Explain in detail the process state transition in an Unix OS, and draw a neat diagram illustrating the various states a process can go through during its execution.

Process is an instance of a program in execution. A set of processes combined together make a complete program. There are two categories of processes in Unix, namely

- **User processes**: They are operated in user mode.
- **Kernel processes**: They are operated in kernel mode.

Process States

The lifetime of a process can be divided into a set of states, each with certain characteristics that describe the process. It is essential to understand the following states now:

1. The process is currently executing in user mode.
2. The process is currently executing in kernel mode.
3. The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next.
4. The process is sleeping. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for 1/0 to complete.
5.The process is ready to run, but the swapper(process 0) must swap the process into main memory before the kernel can schedule it to execute.
6.The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3(“ready to run”) will be brought out shortly.
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the exit system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state i. the final state of a process.

Because a processor can execute only one process at a time, at most one process may be in states 1 and 2. The two states correspond to the two modes of execution, user and kernel.

The states that a Process enters in working from start till end are known as Process states. These are listed below as:

- **Created**-Process is newly created by system call, is not ready to run
- **User running**-Process is running in user mode which means it is a user process.
- **Kernel Running**-Indicates process is a kernel process running in kernel mode.
- **Zombie-** Process does not exist/ is terminated.
- **Preempted-** When process runs from kernel to user mode, it is said to be preempted.
- **Ready to run in memory-** It indicated that process has reached a state where it is ready to run in memory and is waiting for kernel to schedule it.
- **Ready to run, swapped**– Process is ready to run but no empty main memory is present
- **Sleep, swapped-** Process has been swapped to secondary storage and is at a blocked state.
- **Asleep in memory-** Process is in memory(not swapped to secondary storage) but is in blocked state.

The numbers indicate the steps that are followed.

Process Transitions

The working of Process is explained in following steps:

1. **User-running:** Process is in user-running.
2. **Kernel-running:** Process is allocated to kernel and hence, is in kernel mode.
3. **Ready to run in memory:** Further, after processing in main memory process is rescheduled to the Kernel.i.e.The process is not executing but is ready to run as soon as the kernel schedules it.

4. **Asleep in memory:** Process is sleeping but resides in main memory. It is waiting for the task to begin.
5. **Ready to run, swapped:** Process is ready to run and be swapped by the processor into main memory, thereby allowing kernel to schedule it for execution.
6. **Sleep, Swapped:** Process is in sleep state in secondary memory, making space for execution of other processes in main memory. It may resume once the task is fulfilled.
7. **Pre-empted:** Kernel preempts an on-going process for allocation of another process, while the first process is moving from kernel to user mode.
8. **Created:** Process is newly created but not running. This is the start state for all processes.
9. **Zombie:** Process has been executed thoroughly and exit call has been enabled. The process, thereby, no longer exists. But, it stores a statistical record for the process. This is the final state of all processes.

Analyze the significance of the fork system call in operating systems. Elaborate on the intricate sequence of operations carried out by the kernel when initiating the fork system call.

Fork System Call in Operating System

Last Updated : 07 Sep, 2023

In many operating systems, the fork system call is an essential operation. The fork system call allows the creation of a new process. When a process calls the fork(), it duplicates itself, resulting in two processes running at the same time. The new process that is created is called a child process. It is a copy of the parent process. The fork system call is required for process creation and enables many important features such as parallel processing, multitasking, and the creation of complex process hierarchies.

It develops an entirely new process with a distinct execution setting. The new process has its own address space, and memory, and is a perfect duplicate of the caller process.

Basic Terminologies Used in Fork System Call in Operating System

- **Process:** In an operating system, a process is an instance of a program that is currently running. It is a separate entity with its own memory, resources, CPU, I/O hardware, and files.
- **Parent Process:** The process that uses the fork system call to start a new child process is referred to as the parent process. It acts as the parent process's beginning point and can go on running after the fork.
- **Child Process:** The newly generated process as a consequence of the fork system call is referred to as the child process. It has its own distinct process ID (PID), and memory, and is a duplicate of the parent process.
- **Process ID:** A process ID (PID) is a special identification that the operating system assigns to each process.
- **Copy-on-Write:** The fork system call makes use of the memory management strategy known as copy-on-write. Until one of them makes changes to the shared memory, it enables the parent and child processes to share the same physical memory. To preserve data integrity, a second copy is then made.
- **Return Value:** The fork system call's return value gives both the parent and child process information. It assists in handling mistakes during process formation and determining the execution route.

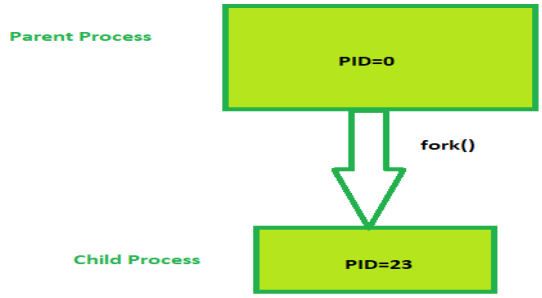
Process Creation

When the fork system call is used, the operating system completely copies the parent process to produce a new child process. The memory, open file descriptors, and other pertinent properties of the parent process are passed down to the child process. The child process, however, has a unique execution route and PID.

The copy-on-write method is used by the fork system call to maximize memory use. At first, the physical memory pages used by the parent and child processes are the same. To avoid unintentional changes, a separate copy is made whenever either process alters a shared memory page.

The return value of the fork call can be used by the parent to determine the execution path of the child process. If it returns 0 then it is executing

the child process, if it returns -1 then there is some error; and if it returns some positive value, then it is the PID of the child process.



Fork() System call

Advantages of Fork System Call

- Creating new processes with the fork system call facilitates the running of several tasks concurrently within an operating system. The system's efficiency and multitasking skills are improved by this concurrency.
- **Code reuse:** The child process inherits an exact duplicate of the parent process, including every code segment, when the fork system call is used. By using existing code, this feature encourages code reuse and streamlines the creation of complicated programmes.
- **Memory Optimisation:** When using the fork system call, the copy-on-write method optimises the use of memory. Initial memory overhead is minimised since parent and child processes share the same physical memory pages. Only when a process changes a shared memory page, improving memory efficiency, does copying take place.
- Process isolation is achieved by giving each process started by the fork system call its own memory area and set of resources. System stability and security are improved because of this isolation, which prevents processes from interfering with one another.

Disadvantages of Fork System Call

- **Memory Overhead:** The fork system call has memory overhead even with the copy-on-write optimisation. The parent process is first copied in its entirety, including all of its memory, which increases memory use.
- **Duplication of Resources:** When a process forks, the child process duplicates all open file descriptors, network connections, and other resources. This duplication may waste resources and perhaps result in inefficiencies.
- **Communication complexity:** The fork system call generates independent processes that can need to coordinate and communicate with one another. To enable data transmission across processes, interprocess communication methods, such as pipes or shared memory, must be built, which might add complexity.
- **Impact on System Performance:** Forking a process duplicates memory allocation, resource management, and other system tasks. Performance of the system may be affected by this, particularly in situations where processes are often started and stopped.

E. Describe the process of system boot and explain the role of the Init process in initializing an operating system

```
algorithm init          /* init process, process 1 of the system */
input:  none
output: none
{
        fd = open("/etc/inittab", O_RDONLY);
        while (line_read(fd, buffer))
        {
                /* read every line of file */
                if (invoked state != buffer state)
                        continue;       /* loop back to while */
                /* state matched */
                if (fork() == 0)
                {
                        execl("process specified in buffer");
                        exit();
                }
                /* init process does not wait */
                /* loop back to while */
        }

        while ((id = wait((int *) 0)) != -1)
        {
                /* check here if a spawned child died;
                 * consider respawning it */
                /* otherwise, just continue */
        }
}
```

Figure 7.31. Algorithm for Init

SYSTEM BOOT AND THE INIT PROCESS

```
algorithm start             /* system startup procedure */
input:  none
output: none
{
        initialize all kernel data structures;
        pseudo-mount of root;
        hand-craft environment of process 0;
        fork process 1:
        {
                /* process 1 in here */
                allocate region;
                attach region to init address space;
                grow region to accommodate code about to copy in;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
                /* init never gets here---as result of above change mode,
                 * init exec's /etc/init and becomes a "normal" user process
                 * with respect to invocation of system calls
                 */
        }
        /* proc 0 continues here */
        fork kernel processes;
        /* process 0 invokes the swapper to manage the allocation of
         * process address space to main memory and the swap devices.
         * This is an infinite loop; process 0 usually sleeps in the
         * loop unless there is work for it to do.
         */
        execute code for swapper algorithm;
}
```

Figure 7.30. Algorithm for Booting the System

Explain the swapping of a process between swap space and main memory?

Swapping in OS is one of those schemes which fulfill the goal of maximum utilization of **CPU** and memory management by swapping in and swapping out processes from the main memory. Swap in removes the process from **hard drive**(secondary memory) and swap out removes the process from **RAM**(main memory).

What is Swapping in Operating Systems (OS)?

Let's suppose there are several processes like P1, P2, P3, and P4 that are ready to be executed inside the ready queue, and processes P1 and P2 are very memory consuming so when the processes start executing there may be a scenario where the memory will not be available for the execution of the process P3 and P4 as there is a limited amount of memory available for process execution.

Swapping in the operating system is a memory management scheme that temporarily swaps out an idle or blocked process from the main memory to secondary memory which ensures proper memory utilization and memory availability for those processes which are ready to be executed.

When that memory-consuming process goes into a termination state means its execution is over due to which the memory dedicated to their execution becomes free Then the swapped-out processes are brought back into the main memory and their execution starts.

The area of the secondary memory where swapped-out processes are stored is called **swap space**. The swapping method forms a temporary queue of swapped processes in the secondary memory.

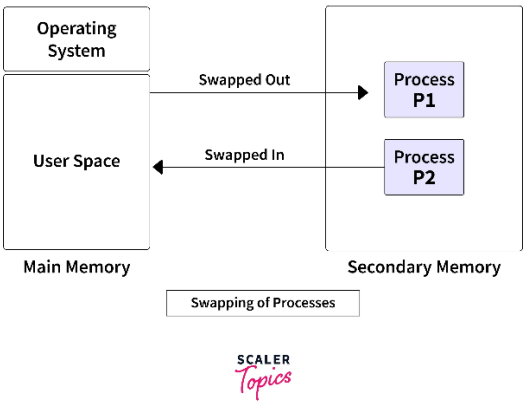
In the case of high-priority processes, the process with low priority is swapped out of the main memory and stored in swap space then the process with high priority is swapped into the main memory to be executed first.

The main goals of an operating system include **Maximum utilization of the CPU**. This means that there should be a process execution every time, the **CPU** should never stay idle and there should not be any **Process starvation** or **blocking**.

Different process management and memory management schemes are designed to fulfill such goals of an operating system.

Swapping in **OS** is done to get access to data present in secondary memory and transfer it to the main memory so that it can be used by the application programs.

It can affect the performance of the system but it helps in running more than one process by managing the memory. Therefore swapping in os is also known as the **memory compaction technique**.



There are two important concepts in the process of swapping which are as follows:

1. Swap In
2. Swap Out

Refer to the 'Swap In and Swap Out in OS' section for a detailed explanation.

Swap In and Swap Out in OS

Swap In:

The method of removing a process from secondary memory (**Hard Drive**) and restoring it to the main memory (**RAM**) for execution is known as the Swap In method.

Swap Out:

It is a method of bringing out a process from the main memory(**RAM**) and sending it to the secondary memory(**hard drive**) so that the processes with higher priority or more memory consumption will be executed known as the Swap Out method.

Note:- Swap In and Swap Out method is done by **Medium Term Scheduler(MTS)**.

Advantages of Swapping in OS

The advantages of the swapping method are listed as follows:

- Swapping in **OS** helps in achieving the goal of **Maximum CPU Utilization**.
- Swapping ensures proper memory availability for every process that needs to be executed.
- Swapping helps avoid the problem of process starvation means a process should not take much time for execution so that the next process should be executed.
- **CPU** can perform various tasks simultaneously with the help of swapping so that processes do not have to wait much longer before execution.
- Swapping ensures proper **RAM**(main memory) utilization.
- Swapping creates a dedicated disk partition in the **hard drive** for swapped processes which is called **swap space**.
- Swapping in **OS**is an economical process.
- Swapping method can be applied on **priority-based** process scheduling where a **high priority** process is swapped in and a **low priority** process is swapped out which improves the performance.

Disadvantages of Swapping in OS

There are some limited disadvantages of the swapping method which are listed as follows:

- If the system deals with **power-cut** during bulky swapping activity then the user may lose all the information which is related to the program.

- If the swapping method uses an algorithm that is not up to the mark then the number of page faults can be increased and therefore this decreases the complete performance.
- There may be inefficiency in a case when there is some common resource used by the processes that are participating in the swapping process.

Example:

Let's understand the concept of **swapping** with an example:

Suppose we have a user whose process size is 4096KB. Here the user is having a standard hard disk drive in which the swapping has a transfer rate of 4Mbps. Now we will compute how long it takes to transfer the data from the main memory which is **RAM** to the secondary memory which is the **hard disk**.

```
process size of the user is 4096Kb
the transfer rate of data is 4Mbps
Now,
4 Mbps is equal to 4096 kbps
Time taken to transfer the data = process size of user/ transfer rate of data
Now, coming to the Calculation part:
Time Taken = 4096 / 4096
        = 1 second
        = 1000 milliseconds
```

Now, taking both swap in and swap out time into account the total time taken for transferring the data from **main memory** to **secondary memory** is equal to 2000 milliseconds which is 2 Seconds.

G. Write short note - Demand paging

What is Demand Paging?

Demand paging can be described as a memory management technique that is used in operating systems to improve memory usage and system performance. Demand paging is a technique used in virtual memory

systems where pages enter main memory only when requested or needed by the CPU.

In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start. A page fault occurred when the program needed to access a page that is not currently in memory. The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.

What is Page Fault?

The term “page miss” or “page fault” refers to a situation where a referenced page is not found in the main memory.

When a program tries to access a page, or fixed-size block of memory, that isn’t currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.

In modern operating systems, page faults are a common component of virtual memory management. By enabling programs to operate with more data than can fit in physical memory at once, they enable the efficient use of physical memory. The operating system is responsible for coordinating the transfer of data between physical memory and secondary storage as needed.

What is Thrashing?

Thrashing is the term used to describe a state in which excessive paging activity takes place in computer systems, especially in operating systems that use virtual memory, severely impairing system performance. Thrashing occurs when a system’s high memory demand and low physical memory capacity cause it to spend a large amount of time rotating pages between main memory (RAM) and secondary storage, which is typically a hard disc.

It is caused due to insufficient physical memory, overloading and poor memory management. The operating system may use a variety of techniques to lessen thrashing, including lowering the number of running processes, adjusting paging parameters, and improving

memory allocation algorithms. Increasing the system's physical memory (RAM) capacity can also lessen thrashing by lowering the frequency of page swaps between RAM and the disc.

Pure Demand Paging

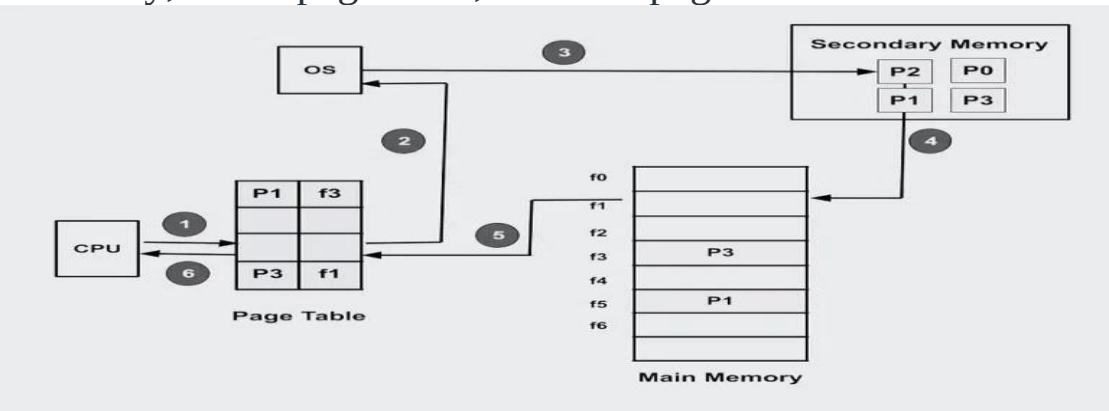
Pure demand paging is a specific implementation of demand paging. The operating system only loads pages into memory when the program needs them. In on-demand paging only, no pages are initially loaded into memory when the program starts, and all pages are initially marked as being on disk.

Operating systems that use pure demand paging as a memory management strategy do so without preloading any pages into physical memory prior to the commencement of a task. Demand paging loads a process's whole address space into memory one step at a time, bringing just the parts of the process that are actively being used into memory from disc as needed.

It is useful for executing huge programs that might not fit totally in memory or for computers with limited physical memory. If the program accesses a lot of pages that are not in memory right now, it could also result in a rise in page faults and possible performance overhead. Operating systems frequently use caching techniques and improve page replacement algorithms to lessen the negative effects of page faults on system performance as a whole.

Working Process of Demand Paging

So, let us understand this with the help of an example. Suppose we want to run a process P which have four pages P0, P1, P2, and P3. Currently, in the page table, we have pages P1 and P3.



Therefore, the operating system's demand paging mechanism follows a few steps in its operation.

- **Program Execution:** Upon launching a program, the operating system allocates a certain amount of memory to the program and establishes a process for it.
- **Creating page tables:** To keep track of which program pages are currently in memory and which are on disk, the operating system makes page tables for each process.
- **Handling Page Fault:** When a program tries to access a page that isn't in memory at the moment, a page fault happens. In order to determine whether the necessary page is on disk, the operating system pauses the application and consults the page tables.
- **Page Fetch:** The operating system loads the necessary page into memory by retrieving it from the disk if it is there.
- The page's new location in memory is then reflected in the page table.
- **Resuming the program:** The operating system picks up where it left off when the necessary pages are loaded into memory.
- **Page replacement:** If there is not enough free memory to hold all the pages a program needs, the operating system may need to replace one or more pages currently in memory with pages currently in memory. on the disk. The page replacement algorithm used by the operating system determines which pages are selected for replacement.
- **Page cleanup:** When a process terminates, the operating system frees the memory allocated to the process and cleans up the corresponding entries in the page tables.

Advantages of Demand Paging

So in the Demand Paging technique, there are some benefits that provide efficiency of the operating system.

- **Efficient use of physical memory**: Query paging allows for more efficient use because only the necessary pages are loaded into memory at any given time.

- **Support for larger programs:** Programs can be larger than the physical memory available on the system because only the necessary pages will be loaded into memory.
- **Faster program start:** Because only part of a program is initially loaded into memory, programs can start faster than if the entire program were loaded at once.
- **Reduce memory usage:** Query paging can help reduce the amount of memory a program needs, which can improve system performance by reducing the amount of disk I/O required.

Disadvantages of Demand Paging

- **Page Fault Overload:** The process of swapping pages between memory and disk can cause a performance overhead, especially if the program frequently accesses pages that are not currently in memory.
- **Degraded performance:** If a program frequently accesses pages that are not currently in memory, the system spends a lot of time swapping out pages, which degrades performance.
- **Fragmentation:** Query paging can cause physical memory fragmentation, degrading system performance over time.
- **Complexity:** Implementing query paging in an operating system can be complex, requiring complex algorithms and data structures to manage page tables and swap space.

Design a diagram illustrating the system memory layout in Unix, including the text, data, stack segment and any other relevant components.

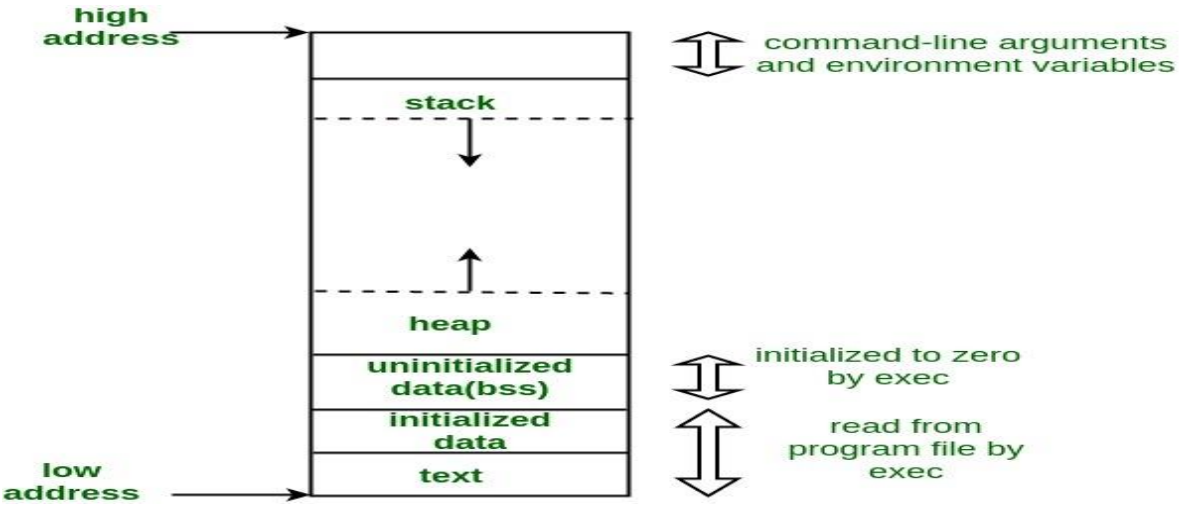
Memory Layout of C Programs

Last Updated : 24 Aug, 2022

A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)

4. Heap
5. Stack



A typical memory layout of a running process

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 the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
Note that, the 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 the initialized read-only area and the 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 the 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 the initialized read-only area and the character pointer variable string in the initialized read-write area.
Ex: static int i = 10 will be stored in the 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 in 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 in 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 in 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 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.

I. What is the use of signal? Explain the types of signals

A signal is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process.
There are fix set of signals that can be sent to a process. signal are identified by integers.
Signal number have symbolic names. For example **SIGCHLD** is number of the signal sent to the parent process when child terminates.
Examples:

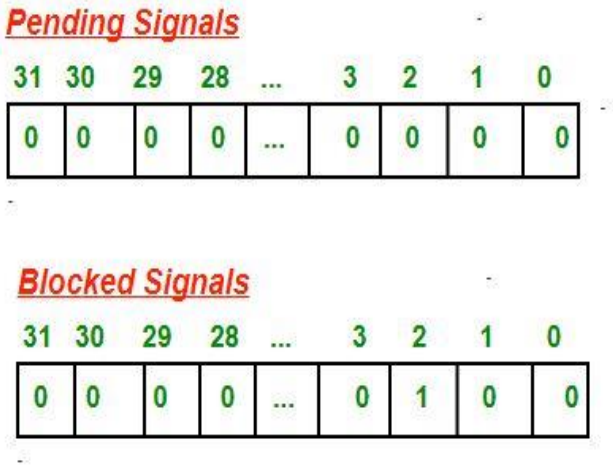
```
#define SIGHUP  1   /* Hangup the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */
```

OS Structures for Signals

- For each process, the operating system maintains 2 integers with the bits corresponding to a signal numbers.
- The two integers keep track of: **pending signals and blocked signals**
- With 32 bit integers, up to 32 different signals can be represented.

Example :
In the example below, the SIGINT (= 2) signal is blocked and no

signals are pending.



A signal is sent to a process setting the corresponding bit in the pending signals integer for the process. Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked. If no signals are pending, the process is restarted normally and continues executing at its next instruction.

If 1 or more signals are pending, but each one is blocked, the process is also restarted normally but with the signals still marked as pending. If 1 or more signals are pending and NOT blocked, the OS executes the routines in the process's code to handle the signals.

Default Signal Handlers

There are several default signal handler routines. Each signal is associated with one of these default handler routine. The different default handler routines typically have one of the following actions:

- Ign: Ignore the signal; i.e., do nothing, just return
- Term: terminate the process
- Cont: unblock a stopped process
- Stop: block the process

```
// CPP program to illustrate
// default Signal Handler
#include<stdio.h>
#include<signal.h>

int main()
```

```
{
   signal(SIGINT, handle_sigint);
   while (1)
   {
      printf("hello world\n");
      sleep(1);
   }
   return 0;
}
```

Output: Print hello world infinite times. If user presses ctrl-c to terminate the process because of **SIGINT** signal sent and its default handler to terminate the process.

```
hello world
hello world
hello world
terminated
```

User Defined Signal Handlers

A process can replace the default signal handler for almost all signals (but not SIGKILL) by its user's own handler function. A signal handler function can have any name, but must have return type void and have one int parameter. **Example:** you might choose the name sigchld_handler for a signal handler for the **SIGCHLD** signal (termination of a child process). Then the declaration would be:

```
void sigchld_handler(int sig);
```

When a signal handler executes, the parameter passed to it is the number of the signal. A programmer can use the same signal handler function to handle several signals. In this case the handler would need to check the parameter to see which signal was sent. On the other hand, if a signal handler function only handles one signal, it isn't necessary to bother examining the parameter since it will always be that signal number.