apply to all software running in the machine, including the kernel itself. Thus a portion of the virtual address space must be reserved to the kernel



What is pipeline? 5 stage pipeline in RISC? data hazard?

Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.



Dependencies in a pipelined processor

There are mainly three types of dependencies possible in a pipelined processor. These are :

1) Structural Dependency
This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

solution: To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

2) **Control Dependency**
This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

solution: To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

3) Data Dependency (Data Hazard)
Example: Let there be two instructions I1 and I2 such that:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2

When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I2 tries to read the data before I1 writes it, therefore, I2 incorrectly gets the old value from I1.

solution: operand forwarding

These dependencies may introduce stalls in the pipeline.
Stall : A stall is a cycle in the pipeline without new input.

Difference between linked list and array? when to use linked list?

Following are the points in favour of Linked Lists.
1) **The size of the arrays is fixed**: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2) **Inserting a new element in an array of elements is expensive**, because room has to be created for the new elements and to create room existing elements have to shifted.

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays:
1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:
1) **Random access is not allowed**. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) **Extra memory space for a pointer** is required with each element of the list.
3) **Arrays have better cache locality** that can make a pretty big difference in performance.

How are interrupts handled in RTOS? (Qualcomm)

In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a **function called an interrupt handler (or an interrupt service routine, ISR)** to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.[1] There are two types of interrupts: hardware interrupts and software interrupts.

User mode and kernel mode

In any modern operating system, the CPU is actually spending time in two very distinct modes:

In Kernel mode, the executing code has complete and unrestricted access to the underlying 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 are catastrophic; they will halt the entire PC.
User Mode

In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

Buffer overflows and impacts/problems

A buffer is a temporary area for data storage. When more data (than was originally allocated to be stored) gets placed by a program or system process, the extra data overflows. It causes some of that data to leak out into other buffers, which can corrupt or overwrite whatever data they were holding.

In a buffer-overflow attack, the extra data sometimes holds specific instructions for actions intended by a hacker or malicious user; for example, the data could trigger a response that damages files, changes data or unveils private information.

Attacker would use a buffer-overflow exploit to take advantage of a program that is waiting on a user's input. There are two types of buffer overflows: stack-based and heap-based. Heap-based, which are difficult to execute and the least common of the two, attack an application by flooding the memory space reserved for a program. Stack-based buffer overflows, which are more common among attackers, exploit applications and programs by using what is known as a stack: memory space used to store user input.

Let us study some real program examples that show the danger of such situations based on the C.

In the examples, we do not implement any malicious code injection but just to show that the buffer can be overflow. Modern compilers normally provide overflow checking option during the compile/link time but during the run time it is quite difficult to check this problem without any extra protection mechanism such as using exception handling.

```
// A C program to demonstrate buffer overflow
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

```
    // Reserve 5 byte of buffer plus the terminating NULL.
    // should allocate 8 bytes = 2 double words,
    // To overflow, need more than 8 bytes...
    char buffer[5];  // If more than 8 characters input
                     // by user, there will be access
                     // violation, segmentation fault

    // a prompt how to execute the program...
    if (argc < 2)
    {
        printf("strcpy() NOT executed....\n");
        printf("Syntax: %s <characters>\n", argv[0]);
        exit(0);
    }

    // copy the user input to mybuffer, without any
    // bound checking a secure version is srtcpy_s()
    strcpy(buffer, argv[1]);
    printf("buffer content= %s\n", buffer);

    // you may want to try strcpy_s()
    printf("strcpy() executed...\n");

    return 0;
}
```

Input : 12345678 (8 bytes), the program run smoothly.
Input : 123456789 (9 bytes)
"Segmentation fault" message will be displayed and the program terminates.

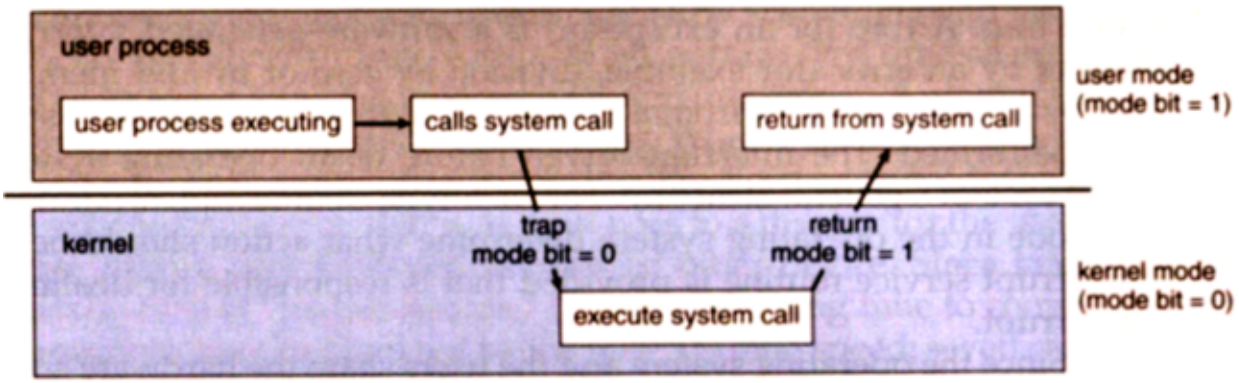
The vulnerability exists because the buffer could be overflowed if the user input (argv[1]) bigger than 8 bytes. Why 8 bytes? For 32 bit (4 bytes) system, we must fill up a double word (32 bits) memory. Character (char) size is 1 byte, so if we request buffer with 5 bytes, the system will allocate 2 double words (8 bytes). That is why when you input more than 8 bytes; the mybuffer will be over flowed

Similar standard functions that are technically less vulnerable, such as strncpy(), strncat(), and memcpy(), do exist. But the problem with these functions is that it is the programmer responsibility to assert the size of the buffer, not the compiler.

Every C/C++ coder or programmer must know the buffer overflow problem before they do the coding. A lot of bugs generated, in most cases can be exploited as a result of buffer overflow.

Difference between library call and a system call

System calls provide an interface to the services made available by an operating system.When a program makes a system call, the mode is switched from user mode to kernel mode.



The functions which are part of standard C library are known as Library functions. For example the standard string manipulation functions like strcmp(), strlen() etc are all library functions.

The functions which change the execution mode of the program from user mode to kernel mode are known as system calls. These calls are required in case some services are required by the program from kernel. For example, if we want to change the date and time of the system or if we want to create a network socket then these services can only be provided by kernel and hence these cases require system calls. For example, socket() is a system call.

System calls acts as entry point to OS kernel. There are certain tasks that can only be done if a process is running in kernel mode. Examples of these tasks can be interacting with hardware etc. So if a process wants to do such kind of task then it would require itself to be running in kernel mode which is made possible by system calls.

1) A library function is linked to the user program and executes in user space while a system call is not linked to a user program and executes in kernel space.
2) A library function execution time is counted in user level time while a system call execution time is counted as a part of system time.
3) Library functions can be debugged easily using a debugger while System calls cannot be debugged as they are executed by the kernel.

CPU scheduling algorithms

A typical process involves both I/O time and CPU time. In a uniprogramming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

First Come First Serve (FCFS): Simplest scheduling algorithm that schedules according to arrival times of processes.

Shortest Job First(SJF): Process which have the shortest burst time are scheduled first.

Shortest Remaining Time First(SRTF): It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

Round Robin Scheduling: Each process is assigned a fixed time in cyclic way.

Priority Based scheduling (Non Preemptive): In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is schedule first. If priorities of two processes match, then schedule according to arrival time.

Highest Response Ratio Next (HRRN) In this scheduling, processes with highest response ratio is scheduled. This algorithm avoids starvation.

Response Ratio = (Waiting Time + Burst time) / Burst time
Multilevel Queue Scheduling: According to the priority of process, processes are placed in the different queues. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled.

Multi level Feedback Queue Scheduling: It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue.

1) FCFS can cause long waiting times, especially when the first job takes too much CPU time.

2) Both SJF and Shortest Remaining time first algorithms may cause starvation. Consider a situation when long process is there in ready queue and shorter processes keep coming.

3) If time quantum for Round Robin scheduling is very large, then it behaves same as FCFS scheduling.

4) SJF is optimal in terms of average waiting time for a given set of processes,i.e., average waiting time is minimum with this scheduling, but problems is, how to know/predict time of next job.

Memory Management

1: Single Partition Allocation Schemes: The memory is divided into two parts. One part is kept for use by the OS and the other for use by the users.

2: Multiple Partition Schemes:
Fixed Partition: The memory is divided into fixed size partitions.
Variable Partition: The memory is divided into variable sized partitions.

Variable partition allocation schemes:
First Fit: The arriving process is allotted the first hole of memory in which it fits completely.
Best Fit: The arriving process is allotted the hole of memory in which it fits the best by leaving the minimum memory empty.
Worst Fit: The arriving process is allotted the hole of memory in which it leaves the maximum gap. Note: Best fit does necessarily give the best results for memory allocation.

Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

The Physical Address Space is conceptually divided into a number of fixed-size blocks, called frames.
The Logical address Space is also splitted into fixed-size blocks, called pages.
Page Size = Frame Size

Segmentation

A Memory Management technique in which memory is divided into variable sized chunks which can be allocated to processes. Each chunk is called a Segment.

Base Address: It contains the starting physical address where the segments reside in memory.
Limit: It specifies the length of the segment.

Paging is the concept used to implement virtual memory in the system, where some portion of secondary memory is divided into equal size pages which can be swap by the same size frames from the mail memory at the time of page fault , so that ,that portion of secondary memory can act like main memory of the system.

Segmentation is totally different concept in which **physical memory or RAM is divided into variable size logical segments which belongs to the same module**. Segmentation is done on the basis of user's requirement.

Page Replacement Algorithms

First In First Out
This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

For example, consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.
when 3 comes, it is already in memory so —> 0 Page Faults.
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.
Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.

Least Recently Used

In this algorithm page will be replaced which is least recently used.

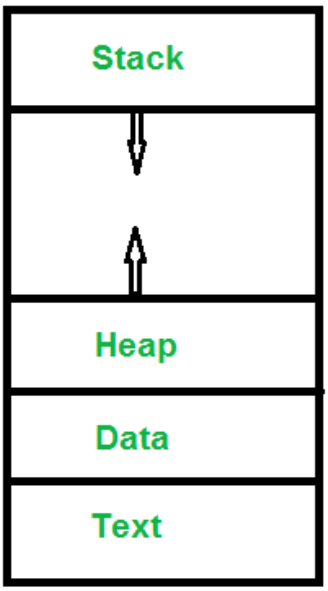
Belady's anomaly
Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3 2 1 0 3 2 4 3 2 1 0 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Page Fault

A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Memory map of program, Storage classes and their mapping

1. Text Segment:
A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.



As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = "hello world" in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = "hello world" makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:
Uninitialized data segment, often called the "**bss**" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing. Uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.
For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:
The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame"; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

5. Heap:
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single "heap area" is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Describe inheritance. (Qualcomm)

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

If we declare more number of variables than the registers available on the processor? Where they will be stored.

Register variables are a special case of automatic variables. Automatic variables are allocated storage in the memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing in the CPU. These computers often have small amounts of storage within the CPU itself where data can be stored and accessed quickly. These storage cells are called registers.

Normally, the compiler determines what data is to be stored in the registers of the CPU at what times. However, the C language provides the storage class register so that the programmer can ``suggest'' to the compiler that particular automatic variables should be allocated to CPU registers, if possible. Thus, register variables provide a certain control over efficiency of program execution. Variables which are used repeatedly or whose access times are critical, may be declared to be of storage class register.

The register class designation is merely a suggestion to the compiler. **Not all implementations will allocate storage in registers for these variables**, depending on the number of registers available for the particular computer, or the use of these registers by the compiler. **They may be treated just like automatic variables and provided storage in memory.**

Finally, even the availability of register storage does not guarantee faster execution of the program. For example, **if too many register variables are declared, or there are not enough registers available to store all of them, values in some registers would have to be moved to temporary storage in memory in order to clear those registers for other variables**. Thus, much time may be wasted in moving data back and forth between registers and memory locations. In addition, the use of registers for variable storage may interfere with other uses of registers by the compiler, such as storage of temporary values in expression evaluation. In the end, use of register variables could actually result in slower execution. Register variables should only be used if you have a detailed knowledge of the architecture and compiler for the computer you are using. It is best to check the appropriate manuals if you should need to use register variables.

What is thrashing?

Initial degree of multi programming upto some extent of point(lamda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multi programming the CPU utilization will drastically fall down and the system will spent more time

only in the page replacement and the **time taken to complete the execution of the process will increase**. This situation in the system is called as thrashing.

Causes of Thrashing :

1). High degree of multiprogramming : If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:
Let free frames = 400
Case 1: Number of process = 100
Then, each process will get 4 frames.

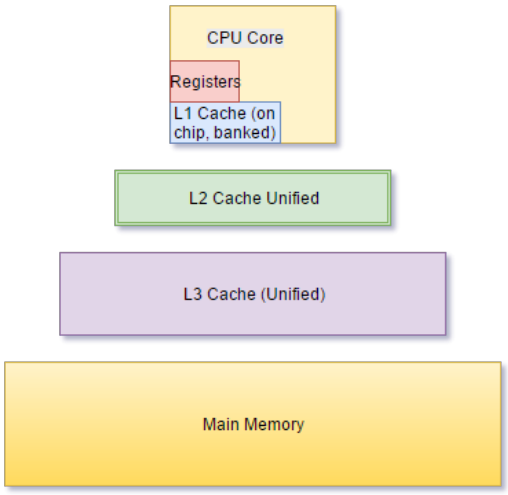
Case 2: Number of process = 400
Each process will get 1 frame.
Case 2 is a condition of thrashing, as the number of processes are increased,frames per process are decreased. Hence CPU time will be consumed in just swapping pages.

2). Lacks of Frames:If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

What is Cache?

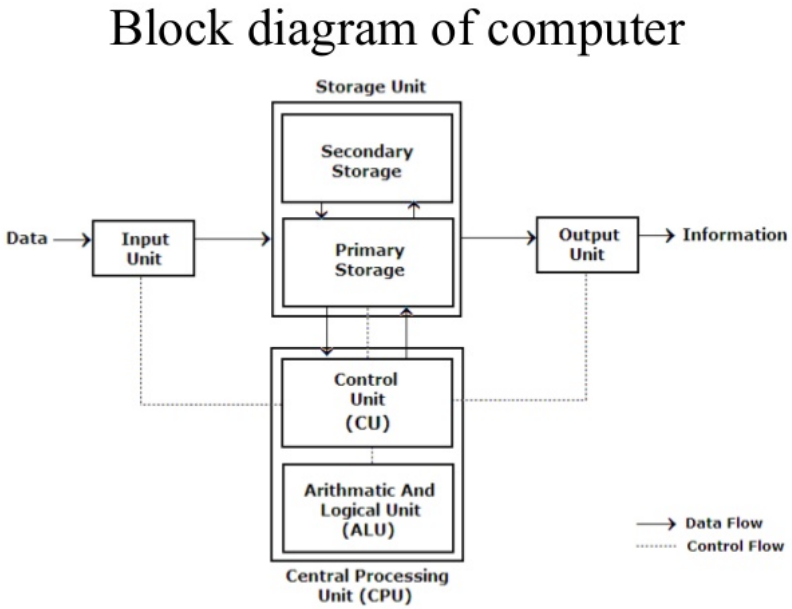
A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.).

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory.
Most modern desktop and server CPUs have at least three independent caches: **an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store, and a translation lookaside buffer (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data.** A single TLB could be provided for access to both instructions and data, or a separate Instruction TLB (ITLB) and data TLB (DTLB) can be provided.[4] The data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.; see also multi-level caches below). **However, the TLB cache is part of the memory management unit (MMU) and not directly related to the CPU caches.**



Cache size:
L1 - 32 KB (8 ~ 64 KB), L2 - 256 or 512 KB, L3 - 8 MB

Draw the block diagram of a computer and explain? How will you make a computer?



InputUnit:
Computers need to receive data and instruction in order to solve any problem. Therefore we need to input the data and instructions into the computers. The input unit consists of one or more input devices. Keyboard is the one of the most commonly used input device. Other commonly used input devices are the mouse, floppy disk drive, magnetic tape, etc. All the input devices perform the following functions.

- Accept the data and instructions from the outside world.

- Convert it to a form that the computer can understand.
- Supply the converted data to the computer system for further processing.

Storage Unit:
The storage unit of the computer holds data and instructions that are entered through the input unit, before they are processed. It preserves the intermediate and final results before these are sent to the output devices. It also saves the data for the later use. The various storage devices of a computer system are divided into two categories.

1. Primary Storage: Stores and provides very fast. This memory is generally used to hold the **program being currently executed** in the computer, the data being received from the input unit, the intermediate and final results of the program. The primary memory is temporary in nature. **The data is lost, when the computer is switched off**. In order to store the data permanently, the data has to be transferred to the secondary memory. The cost of the primary storage is more compared to the secondary storage. Therefore most computers have limited primary storage capacity.

2. Secondary Storage: Secondary storage is used like an archive. It stores several programs, documents, data bases etc. The programs that you run on the computer are first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory. Some of the commonly used secondary memory devices are Hard disk, CD, etc.,

OutputUnit:
The output unit of a computer provides the information and results of a computation to outside world. Printers, Visual Display Unit (VDU) are the commonly used output devices. Other commonly used output devices are floppy disk drive, hard disk drive, and magnetic tape drive.

ArithmeticLogicalUnit:
All calculations are performed in the Arithmetic Logic Unit (ALU) of the computer. It also does comparison and takes decision. The ALU can perform basic operations such as addition, subtraction, multiplication, division, etc and does logic operations viz, >, <, =, ‘etc. Whenever calculations are required, the control unit transfers the data from storage unit to ALU once the computations are done, the results are transferred to the storage unit by the control unit and then it is send to the output unit for displaying results.

ControlUnit:
It controls all other units in the computer. The control unit instructs the input unit, where to store the data after receiving it from the user. It controls the flow of data and instructions from the storage unit to ALU. It also controls the flow of results from the ALU to the storage unit. The control unit is generally referred as the central nervous system of the computer that control and synchronizes its working.

Spinlocks

A spinlock is a lock, and therefore a mutual exclusion (strictly 1 to 1) mechanism. It works by repeatedly querying and/or modifying a memory location, usually in an atomic manner. This

means that acquiring a spinlock is a "busy" operation that possibly burns CPU cycles for a long time (maybe forever!) while it effectively achieves “nothing".

The main incentive for such an approach is the fact that a **context switch has an overhead equivalent to spinning a few hundred** (or maybe thousand) times, so **if a lock can be acquired by burning a few cycles spinning, this may overall very well be more efficient**. Also, for realtime applications it may not be acceptable to block and wait for the scheduler to come back to them at some far away time in the future.

A spin-lock is usually used when there is low contention for the resource and the CPU will therefore only **make a few iterations before it can move on to do productive work**. The regular lock is used if the resource cannot be acquired in a reasonable time-frame. **This is done to reduce the overhead with context switches in settings where locks usually are quickly obtained.**

Spin locks perform a busy wait - i.e. it keeps running loop:

```
while (try_acquire_resource ());
...
release();
```

It performs **very lightweight locking/unlocking** but if the locking thread will be preempted by other which will try to access the same resource the second one will simply try to acquire resource until it run out of it CPU quanta.

Spin locks are a low-level synchronization mechanism suitable primarily for use on **shared memory multiprocessors.**

What if OS does not release spinlock?

Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use.

Spinlocks are, by their nature, intended for use on **multiprocessor systems**, although a uniprocessor workstation running a preemptive kernel behaves like SMP, as far as concurrency is concerned. **If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock**. For this reason, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status. Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking.

Context of a process

The context of a process includes its **address space, stack space, virtual address space, register set image (e.g. Program Counter (PC), Stack Pointer (SP), Instruction Register (IR), Program Status Word (PSW) and other general processor registers), updating**

profiling or accounting information, making a snapshot image of its associated kernel data structures and updating the current state of the process (waiting, ready, etc).

1. Process Id: A unique identifier assigned by operating system
2. Process State: Can be ready, running, .. etc
3. CPU registers: Like Program Counter (CPU registers must be saved and restored when a process is swapped out and in of CPU)
4. address space, stack and etc..
5. Accounts information:
6. I/O status information: For example devices allocated to process, open files, etc
8. CPU scheduling information: For example Priority (Different processes may have different priorities, for example a short process may be assigned low priority in shortest job first scheduling)

This state information is saved in the process's process control block which is then moved to the appropriate scheduling queue. The new process is moved to the CPU by copying the PCB info into the appropriate locations (e.g. the program counter is loaded with the address of the next instruction to execute).

Context switching

Switching of CPU to another process means saving the state of old process and loading saved state for new process. **Context switching involved mode switch, because it can only occur in kernel mode.**

In Context Switching the process is stored in the Process Control Block to serve the new process, so that old process can be resumed from the same part it was left.

Switching from one process to another requires a certain amount of time for doing the administration – saving and loading registers and memory maps, updating various tables and lists, etc. What is actually involved in a context switch varies between these senses and between processors and operating systems. For example, in the Linux kernel, context switching involves **switching registers, stack pointer, and program counter, but is independent of address space switching,** though in a process switch an address space switch also happens.

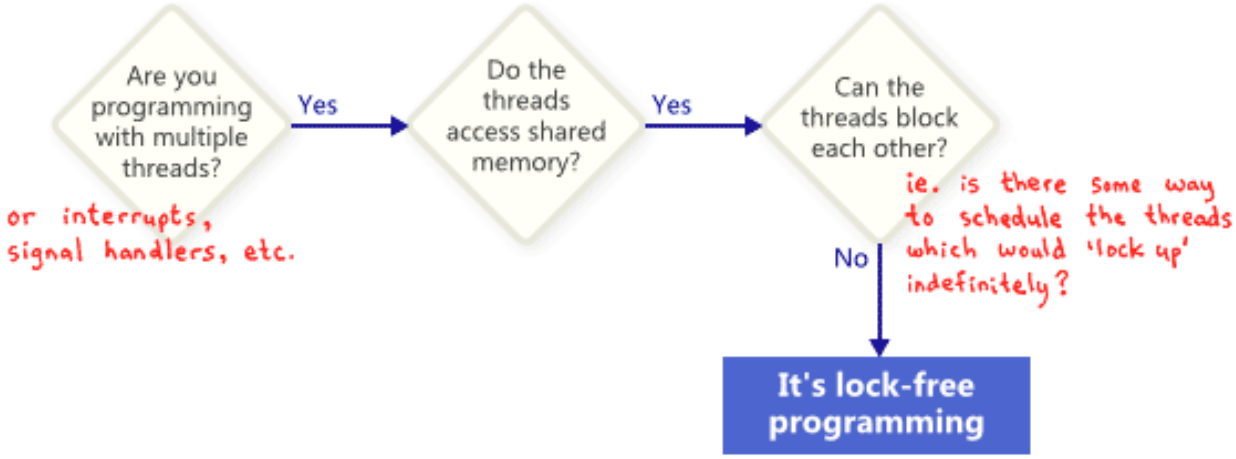
When does Context switching happen?
1. When a high priority process comes to ready state, compared to priority of running process
2. Interrupt Occurs
3. User and Kernel mode switch: (It is not necessary though)
4. Preemptive CPU scheduling used.

What is atomic programming/non-locking operation?

In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible **if it appears to the rest of the system to occur instantaneously.** It is a

safety property which ensures that operations do not complete in an unexpected or unpredictable manner.

An operation acting on shared memory is atomic if **it completes in a single step relative to other threads**. When an atomic store is performed on a shared variable, no other thread can observe the modification half-complete. When an atomic load is performed on a shared variable, it reads the entire value as it appeared at a single moment in time. Non-atomic loads and stores do not make those guarantees.



Lock-free operations:
The implication is that multiple threads can access the data structure concurrently without race conditions or data corruption, even though there are no locks.

Explain and describe how binary search tree work

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

Algorithm	**Average**	**Worst Case**
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

What is recursion? What actually happens during recursion? does the memory get stored on stack? what gets called and how does the program know from where to call?

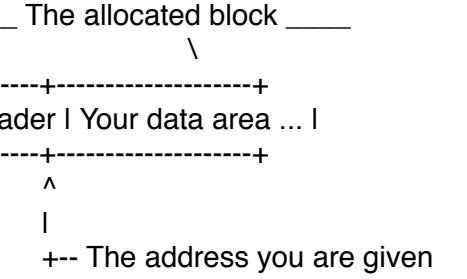
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). Most computer programming languages support recursion by allowing a function to call itself within the program text. When recursion is invoked, **a stack frame will be assigned to the invoke function to store its local variables and the return address will be put at the bottom of that frame** where recursion function could go up when it finished.

Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. **Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.**

What is free()? how does free know how much memory to de-allocate?

When you call malloc(), you specify the amount of memory to allocate. The amount of memory actually used is slightly more than this, and includes **extra information** that records (at least) how big the block is. You can't (reliably) access that other information - and nor should you :-).

When you call free(), it simply looks at the extra information to find out how big the block is.



---

what is the difference between class and object? does class or object create memory?

Many programmers still get confused by the difference between class and object. In object-oriented terminology, a Class is a template for Objects and every Object must belong to a Class. The terms "Class" and "Object" are related to one another and each term holds its own distinct meaning.

A class is a construct that **defines a collection of properties and methods in a single unit**, which does not change during the execution of a program. Objects are created and eventually destroyed during the execution of a program, so they only live in the program for a short time. While objects are "living" their attributes may also be changed at execution of a program.

Every object belongs to a class and every class contains one or more related objects (instance of a class). That means, a Class is created once and Object is created from the same Class many time as they require. There is **no memory space allocation for a Class** when it is crated, while **memory space is allocated for an Object when it is created.**

How post increment works.

Depends on how you use them.

- i++ makes a copy, increases i, and returns the copy (old value).
- ++i increases i, and returns i.
In your example it is all about speed. ++i will be the faster than i++ since it doesn't make a copy.

However a compiler will probably optimize it away since you are not storing the returned value from the increment operator in your example, but this is only possible for fundamental types like a int.

---

what is virtual function?

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function. Virtual functions should be accessed using pointer or reference of base class type to achieve **run time polymorphism.**

Virtual functions allow us to **create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.** For example, consider a employee management software for an organization, let the code has a simple base class Employee , the class contains virtual functions like raiseSalary(), transfer(), promote(),.. etc. Different types of employees like Manager, Engineer, ..etc may have their own implementations of the virtual functions present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that function would be called.

Class with pure virtual function is **abstract base class** which cannot **create a object of its own.** But its derived classes can still use the polymorphism feature. A class that declares or inherits a virtual function is called a **polymorphic class**.

What is the difference between the stack and the heap

Stack is used for **static memory allocation** and Heap for **dynamic memory allocation**, both **stored in the computer's RAM** .

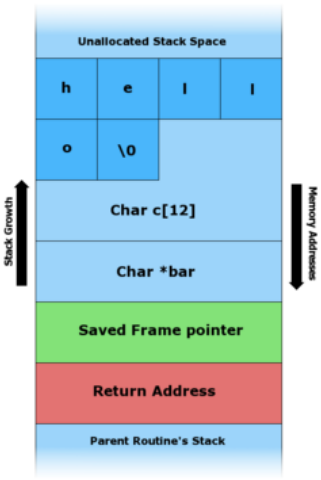
Variables allocated on the **stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.** When a function or a method calls another function which in turns calls another function etc., the execution of all those functions remains suspended until the very last function returns its value. The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

Variables allocated on the heap have their memory **allocated at run tim**e and accessing this memory is a bit slower, but the heap size is **only limited by the size of virtual memory** . Element of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.

You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

In a multi-threaded situation each thread will have its **own completely independent stack** but they will **share the heap**. Stack is thread specific and Heap is application specific. The stack is important to consider in exception handling and thread executions.

Stack Overflow



In software, a stack buffer overflow or stack buffer overrun occurs when **a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer**.[1][2] Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than what is actually allocated for that buffer. This almost always results in corruption of adjacent data on the stack, and in cases where the overflow was triggered by mistake, will often cause the program to crash or operate incorrectly. Stack buffer overflow is a type of the more general programming malfunction known as buffer overflow (or buffer overrun).[1]

What happens when you try to free a null pointer

If ptr is a null pointer, no action occurs. free(NULL) is safe.

Difference between CDMA and GSM technologies

1. Technology
The CDMA is **based on spread spectrum technology** which makes the optimal use of available bandwidth. It allows each user to transmit over the entire frequency spectrum all the time. On the other hand GSM operates on the wedge spectrum called a carrier. This carrier is divided into a number of time slots and each user is assigned a different time slot so that until the ongoing call is finished, no other subscriber can have access to this. GSM uses both **Time Division Multiple Access (TDMA)** and **Frequency Division Multiple Access (FDMA)** for user and cell separation. TDMA provides multiuser access by chopping up the channel into different time slices and FDMA provides multiuser access by separating the used frequencies.

2. **Security**
More security is provided in CDMA technology as compared with the GSM technology as **encryption is inbuilt in the CDMA.** A unique code is provided to every user and all the conversation between two users are encoded ensuring a greater level of security for CDMA users. The signal cannot be detected easily in CDMA as compared to the signals of GSM, which are concentrated in the narrow bandwidth. Therefore, the **CDMA phone calls are more secure than the GSM calls.** In terms of encryption the GSM technology has to be upgraded so as to make it operate more securely.

3. **Data Transfer Rate**
CDMA has **faster data rate** as compared to GSM as EVDO data transfer technology is used in CDMA which offers a maximum download speed of 2 mbps. EVDO ready mobile phones are required to use this technology. GSM uses EDGE data transfer technology that has a maximum download speed of 384 kbps which is slower as compared to CDMA. For browsing the web, to watch videos and to download music, CDMA is better choice as compared to GSM. So CDMA is known to cover more area with fewer towers.

What is 3G LTE

In telecommunication, Long-Term Evolution (LTE) is a standard for high-speed wireless communication for mobile devices and data terminals, based on the GSM/EDGE and UMTS/HSPA technologies. It increases the capacity and speed using a different radio interface together with core network improvements.[1][2] The standard is developed by the 3GPP (3rd Generation Partnership Project) and is specified in its Release 8 document series, with minor enhancements described in Release 9. LTE is the upgrade path for carriers with both GSM/UMTS networks and CDMA2000 networks. The different LTE frequencies and bands used in different countries mean that only multi-band phones are able to use LTE in all countries where it is supported.

LTE is commonly marketed as 4G LTE, but it does not meet the technical criteria of a 4G wireless service, as specified in the 3GPP Release 8 and 9 document series, for LTE Advanced. The requirements were originally set forth by the ITU-R organization in the IMT Advanced specification. However, due to marketing pressures and the significant advancements that WiMAX, Evolved High Speed Packet Access and LTE bring to the original 3G technologies, ITU later decided that LTE together with the aforementioned technologies can be called 4G technologies.[3] The LTE Advanced standard formally satisfies the ITU-R requirements to be

considered IMT-Advanced.[4] To differentiate LTE Advanced and WiMAX-Advanced from current 4G technologies, ITU has defined them as "True 4G".

Long-Term Evolution Time-Division Duplex (LTE-TDD), also referred to as TDD LTE, is a 4G telecommunications technology and standard co-developed by an international coalition of companies, including China Mobile, Datang Telecom, Huawei, ZTE, Nokia Solutions and Networks, Qualcomm, Samsung, and ST-Ericsson. It is one of the two mobile data transmission technologies of the Long-Term Evolution (LTE) technology standard, the other being Frequency-Division Long-Term Evolution (LTE-FDD).

Feature	NMT	GSM	IS-95 (CDMA one)	IS-2000 (CDMA 2000)	UMTS (3GSM)	LTE
Technology	FDMA	TDMA and FDMA	CDMA	CDMA	W-CDMA	OFDMA
Generation	1G	2G	2G	3G	3G	4G
Encoding	Analog	Digital	Digital	Digital	Digital	Digital
Year of First Use	1981	1991	1995	2000 / 2002	2001	2009
Roaming	Nordics and several other European countries	Worldwide, all countries except Japan and South Korea	Limited	Limited	Worldwide	Limited
Handset interoperability	None	SIM card	None	RUIM (rarely used)	SIM card	SIM card
Common Interference	None	Some electronics, e.g. amplifiers	None	None	None	None
Signal quality/coverage area	Good coverage due to low frequencies	Good coverage indoors on 850/900 MHz. Repeaters possible. 35 km hard limit.	Unlimited cell size, low transmitter power permits large cells	Unlimited cell size, low transmitter power permits large cells	Smaller cells and lower indoors coverage on 2100 MHz; equivalent coverage indoors and superior range to GSM on 850/900 MHz.	
Frequency utilization/Call density	Very low density	0.2 MHz = 8 timeslots. Each timeslot can hold up to 2 calls (4 calls with VAMOS) through interleaving.	Lower than CDMA-2000?	1.228 MHz = 3Mbit/s	5 MHz = 2 Mbit/s. 42Mbit/s for HSPA+. Each call uses 1.8-12 kbit/s depending on chosen quality and audio complexity.	20 MHz
Handoff	Hard	Hard	Soft	Soft	Soft	Hard
Voice and Data at the same time	No	Yes GPRS Class A	No	No EVDO / Yes SVDO[2]	Yes[3]	No (data only) Voice possible through VoLTE or fallback to 2G/3G

JTAG

The Joint Test Action Group (JTAG) is an electronics industry association formed in 1985 for developing a method of verifying designs and testing printed circuit boards after manufacture. In 1990 the Institute of Electrical and Electronics Engineers codified the results of the effort in IEEE Standard 1149.1-1990, entitled Standard Test Access Port and Boundary-Scan Architecture.
JTAG implements standards for on-chip instrumentation in electronic design automation (EDA) as a complementary tool to digital simulation.[1] It specifies the use of a dedicated debug port implementing a serial communications interface for low-overhead access without requiring direct external access to the system address and data buses. The interface connects to an on-chip test access port (TAP) that implements a stateful protocol to access a set of test registers that present chip logic levels and device capabilities of various parts.

SPI and I2C

SPI (SS, SCK, MOSI, MISO)
SPI is quite straightforward – it defines features any digital electronic engineer would think of if it were to quickly define a way to communicate between 2 digital devices. SPI is a protocol on 4 signal lines (please refer to figure 1):

– A clock signal named SCLK, sent from the bus master to all slaves; all the SPI signals are synchronous to this clock signal;
– A slave select signal for each slave, SSn, used to select the slave the master communicates with;
– A data line from the master to the slaves, named MOSI (Master Out-Slave In)
– A data line from the slaves to the master, named MISO (Master In-Slave Out).

I²C (SDA, SCL)
I²C is a multi-master protocol that uses 2 signal lines. The two I²C signals are called 'serial data' (SDA) and 'serial clock' (SCL). There is no need of chip select (slave select) or arbitration logic. Virtually any number of slaves and any number of masters can be connected onto these 2 signal lines and communicate between each other using a protocol that defines:

– 7-bits slave addresses: each device connected to the bus has got such a unique address;
– data divided into 8-bit bytes
– a few control bits for controlling the communication start, end, direction and for an acknowledgment mechanism.
The data rate has to be chosen between 100 kbps, 400 kbps and 3.4 Mbps, respectively called standard mode, fast mode and high speed mode. Some I²C variants include 10 kbps (low speed mode) and 1 Mbps (fast mode +) as valid speeds.

Difference
SPI has higher throughput.
SPI has simple receiver hardware -> simple shift registers
SPI supports multiple slaves
SPI has lower power consumption

SPI require many pins (SS).
only 1 master on SPI bus.

I2C allows multiple masters and slaves on the bus. On the other hand SPI can only work with one master device controlling multiple slaves. In both I2C and SPI the master device controls the clock for all slaves, but an I2C slave device can modify the main bus clock.

What is TCP and UDP? What is the difference?

TCP header size: min 20 byte
UDP header size: 8 byte

TCP and UDP are both transport layer protocols. **TCP is connection oriented but UDP is not.** It means TCP has handshake mechanism that ensures proper TCP header parameters are set before the communication starts resulting in a reliable communication(error and flow control). Whereas no handshake takes place between machines communicating with UDP protocol resulting in unreliable serivice. TCP is used for file/doc transfer. But UDP is used for video/audio transfer.

Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is a connection oriented protocol, **which means the devices should open a connection before transmitting data and should close the connection gracefully after transmitting the data.**

Transmission Control Protocol (TCP) assure reliable delivery of data to the destination.

TCP Segment Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset	Res	Flags	Window Size
128	Header and Data Checksum		Urgent Pointer	
160...	Options			

UDP Datagram Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Length		Header and Data Checksum	

Transmission Control Protocol (TCP) **protocol provides extensive error checking mechanisms such as flow control and acknowledgment of data.**

Sequencing of data is a feature of Transmission Control Protocol (TCP).

Delivery of data is guaranteed if you are using Transmission Control Protocol (TCP).

Transmission Control Protocol (TCP) is comparatively slow because of these extensive error checking mechanism.

Multiplexing and Demultiplexing is possible in Transmission Control Protocol (TCP) using TCP port numbers.

Retransmission of lost packets is possible in Transmission Control Protocol (TCP).

User Datagram Protocol (UDP)

User Datagram Protocol (UDP) is **Datagram oriented protocol** with no overhead for opening a connection (using three-way handshake), maintaining a connection, and closing (terminating) a connection.

User Datagram Protocol (UDP) is **efficient for broadcast/multicast type of network transmission.**

User Datagram Protocol (UDP) has only the basic error checking mechanism using checksums.

There is no sequencing of data in User Datagram Protocol (UDP).

The delivery of data cannot be guaranteed in User Datagram Protocol (UDP).

User Datagram Protocol (UDP) is **faster, simpler and more efficient than TCP. However, User Datagram Protocol (UDP) it is less robust then TCP.**

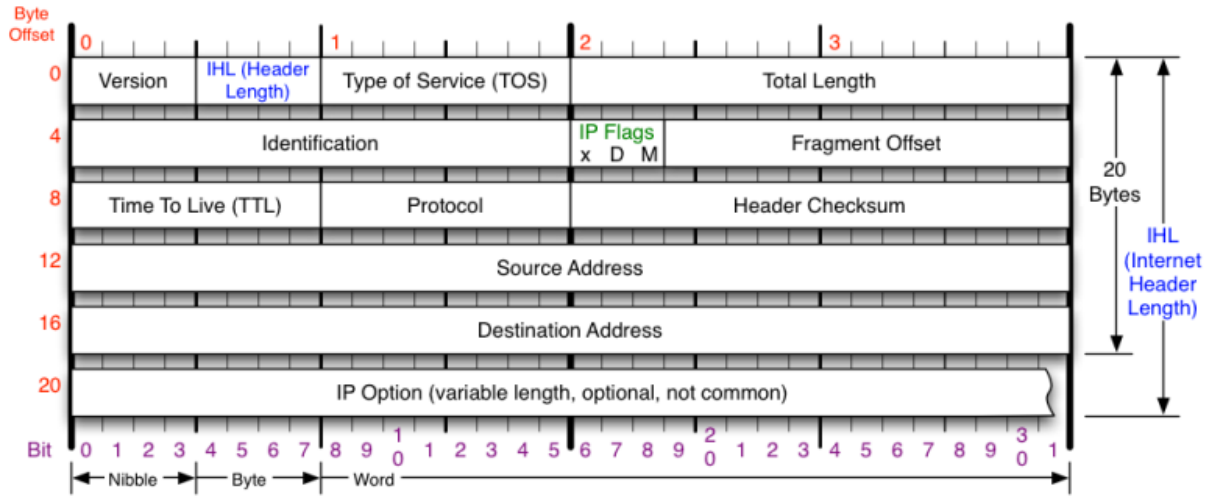
Multiplexing and Demultiplexing is possible in User Datagram Protocol (UDP) using UDP port numbers.

There is no retransmission of lost packets in User Datagram Protocol (UDP).

What is IP protocol?

The Internet Protocol (IP) is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables internet working, and essentially establishes the Internet.

IP has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers. For this purpose, IP defines packet structures that encapsulate the data to be delivered. It also defines addressing methods that are used to label the datagram with source and destination information.



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.

Header Length

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

Protocol

IP Protocol ID. Including (but not limited to):

1 ICMP	17 UDP	57 SKIP
2 IGMP	47 GRE	88 EIGRP
6 TCP	50 ESP	89 OSPF
9 IGRP	51 AH	115 L2TP

Total Length

Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.

Fragment Offset

Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.

Header Checksum

Checksum of entire IP header

IP Flags

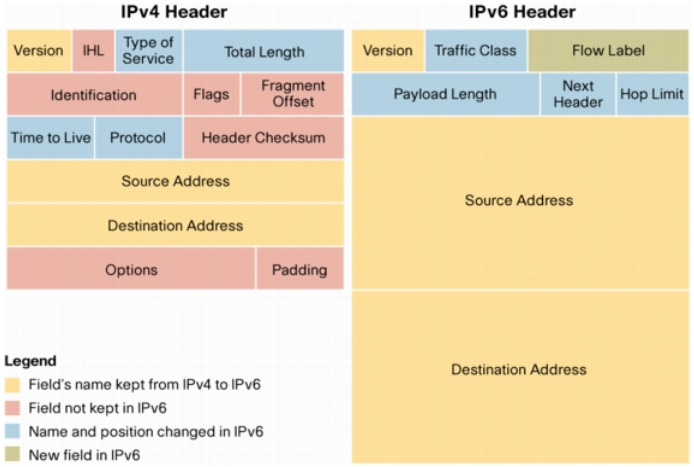
x D M

x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow

RFC 791

Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.

IPv4 vs. IPv6



IP v6 was developed by Internet Engineering Task Force (IETF) to deal with the problem of IP v4 exhaustion. IP v6 is 128-bits address having an address space of 2^128, which is way bigger than IPv4. (40 byte header size)

Difference:

- 32-bit address vs 128-bit address

- Less header fields
- No checksum
- No fragmentation in forwarding routers
- Address in Hex rather than decimal

Advantage:

1. More Efficient Routing

IPv6 **reduces the size of routing tables** and makes routing more efficient and hierarchical. IPv6 allows ISPs to aggregate the prefixes of their customers' networks into a single prefix and announce this one prefix to the IPv6 Internet. In addition, in IPv6 networks, **fragmentation is handled by the source device, rather than the router**, using a protocol for discovery of the path's maximum transmission unit (MTU).

2. More Efficient Packet Processing

IPv6's **simplified packet header makes packet processing more efficient**. Compared with IPv4, IPv6 contains no IP-level checksum, so the checksum does not need to be recalculated at every router hop. Getting rid of the IP-level checksum was possible because **most link-layer technologies already contain checksum and error-control capabilities**. In addition, most transport layers, which handle end-to-end connectivity, have a checksum that enables error detection.

3. Directed Data Flows

IPv6 **supports multicast rather than broadcast**. Multicast allows bandwidth-intensive packet flows (like multimedia streams) to be sent to multiple destinations simultaneously, saving network bandwidth. Disinterested hosts no longer must process broadcast packets. In addition, the IPv6 header has a new field, named Flow Label, that can identify packets belonging to the same flow.

4. Simplified Network Configuration

Address auto-configuration (address assignment) is built in to IPv6. A router will send the prefix of the local link in its router advertisements. A host can generate its own IP address by appending its link-layer (MAC) address, converted into Extended Universal Identifier (EUI) 64-bit format, to the 64 bits of the local link prefix.

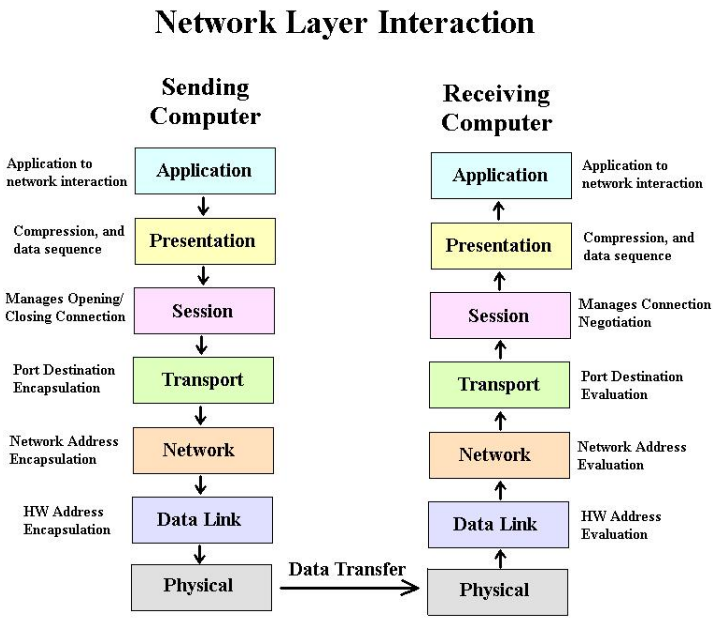
5. Support For New Services

By eliminating **Network Address Translation** (NAT), true end-to-end connectivity at the IP layer is restored, enabling new and valuable services. Peer-to-peer networks are easier to create and maintain, and services such as VoIP and Quality of Service (QoS) become more robust.

6. Security

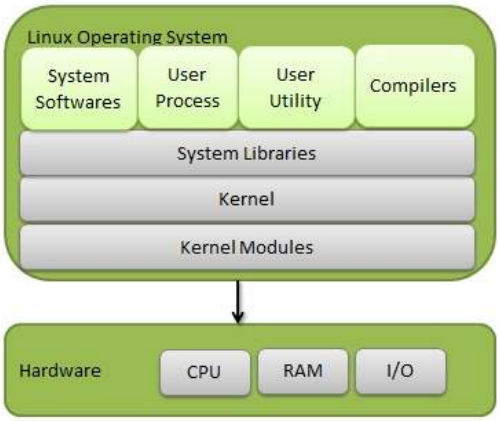
IPSec, which provides confidentiality, authentication and data integrity, is baked into in IPv6. Because of their potential to carry malware, IPv4 ICMP packets are often blocked by corporate firewalls, but ICMPv6, the implementation of the Internet Control Message Protocol for IPv6, may be permitted because IPSec can be applied to the ICMPv6 packets.

What is the network layer structure?



OS Composition?

- Process management
- I/O management
- Main Memory management
- File & Storage Management
- Protection
- Networking
- Command Interpreter



What is the difference between struct and union in C?

A structure is a user-defined data type available in C that allows to combining data items of different kinds. Structures are used to represent a record.

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Similarities between Structure and Union

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. '.' operator is used for accessing members.

Size of a struct: **At least** the size of all variable it holds.
Size of a Union: **At least** the size of the variable with the largest size

Quick Sort vs. Merge Sort

Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an **in-place** sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts

have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as **it has good locality of referenc**e when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists?
In case of linked lists the case is different mainly due to **difference in memory allocation of arrays and linked lists.** Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented **without extra space for linked lists.**

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, **we can not do random access in linked list. Quick Sort requires a lot of this kind of access**. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. **Merge sort accesses data sequentially and the need of random access is low.**

What is polymorphism, what is it for, and how is it used?

Polymorphism means taking many forms.
Here Poly means "many" and morphs means "forms".

In this we have one base form and other are overridden from the base form
It is the ability of object to take different forms of objects example: function overloading ,function overriding, virtual functions.

In this same operation has different affect on object. Example '+' for number is addition and '+' for string is concatenation.

There are two types of polymorphism:- compile time (operator and function overloading) and run time(virtual functions)

When is a null pointer used?

The null pointer is used in three ways:

1. To stop indirection in a recursive data structure.
2. As an error value.

3. As a sentinel value.

What is protected keyword? Difference from private member?

Private members are only accessible within the class defining them.

Protected members are accessible in the class that defines them and in classes that inherit from that class.

Edit: Both are also accessible by friends of their class, and in the case of protected members, by friends of their derived classes.

Edit 2: Use whatever makes sense in the context of your problem. You should try to make members private whenever you can to reduce coupling and protect the implementation of the base class, but if that's not possible then use protected members. Check C++ FAQ for a better understanding of the issue. This question about protected variables might also help.

Friends class/function

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

What is pure virtual function and abstract class?

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

Function Overloading and Overriding

Function Overloading:
It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play anyrole.
Function overloading can be considered as an example of polymorphism feature in C++.

example:
void area(int a);
void area(int a, int b);

Function Overriding:
It is the redefinition of base class function in its derived class with same signature i.e return type and parameters.

What is preemptive multitasking?

Preemptive multitasking is also known as time-shared multitasking.

Preemptive multitasking is a type of multitasking that allows computer programs to share operating systems (OS) and underlying hardware resources. It divides the overall operating and computing time between processes, and the switching of resources between different processes occurs through predefined criteria.

Preemptive multitasking allows an operating system to switch between software programs. This in turn allows multiple programs to run without necessarily taking complete control over the processor and resulting in system crashes.

Preemptive multitasking helps prevent a program from taking complete control of the computer processor and allows multiple programs to continue to operate without crashing

What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

Pre-processing
Compilation
Assembly
Linking

Pre-processing

This is the first phase through which source code is passed. This phase include:

Removal of Comments
Expansion of Macros

Expansion of the included files.

Compiling

The next step is to compile filename.i and produce an; intermediate compiled output file filename.s. This file is in assembly level instructions.

Assembly

In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved.

Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using $size filename.o and $size filename. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

Coding Questions

Reverse an 8 bit type. (Qualcomm)

Method1
Loop through all the bits of an integer. If a bit at ith position is set in the i/p no. then set the bit at (NO_OF_BITS – 1) – i in o/p. Where NO_OF_BITS is number of bits present in the given number.

```
/* Function to reverse bits of num */
unsigned int reverseBits(unsigned int num)
{
   unsigned int  NO_OF_BITS = sizeof(num) * 8;
   unsigned int reverse_num = 0, i, temp;

   for (i = 0; i < NO_OF_BITS; i++)
   {
      temp = (num & (1 << i));
      if(temp)
         reverse_num |= (1 << ((NO_OF_BITS - 1) - i));
   }

   return reverse_num;
}
```

Method 2:

```
uint32_t reverseBits(uint32_t n) {
    uint32_t m = 0;
    for (int i = 0; i < 8; i++, n >>= 1) {
        m <<= 1;
        m |= n & 1;
    }
    return m;
}
```

C Program to reverse the words in a sentence . (QualComm)

C++: (use stack)

```
class Solution {
public:
    void reverseWords(string &s) {
        stack <string> ret;
        for (int i = -1, j = 0; j <= s.size(); j ++){
            if (s[j] == ' '){
                if (i != j-1) ret.push(s.substr(i+1, j-i-1));
                i = j;
            }
            if (j == s.size()-1 && s[j] != ' ') ret.push(s.substr(i+1, j-i));
        }
        s = "";
        while(!ret.empty()){
            s += ret.top();
            ret.pop();
            if (!ret.empty()) s += " ";
        }
    }
};
```

C: (reverse the whole sentence and then reverse words one by one)

```
#include<stdio.h>

/* function prototype for utility function to
  reverse a string from begin to end  */
void reverse(char *begin, char *end);

/*Function to reverse words*/
void reverseWords(char *s)
{
  char *word_begin = s;
  char *temp = s; /* temp is for word boundry */

  /*STEP 1 of the above algorithm */
  while( *temp )
  {
    temp++;
```

```
    if (*temp == '\0')
    {
      reverse(word_begin, temp-1);
    }
    else if(*temp == ' ')
    {
      reverse(word_begin, temp-1);
      word_begin = temp+1;
    }
  } /* End of while */

   /*STEP 2 of the above algorithm */
  reverse(s, temp-1);
}

/* UTILITY FUNCTIONS */
/*Function to reverse any sequence starting with pointer
  begin and ending with pointer end  */
void reverse(char *begin, char *end)
{
  char temp;
  while (begin < end)
  {
    temp = *begin;
    *begin++ = *end;
    *end-- = temp;
  }
}
```

Count the number of set bits in an integer. (QualComm)

simple solution:

```
class Solution {
public:
    int SetBits(uint32_t n) {
        int count = 0;

        while (n){
            count += n & 1;
            n = n >> 1;
        }

        return count;
    }
};
```

Brian Kernighan's Algorithm:

Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit(including the righmost set bit). So if we subtract a number by 1 and do bitwise & with itself (n & (n-1)), we unset the righmost set bit. If we do n & (n-1) in a loop and count the no of times loop executes we get the set bit count.

Beauty of the this solution is number of times it loops is equal to the number of set bits in a given integer.

```
class Solution {
public:

int hammingWeight(uint32_t n) {
    unsigned int count = 0;
    while (n)
    {
      n &= (n-1) ;
      count++;
    }
    return count;
    }

};
```

Round a number to next largest multiple of N

```
int roundUp(int numToRound, int multiple)
{
    if (multiple == 0)
        return numToRound;

    int remainder = abs(numToRound) % multiple;
    if (remainder == 0)
        return numToRound;

    if (numToRound < 0)
        return -(abs(numToRound) - remainder);
    else
        return numToRound + multiple - remainder;
}
```

V2:

```
int roundUp(int numToRound, int multiple)
{
    assert(multiple);
    int isPositive = (int)(numToRound >= 0);
    return ((numToRound + isPositive * (multiple - 1)) / multiple) * multiple;
}
```

Aligned Malloc and aligned free (memalign)

```
#include <stdlib.h>
#include <stdio.h>

void* aligned_malloc(size_t required_bytes, size_t alignment)
{
   void* p1; // original block
   void** p2; // aligned block
   int offset = alignment - 1 + sizeof(void*);
   if ((p1 = (void*)malloc(required_bytes + offset)) == NULL)
   {
      return NULL;
   }
   p2 = (void**)(((size_t)(p1) + offset) & ~(alignment - 1));
   p2[-1] = p1;
   return p2;
}

void aligned_free(void *p)
{
   free(((void**)p)[-1]);
}

void main (int argc, char *argv[])
{
   char **endptr;
   int *p = aligned_malloc (100, 12);

   printf (“%p\n”, p);
   aligned_free (p);
}
```

swap even and odd bits of a given number

```
unsigned int swapBits(unsigned int x)
{
   // Get all even bits of x
   unsigned int even_bits = x & 0xAAAAAAAA;

   // Get all odd bits of x
   unsigned int odd_bits  = x & 0x55555555;

   even_bits >>= 1;  // Right shift even bits
   odd_bits <<= 1;   // Left shift odd bits

   return (even_bits | odd_bits); // Combine even and odd bits
}
```

Reverse string, reverse words in a string, find duplicates in an array (QualComm)

ReverseString:

```
class Solution {
public:
    string reverseString(string s) {
        int beg = 0;
        int end = s.length()- 1;
        while (beg < end)
        {
            std::swap(s[beg],s[end]);
            beg++;
            end --;
        }

        return s;

    }
};
```

Write a Link List for deleting a node (Qualcomm)

method 1:

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        if (node == NULL)
            return;

        node->val = node->next->val;
        node->next = node->next->next;
    }
};
```

method 2:

```
void deleteNode(struct ListNode* node) {
    struct ListNode* next = node->next;
    *node = *next;
    free(next);
}
```

Find the first non-recurring character in a string

Method 1:

1) Scan the string from left to right and construct the count array.

2) Again, scan the string from left to right and check for count of each character, if you find an element who's count is 1, return it.

```
int firstUniqChar(char* s) {
        int co[26] = {0};

        for (int i = 0; i < strlen(s); i ++) {
            co[s[i] - 'a'] ++;
        }

        for (int i = 0; i < strlen(s); i++){
            if (co[s[i] - 'a'] == 1) return i;
        }

        return rest == INT_MAX ? -1 : rest;
}
```

Method 2:
Instead of iterate through the original string, iterate through the storage array

```
int firstUniqChar(char* s) {
    typedef struct count {
        int cot;  // dont initialize in struct definition
        int pos;
    } count;

    count* co = (count*) calloc(26, sizeof(count));;

    int rest = INT_MAX;
    for (int i = 0; i < strlen(s); i ++) {
        co[s[i] - ‘a'].cot ++;
        co[s[i] - ‘a'].pos = i;
    }

    for (int i = 0; i < 26; i++){
        if (co[s[i] - ‘a'].cot == 1 && rest > co[s[i] - ‘a’].pos) rest = co[s[i] - ‘a'].pos;
    }

    free(co);
    return rest == INT_MAX ? 1 : rest;
}
```

Implement a queue/fifo with push/pop functionality using linked lists

```
/*
The structure of the node of the queue is
struct QueueNode
{
    int data;
    QueueNode *next;
```

```
};
and the structure of the class is
class Queue {
private:
   QueueNode *front;
   QueueNode *rear;
public :
   void push(int);
   int pop();
};
 */
/* The method push to push element into the queue*/
void Queue:: push(int x)
{
       // Your Code
       QueueNode* new_node = new QueueNode;
       new_node->data = x;
       new_node->next = NULL;

       if (!front) {
          front = new_node;
          rear = front;
       }
       else {
          rear->next = new_node;
          rear = new_node;
       }
}
/*The method pop which return the element poped out of the queue*/
int Queue :: pop()
{
       // Your Code
       if (!front) return -1;
       int ret = front->data;
       QueueNode* del = front;
       front = front->next;
       delete del;
       return ret;
}
```

Implement Strlen

```
int my_strlen(char *s)
{
          char *p=s;

          while(*p!='\0')
          p++;
```

```
        return(p-s);
}

int my_strlen(char *string)
{
        int length;
        for (length = 0; *string != '\0', string++)
                length++;

        return(length);
}
```

Swap the values of two pointers without a temp variable

Method 1:

```
void swap(int* x, int* y){
        *y = *x + *y;
        *x = *y - *x;
        *y = *y - *x;
}
```

Method 2:

```
#include <stdio.h>
int main()
{
  int x = 10, y = 5;

  // Code to swap 'x' (1010) and 'y' (0101)
  x = x ^ y;  // x now becomes 15 (1111)
  y = x ^ y;  // y becomes 10 (1010)
  x = x ^ y;  // x becomes 5 (0101)

  printf("After Swapping: x = %d, y = %d", x, y);

  return 0;
}
```

Write a function that determines if a given variable is a power of 2 or not

Method 1: (count bits, if there is only one '1', then it is a power of two)

```
bool isPowerOfTwo(int n) {
    int count = 0;
    while (n)
    {
      n &= (n-1) ;
      count++;
    }
```

```
    return count == 1;
}
```

or

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        if (n <= 0) return false;
        bitset<32> digits(n);
        return digits.count() == 1;
    }
};
```

Method 2: (recursion)

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        if (n == 1) return true;
        if (n%2 != 0 || n == 0) return false;
        return isPowerOfTwo(n/2);
    }
};
```

Reverse a linked list

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode* fir = head;
        ListNode* sec = head->next;
        ListNode* tmp = NULL;
        while (sec){
            tmp = sec;
            if (fir == head) {
                fir->next = NULL;
            }
            sec = sec->next;
            tmp->next = fir;
            fir = tmp;
```

```
        }
        head = fir;
        return head;
    }
};
```

Find a loop in a linked list.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
 /**
 use faster and lower runner solution. (2 pointers)
 the faster one move 2 steps, and slower one move only one step.
 if there's a circle, the faster one will finally "catch" the slower one.
 (the distance between these 2 pointers will decrease one every time.)

 if there's no circle, the faster runner will reach the end of linked list. (NULL)
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head == NULL || head -> next == NULL)
            return false;

        ListNode *fast = head;
        ListNode *slow = head;

        while(fast -> next && fast -> next -> next){
            fast = fast -> next -> next;
            slow = slow -> next;
            if(fast == slow)
                return true;
        }

        return false;
    }
};
```

Given a list from 1 to 100, name all the different ways you can determine if there are duplicates. Which is the most efficient?

Method 1:
use array of size 100. O(n) speed, O(n) space

suppose the input array is sample[]. create array[100].

```
for (auto i : sample){
        array[i] ++;
        if (array[i]) > 1)
        found duplicate!
}
```

Method 2:
use hash table such as unordered_map<int, int>. similar to approach 1. O(n) speed, O(n) space

Method 3:
sort the array and check whether adjacent elements are the same. O(nlogn) speed, O(1) space

Method 4:
create a bit set: bitset<100>, iterate through the array set a bit to 1 to that number position in the bit set, if already set, then we found the duplicate. O(n) speed, O(1) ? space.

```
bitset<100> bit_check;
for (auto i : array){
        if (bit_check[i] == 1) found duplicate!
        bit_ckeck[i] ^= 1;
}
```

Method 6: (apply to if there is only one pair of duplicate)
use xor to detect the duplicate pattern whether exist of the only duplicate element. O(n), O(1)

```
int i;
int dupe = 0;
for(i = 0; i < N; i++) {
   dupe = dupe ^ arr[i] ^ i;
}
```

Method 6: (apply only if there is only one pair of duplicate)
calculate the sum and subtract the expected sum val.

Write a binary search tree.

Insertion in BST:

```
struct node
{
   int key;
   struct node *left, *right;
```

```
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}
/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Search in BST:

```
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
```

```
    return search(root->left, key);
}
```

Implement strcpy function and show me if there are any limitation of this function. what if the 2 buffers passed to the strcpy function overlaps ?

Method1:

```
char *mystrcpy(char *dst, const char *src)
{
        char *ptr;
        ptr = dst;
        while(*dst++ = *src++);
        return(ptr);
}
```

The strcpy function copies src, including the terminating null character, to the location specified by dst. No overflow checking is performed when strings are copied or appended. The behavior of strcpy is undefined if the source and destination strings overlap. It returns the destination string. No return value is reserved to indicate an error.

Note that the prototype of strcpy as per the C standards is

char *strcpy(char *dst, const char *src);

Notice the const for the source, which signifies that the function must not change the source string in anyway!.

Method2:

```
char *my_strcpy(char dest[], const char source[])
{
        int i = 0;
        while (source[i] != '\0')
        {
                dest[i] = source[i];
                i++;
        }
        dest[i] = '\0';
        return(dest);
}
```

Write a own program for strstr function, optimal way

Brute force method:

```
class Solution {
```

```
public:
    int strStr(string haystack, string needle) {
        int m = haystack.length(), n = needle.length();
        if (!n) return 0;
        for (int i = 0; i < m - n + 1; i++) {
            int j = 0;
            for (; j < n; j++)
                if (haystack[i + j] != needle[j])
                    break;
            if (j == n) return i;
        }
        return -1;
    }
};
```

KMP method: http://www.matrix67.com/blog/archives/115

```
class Solution {
public:
    vector<int> KMPpreprocessing(char *s) {
        int n = strlen(s);
        vector<int> match(n,-1);
        int j = -1;
        for(int i=1; i<n; i++) {
            while(j>=0 && s[i]!=s[j+1]) j = match[j];
            if(s[i]==s[j+1]) j++;
            match[i] = j;
        }
        return match;
    }


    int strStr(char *haystack, char *needle) {
        if(!*needle) return 0;
        if(!*haystack) return -1;
        int m = strlen(haystack), n = strlen(needle);
        vector<int> match = KMPpreprocessing(needle);
        int j = -1;
        for(int i=0; i<m; i++) {
            while(j>=0 && haystack[i]!=needle[j+1]) j = match[j];
            if(haystack[i]==needle[j+1]) j++;
            if(j==n-1) return (i-n+1);
        }
        return -1;

    }
};
```

Write a program to convert a given single Linked list to BST

```
class Solution {
public:
   TreeNode *sortedListToBST(ListNode *head)
   {
         return sortedListToBST( head, NULL );
   }

private:
   TreeNode *sortedListToBST(ListNode *head, ListNode *tail)
   {
         if( head == tail )
                  return NULL;
         if( head->next == tail )    //
         {
                  TreeNode *root = new TreeNode( head->val );
                  return root;
         }
         ListNode *mid = head, *temp = head;
         while( temp != tail && temp->next != tail )    // 寻找中间节点
         {
                  mid = mid->next;
                  temp = temp->next->next;
         }
         TreeNode *root = new TreeNode( mid->val );
         root->left = sortedListToBST( head, mid );
         root->right = sortedListToBST( mid->next, tail );
         return root;
   }
};
```

Implement memcpy() on your own (and memcpy problem)

```
void myMemCpy(void *dest, void *src, size_t n)
{
  // Typecast src and dest addresses to (char *)
  char *csrc = (char *)src;
  char *cdest = (char *)dest;

  // Copy contents of src[] to dest[]
  for (int i=0; i<n; i++)
     cdest[i] = csrc[i];
}
```

Overlapping source and destination buffers

The behaviour of memcpy is undefined if the source and destination buffers overlap[1]. Use memmove instead where there is a risk of this happening.

What is memmove()?

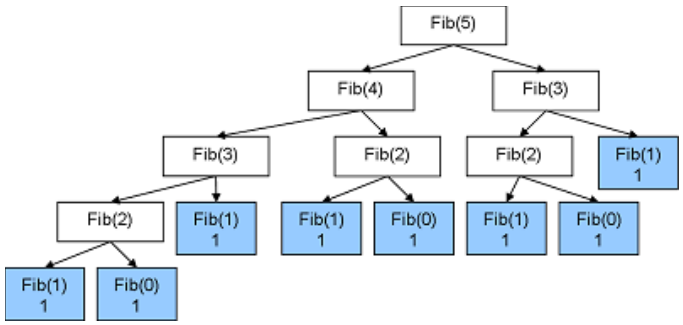
memmove() is similar to memcpy() as it also copies data from a source to destination. memcpy() leads to problems when source and destination addresses overlap as memcpy() simply copies data one by one from one location to another. For example consider below program.

The trick here is to use a temp array instead of directly copying from src to dest. The use of temp array is important to handle cases when source and destination addresses are overlapping.

```
void myMemMove(void *dest, void *src, size_t n)
{
  // Typecast src and dest addresses to (char *)
  char *csrc = (char *)src;
  char *cdest = (char *)dest;

  // Create a temporary array to hold data of src
  char *temp = new char[n];

  // Copy data from csrc[] to temp[]
  for (int i=0; i<n; i++)
     temp[i] = csrc[i];

  // Copy data from temp[] to cdest[]
  for (int i=0; i<n; i++)
     cdest[i] = temp[i];

  delete [] temp;
}
```

Calculate Fibonacci Series

```
int Fibo(n)
{
    if(n==0 || n==1)
         return n;

    return(fibo(n-1)+fibo(n-2));
 }
```



Time Complexity :

fibo(n)

||

n-level-Complete
Binary tree [upper bound]

||

2^n -1 nodes

||

2^n function calls

||

O(2^n)

Palindrome Check

```
class Solution {
public:
    bool isPalindrome(int x) {
        if(x<0 || (x!=0 && x%10==0)) return false;
        int sum=0;
        while(x>sum)
        {
            sum = sum*10+x%10;
            x = x/10;
        }
        return (x==sum) || (x==sum/10);
    }
};
```

Prime Number Check

Method 1:

```
bool isPrime(int n)
{
    // Corner case
    if (n <= 1)  return false;

    // Check from 2 to n-1
    for (int i=2; i<n; i++)
        if (n%i == 0)
            return false;

    return true;
}
```

Method 2:

Instead of checking till n, we can check till √n because a larger factor of n must be a multiple of smaller factor that has been already checked.
The algorithm can be improved further by observing that all primes are of the form 6k ± 1, with the exception of 2 and 3. This is because all integers can be expressed as (6k + i) for some integer k and for i = ?1, 0, 1, 2, 3, or 4; 2 divides (6k + 0), (6k + 2), (6k + 4); and 3 divides (6k + 3). So a more efficient method is to test if n is divisible by 2 or 3, then to check through all the numbers of form 6k ± 1. (Source: wikipedia)

```
bool isPrime(int n)
{
   // Corner cases
   if (n <= 1)  return false;
   if (n <= 3)  return true;

   // This is checked so that we can skip
   // middle five numbers in below loop
   if (n%2 == 0 || n%3 == 0) return false;

   for (int i=5; i*i<=n; i=i+6)
      if (n%i == 0 || n%(i+2) == 0)
         return false;

   return true;
}
```

Merge Sort, Quick Sort, Bubble Sort, Insertion Sort, Wiggle sort

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Time Complexity: O(n^2) as there are two nested loops.

Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

```
void swap(int *xp, int *yp)
```

```
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

```
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Worst and Average Case Time Complexity: O(n^2). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
```

```
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

Time Complexity:
Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + \Theta(n)

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is O(nLogn).

Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Insertion Sort
Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm
// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

```
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}
```

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

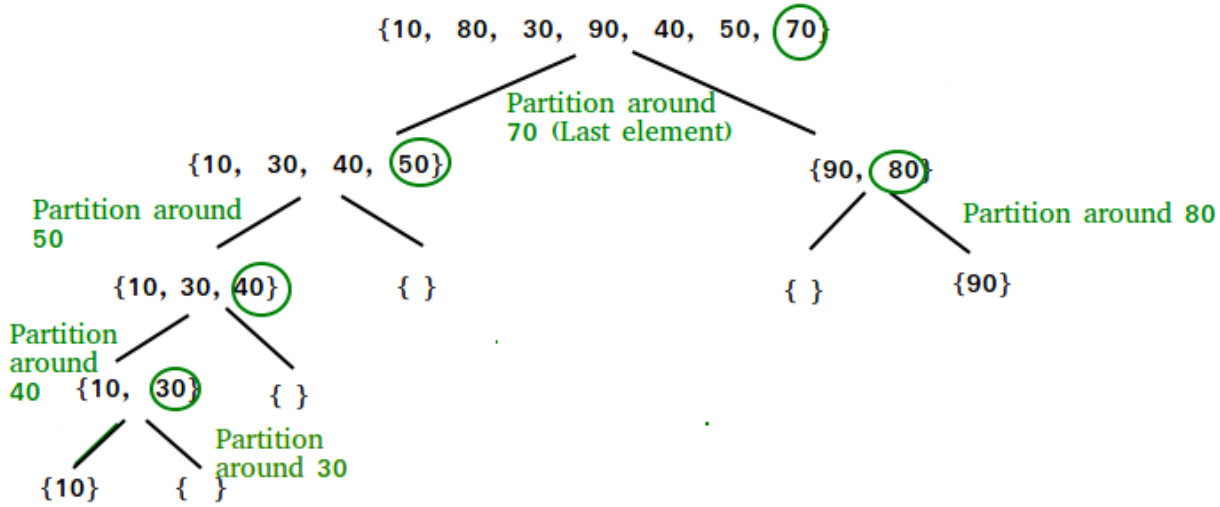
QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

Always pick first element as pivot.

Always pick last element as pivot (implemented below)
Pick a random element as pivot.
Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.




```
* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */

int partrition(int arr[], int start, int end){
    int p_index, pivot, i;
    pivot = arr[end];
    p_index = start;

    for (i = start; i < end; i++){
        if (arr[i] <= pivot){
            swap_int(&arr[i], &arr[p_index]);
            p_index ++;
        }
    }
    swap_int(&arr[end], &arr[p_index]);
    return p_index;
}

void quickSort(int arr[], int start, int end){
    if (start < end){
```

```
        int p = partrition(arr, start, end);
        printf("p index: %d\n", p);
        quickSort(arr, start, p-1);
        quickSort(arr, p+1, end);
    }
}
```

worst case: The solution of above recurrence is O(n^2).
Best case and avg case: O(nlogn)
space: O(1)

General Questions (brain teaser)

How would you design an elevator system

Find out 45 minutes with the help of two ropes. Given that one rope burns completely in 1 Hr and the rate or burning is not consistent.

Have one rope folded and the other one unfolded. Burn the unfolded rope on both ends simultaneously, it gives 30 minutes. At the time it burns out, burn the folded rope the same way to get the 15 minutes.

Why sewer caps are round?

easy to move, wont fell off.

There are 9 coins, find the lightest one. How many times you need to weight?

2 times.
coin 1,2,3 in one, 4,5,6 in another.
then one coin each side.

Behavior Questions

What are your interests?

Describe a time when you had a technical disagreement you felt strongly about, with another person on your team, and how you handled this.

Talk about a time you disagreed with a team mate, and how did you resolve the issue.

What are your weaknesses?

How do you want to see yourself after a year, you intentions, aspirations