PROCESS CONCEPT:-

Process ,Process structure:

- Every process has a unique process ID, a non-negative integer. Unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
- Process ID 0 is usually the scheduler process and is often known as the swapper.
- Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure. The init process never dies.
- process ID 2 is the page daemon.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

PROCESS IDENTIFICATION:-

```
#include <unistd.h>
pid_t getpid (void);
```

Returns: process ID of calling process.

```
pid_t getppid (void);
```

Returns: parent process ID of calling process.

```
uid_t getuid (void);
```

Returns: real user ID of calling process.

```
uid_t geteuid (void);
```

Returns: effective user ID of calling process

```
gid_t getgid (void);
```

Returns: real group ID of calling process.

```
gid_t getegid (void);
```

Returns: effective group ID of calling process.

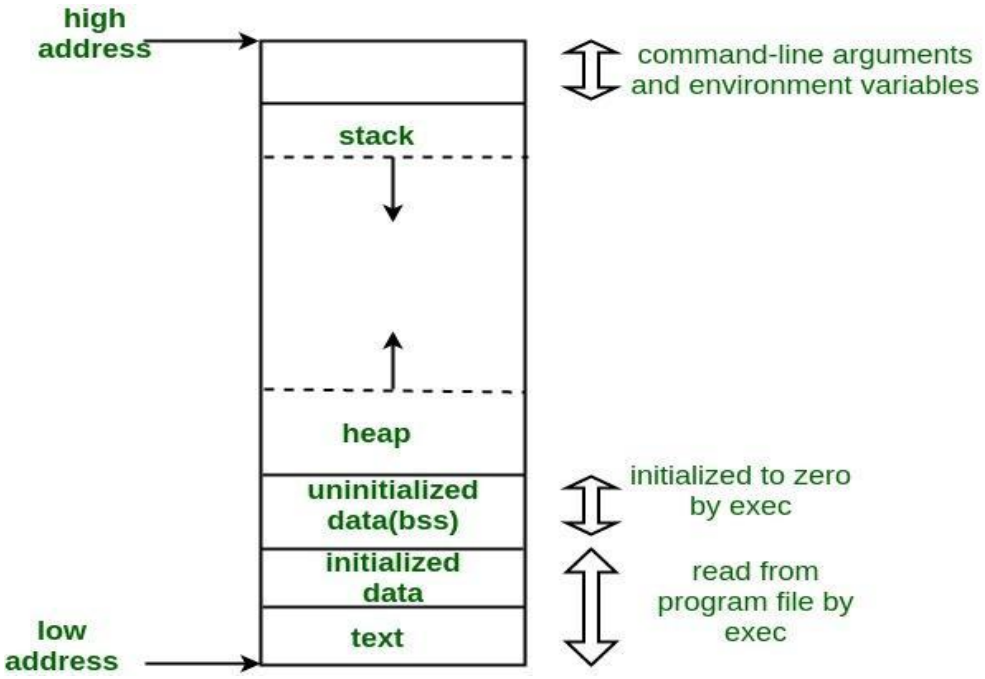
- None of these functions has an error return.

FOCUS ON STACK AND HEAP IN MEMORY LAYOUT

Memory Layout of C Programs

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

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



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 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 inti = 10 will be stored in data segment and global inti = 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 inti; 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.

Process Creation:-

1.fork()

2.vfork()

3.exec()

fork Function:

- An existing process can create a new one by calling the fork function.

#include <unistd.h>

pid_t fork(void);

Returns: 0 in child, process ID of child in parent, -1 on error.

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

Fork Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        pid_t pid;      /* could be int */
    int i;
    pid = fork();
    if( pid > 0 )
            {
            /* parent */
    for( i=0; i < 1000; i++ )
        printf("\t\t\tPARENT %d\n", i);
        }
 else
        {
        /* child */
  for( i=0; I < 1000; i++ )
            printf( "CHILD %d\n", i );
    }
```

```
  return 0;
}
```

vfork :

- The function vfork has the same calling sequence and same return values as fork.
- The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space.
- vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        pid_t pid;      /* could be int */
    int i;
    pid = vfork();
    if( pid > 0 )
            {
            /* parent */
    for( i=0; i < 1000; i++ )
        printf("\t\t\tPARENT %d\n", i);
        }
  else
        {
        /* child */
   for( i=0; I < 1000; i++ )
            printf( "CHILD %d\n", i );
    }
  return 0;
}
```

Exec

- Family of functions for replacing process's program with the one inside the exec() call. e.g.

```
#include <unistd.h>

int execlp(char *file, char *arg0,
           char *arg1, ..., (char *)0);

execlp("sort", "sort", "-n",
                "foobar", (char *)0);
```

- Exec Family: There are 6 versions of the exec function, and they all do about the same thing: they replace the current program with the text of the new program. Main difference is how parameters are passed.
- **int execl(const char *path, const char *arg, ...);**
- **int execlp(const char *file, const char *arg, ...);**
- **int execle(const char *path, const char *arg , ..., char *const envp[]);**
- **int execv(const char *path, char *const argv[]);**
- **int execvp(const char *file, char *const argv[]);**
- **int execve(const char *filename, char *const argv [], char *const envp[]);**

DIFFERENCE BETWEEN FORK AND VFORK

BASIS FOR COMPARISON	**FORK()**	**VFORK()**
Basic	Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Execution	Parent and child process execute simultaneously.	Parent process remains suspended till child

		process completes its execution.
Modification	If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.
Copy-on-write	fork() uses copy-on-write as an alternative where the parent and child shares same pages until any one of them modifies the shared page.	vfork() does not use copy-on-

WAITING FOR A PROCESS:

WAIT and WAITPID:

wait, waitpid - wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int **status*);

pid_twaitpid(pid_t*pid*, int **status*, int*options*);

intwaitid(idtype_t*idtype*, id_t*id*, siginfo_t ****infop , **int options);

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the current process until one of its children terminates. The call *wait(&status)*is equivalent to:

waitpid(-1, &status, 0);

the **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behaviour is modifiable via the *options*argument, as described below.

The value of *pid* can be:

Tag	**Description**
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.

> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

Tag	Description
WNOHANG	return immediately if no child has exited.
WUNTRACED	also return if a child has stopped(but not traced via **ptrace**(2)). Status for *traced*children which have stopped is provided even if this option is not specified.
WCONTINUED	(Since Linux 2.6.10) also return if a stopped child has been resumed by delivery of**SIGCONT**.

Process Termination:

8 ways for a process to terminate. Normal termination occurs in 5 ways:

1.Return from main

2.Calling exit

3.Calling _exit or _Exit

4.Return of the last thread from its start routine

5.Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

1.Calling abort

2.Receipt of a signal

3.Response of the last thread to a cancellation request

Exit Functions

Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);

kill Function:

The kill function sends a signal to a process or a group of processes.

#include <signal.h>

int kill(pid_t pid, int signo);

KERNEL SUPPORT FOR PROCESS:ALL THE SIGNALS AND SYSTEM CALLS MUST BE WRITTEN ALONG WITH SYNTAX AND EXPLANATION

Top of Form

Process kinds:

Zombie and Orphan Processes in C

Prerequisite: fork() in C

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.
Example for ZOMBIE process

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<signal.h>

#include<stdlib.h>

int main()

{

int pid;

pid=fork();

```
if(pid==0)
{
  printf("\n child:My work done");
  exit(0);
}
else
{
 printf("\n parent: executing");
sleep(10);
}
}
```

Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.

However, the orphan process is soon adopted by init process, once its parent process dies

Example for ORPHAN process

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
/*   create orhan  processes   */int main()
{
 int pid;
  pid = fork(); /* create child process */
  if (pid == 0) /* child gets here*/
    {
```

```
    while(1);
    }
  else /* parent gets here */
    {
    exit(0);
    }
}
```

SIGNALS SENT THROUGH PPT.

Signals:

- Signals are software interrupts. Signals provide a way of handling asynchronous events.
- Every signal has a name. These names all begin with the three characters SIG. These names are all defined by positive integer constants (the signal number) in the header <signal.h>.
- No signal has a signal number of 0.

Sources for Generating Signals

* Hardware

  - A process attempts to access addresses outside its own address space.

  - Divides by zero.

* Kernel

  - Notifying the process that an I/O device for which it has been waiting is available.

* Other Processes

  - A child process notifying its parent process that it has terminated.

* User

  - Pressing keyboard sequences that generate a quit, interrupt or stop signal.

Three Courses of Action

Process that receives a signal can take one of three action:

* Perform the system-specified default for the signal

  - notify the parent process that it is terminating;

  - generate a core file;

    (a file containing the current memory image of the process)

  - terminate.

* Ignore the signal

  A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.

* Catch the Signal

  When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine

Signal Function:

- The simplest interface to the signal features of the UNIX System is the signal function.

include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);

Returns: previous disposition of signal if OK, SIG_ERR on error.

- The signo argument is just the name of the signal. The value of func is (a) the constant SIG_IGN, (b) the constant SIG_DFL, or (c) the address of a function to be called when the signal occurs.

Simple program to catch SIGUSR1 & SIGUSR2

```
static void sig_usr (int);
 int main(void)
 {
if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    err_sys("can't catch SIGUSR1");
if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    err_sys("can't catch SIGUSR2");
 for ( ; ; )
pause();
 }
```

```
static void sig_usr (int signo)
{
 if (signo == SIGUSR1)
printf("received SIGUSR1\n");
else if (signo == SIGUSR2)
printf("received SIGUSR2\n");
else err_dump("received signal %d\n", signo);
}
```

Interrupted System Calls:

- It is a system call within the kernel that is interrupted when a signal is caught.
- To support this feature, the system calls are divided into two categories: the "slow" system calls and all the others.
- The slow system calls are those that can block forever.

- Reads that can block the caller forever if data isn't present with certain file types.
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types.
- Opens that block until some condition occurs on certain file types.
- The pause function and the wait function.
- Certain ioctl operations.
- Some of the interprocess communication functions .

kill and raise Functions:

- The kill function sends a signal to a process or a group of processes.
- The raise function allows a process to send a signal to itself.

```
#include <signal.h>

int kill( pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK,-1 on error.

There are four different conditions for the pid argument to kill.

pid > 0 The signal is sent to the process whose process ID is pid.

pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.

pid == 1 The signal is sent to all processes on the system for which the sender has permission to send the signal.

- The super user can send a signal to any process.

alarm function:

- The alarm function allows us to set a timer that will expire at a specified time in the future.
- When the timer expires, the SIGALRM signal is generated.

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

Returns: 0 or number of seconds until previously set alarm.

The seconds value is no.of clock seconds in the future when the signal should be generated.

- If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function.
- If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

pause function:

- The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);
```

Returns: 1 with errno set to EINTR

- The only time pause returns is if a signal handler is executed and that handler returns.

abort Function:

- The abort function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);
```

This function never returns.

- This function sends the SIGABRT signal to the caller.

system Function:

- It is convenient to execute a command string from within a program.

```
#include <stdlib.h>

int system(const char *cmdstring);
```

- If cmdstring is a null pointer, *system* returns nonzero only if a command processor is available. This feature determines whether the *system* function is supported on a given operating system. Under the UNIX System, *system* is always available.

The *system* is implemented by calling fork, exec, and waitpid, there are three types of return values.

- If either the fork fails or waitpid returns an error ,system returns 1 with errno set to indicate the error.
- If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
- Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from *system* is the termination status of the shell, in the format specified for waitpid.

sleep Function:

#include <unistd.h>

unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds.

This function causes the calling process to be suspended until either

- The amount of wall clock time specified by seconds has elapsed. The return value is 0.
- A signal is caught by the process and the signal handler returns. Return value is the number of unslept seconds.

POSIX and SYSTEM V-Defined Signals (1)

* SIGALRM: Alarm timer time-out. Generated by ***alarm( )*** API.
* SIGABRT: Abort process execution. Generated by ***abort( )*** API.
* SIGFPE: Illegal mathematical operation.
* SIGHUP: Controlling terminal hang-up.
* SIGILL: Execution of an illegal machine instruction.
* SIGINT: Process interruption. Can be generated by *<Delete> or <ctrl_C>* keys.
* SIGKILL: Sure kill a process. Can be generated by

  "***kill*** *-9 <process_id>*" command.
* SIGPIPE: Illegal write to a pipe.
* SIGQUIT: Process quit. Generated by *<crtl_\>* keys.
* SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.

POSIX and SYSTEM V-Defined Signals (2)

* SIGTERM: process termination. Can be generated by
  *"**kill** <process_id>"* command.
* SIGUSR1: Reserved to be defined by user.
* SIGUSR2: Reserved to be defined by user.
* SIGCHLD: Sent to a parent process when its child process has terminated.
* SIGCONT: Resume execution of a stopped process.
* SIGSTOP: Stop a process execution.
* SIGTTIN: Stop a background process when it tries to read from from its controlling terminal.
* SIGTSTP: Stop a process execution by the control_Z keys.
* SIGTTOUT: Stop a background process when it tries to write to its controlling terminal.

Unreliable Signals

- Def: Unreliable Signals. Earlier versions of Unix.
  - Signals could get lost without notice by process
- Why?
  - The action for a signal was reset to its default each time the signal occurred.
  - The process could only ignore signals, instead of turning off the signals.
  - Example of "old" programming style for signal handling:

```
int sig_int();   /*  signal handler */
….
signal(SIGINT, sig_INT);
….
sig_int()
{       ….
        signal(SIGINT, sig_INT); /* re-establish handler */
        ….
}
```

Question: What happens if 2nd signal occurs after 1st signal but before re-establishing signal handler? □ default action !

Reliable Signals

- A signal is *generated* when
  - the event that causes the signal occurs: *divide by 0, alarm*…
  - a flag is set in the process table by kernel.
- A signal is *delivered* when
  - the action for the signal is taken.
- A signal is *pending* during the time between its delivery and generation.

Reliable Signals

- A process has option to *block* a signal. A signal is *blocked* (if the action is either default or a with handler) until
  - the process unblocks the signal, or
  - the corresponding action become "ignore".
- System determines what to do with a blocked signal when signal is delivered (not when it is generated). Allows process to change action for signal before it is delivered.
  - A process may know which signals are blocked and pending using the function sigpending()

Reliable Signals

Questions: what happens if a blocked signal is generated more that once before process unblocks signal? What happens if more that one signal is ready to be delivered to a process?

- A signal mask defined for each process. Defines which signals are blocked. sigpromask() allows to examine and modify process mask.

- Signals are queued when
  - a blocked signal is generated more than once.
  - POSIX allows signal queuing, but many Unix systems do not implement this feature.

- Delivery order of signals
  - No order under POSIX.1, but its rationale states that signals related to the current process state, e.g., SIGSEGV, should be delivered first.