What is a process? A process is a program in execution.

What is a program? A program is a file containing the information of a process and how to build it during run time. When you start execution of the program, it is loaded into RAM and starts executing.

Each process is identified with a unique positive integer called as process ID or simply PID (Process Identification number). The kernel usually limits the process ID to 32767, which is configurable. When the process ID reaches this limit, it is reset again, which is after the system processes range. The unused process IDs from that counter are then assigned to newly created processes.

The system call getpid() returns the process ID of the calling process.

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

**#include <unistd.h>**

**pid\_t getpid(void);**

This call returns the process ID of the calling process which is guaranteed to be unique. This call is always successful and thus no return value to indicate an error.

Each process has its unique ID called process ID that is fine but who created it? How to get information about its creator? Creator process is called the parent process. Parent ID or PPID can be obtained through getppid() call.

The system call getppid() returns the Parent PID of the calling process.

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

**#include <unistd.h>**

**pid\_t getppid(void);**

This call returns the parent process ID of the calling process. This call is always successful and thus no return value to indicate an error.Following is a program to know the PID and PPID of the calling process.

**#include <stdio.h>**

**#include <stdlib.h>**

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

**#include <unistd.h>**

**int main() {**

**int mypid, myppid;**

**printf("Program to know PID and PPID's information\n");**

**mypid = getpid();**

**myppid = getppid();**

**printf("My process ID is %d\n", mypid);**

**printf("My parent process ID is %d\n", myppid);**

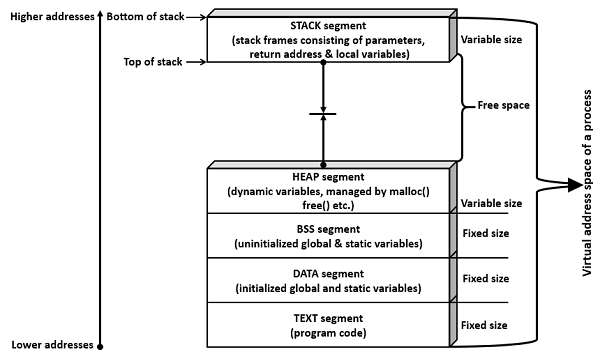
**return 0;**

**}**

**Process Image :** Process image is an executable file required while executing the program. This image usually contains the following sections −

* Code segment or text segment
* Data segment
* Stack segment
* Heap segment

Following is the pictorial representation of the process image.



**Code segment** is a portion of object file or program’s virtual address space that consists of executable instructions. This is usually read-only data segment and has a fixed size.

Data segment is of two types.

* Initialized
* Un-initialized

**Initialized data segment** is a portion of the object file or program’s virtual address space that consists of initialized static and global variables.

**Un-initialized data segment** is a portion of the object file or program’s virtual address space that consists of uninitialized static and global variables. Un-initialized data segment is also called BSS (Block Started by Symbol) segment.

**Data segment** is read-write, since the values of variables could be changed during run time. This segment also has a fixed size.

**Stack segment** is an area of memory allotted for automatic variables and function parameters. It also stores a return address while executing function calls. Stack uses LIFO (Last-In-First-Out) mechanism for storing local or automatic variables, function parameters and storing next address or return address. The return address refers to the address to return after completion of function execution. This segment size is variable as per local variables, function parameters, and function calls. This segment grows from a higher address to a lower address.

**Heap segment** is area of memory allotted for dynamic memory storage such as for malloc() and calloc() calls. This segment size is also variable as per user allocation. This segment grows from a lower address to a higher address.

Let us now check how the segments (data and bss segments) size varies.

**#include<stdio.h>**

**int main() {**

**printf("Hello World\n");**

**return 0;**

**}**

**#include<stdio.h>**

**int main() {**

**static int a;**

**printf("Hello World\n");**

**return 0;**

**}**

In the following program, an initialized static variable is added. This means initialized segment (DATA) size would increase by 4 Bytes.

**#include<stdio.h>**

**int main() {**

**static int a;**

**static int b= 100;**

**printf("Hello World\n");**

**return 0;**

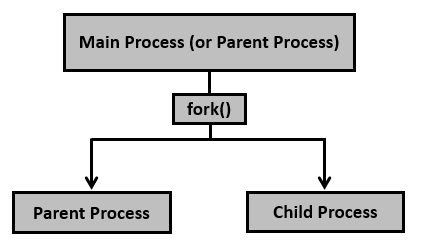
**}**

In the following program, an initialized global variable is added. This means initialized segment (DATA) size would increase by 4 Bytes.

Compile and run all three programs and check the size of each using ***size*** command.

Process: Parent process, Child process

Process creation is achieved through the **fork() system call**. The newly created process is called the child process and the process that initiated it (or the process when execution is started) is called the parent process. After the fork() system call, now we have two processes - parent and child processes. How to differentiate them? Very simple, it is through their return values.



Example

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

**#include <unistd.h>**

**pid\_t fork(void);**

Creates the child process. After this call, there are two processes, the existing one is called the parent process and the newly created one is called the child process.

The fork() system call returns either of the three values −

* Negative value to indicate an error, i.e., unsuccessful in creating the child process.
* Returns a zero for child process.
* Returns a positive value for the parent process. This value is the process ID of the newly created child process.

**#include <stdio.h>**

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

**#include <unistd.h>**

**int main() {**

**fork();**

**printf("Called fork() system call\n");**

**return 0;**

**}**

In the above case, fork() is called once, hence the output is printed twice (2 power 1). If fork() is called, say 3 times, then the output would be printed 8 times (2 power 3). If it is called 5 times, then it prints 32 times and so on and so forth.

**#include <stdio.h>**

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

**#include <unistd.h>**

**int main() {**

**pid\_t pid, mypid, myppid;**

**pid = getpid();**

**printf("Before fork: Process id is %d\n", pid);**

**pid = fork();**

**if (pid < 0) {**

**perror("fork() failure\n");**

**return 1;**

**}**

**// Child process**

**if (pid == 0) {**

**printf("This is child process\n");**

**mypid = getpid();**

**myppid = getppid();**

**printf("Process id is %d and PPID is %d\n", mypid, myppid);**

**} else { // Parent process**

**sleep(2);**

**printf("This is parent process\n");**

**mypid = getpid();**

**myppid = getppid();**

**printf("Process id is %d and PPID is %d\n", mypid, myppid);**

**printf("Newly created process id or child pid is %d\n", pid);**

**}**

**return 0;**

**}**

Execute the above code and observe the output.

A process can terminate in either of the two ways −

* Abnormally, occurs on delivery of certain signals, say terminate signal.
* Normally, using \_exit() system call (or \_Exit() system call) or exit() library function.

The difference between \_exit() and exit() is mainly the cleanup activity. The **exit()** does some cleanup before returning the control back to the kernel, while the **\_exit()** (or \_Exit()) would return the control back to the kernel immediately.

**#include <stdio.h>**

**#include <stdlib.h>**

**void exitfunc() {**

**printf("Called cleanup function - exitfunc()\n");**

**return;**

**}**

**int main() {**

**atexit(exitfunc);**

**printf("Hello, World!\n");**

**exit (0);**

**}**

**#include <stdio.h>**

**#include <unistd.h>**

**void exitfunc() {**

**printf("Called cleanup function - exitfunc()\n");**

**return;**

**}**

**int main() {**

**atexit(exitfunc);**

**printf("Hello, World!\n");**

**\_exit (0);**

**}**

Process Management:

Whenever we create a child process from a program using fork:

* Current process becomes the parent process
* The new process becomes the child process

Run and check the output

**#include<stdio.h>**

**int main() {**

**int pid;**

**pid = fork();**

**// Child process**

**if (pid == 0) {**

**system("ps -ef");**

**sleep(10);**

**system("ps -ef");**

**} else {**

**sleep(3);**

**}**

**return 0;**

**}**

Variants of system calls to monitor the child process/es −

* wait()
* waitpid()
* waitid()

**#include<stdio.h>**

**int main() {**

**int pid;**

**int status;**

**pid = fork();**

**// Child process**

**if (pid == 0) {**

**system("ps -ef");**

**sleep(10);**

**system("ps -ef");**

**return 3; //exit status is 3 from child process**

**} else {**

**sleep(3);**

**wait(&status);**

**printf("In parent process: exit status from child is decimal %d, hexa %0x\n", status, status);**

**}**

**return 0;**

**}**

The wait() system call has limitation such as it can only wait until the exit of the next child. If we need to wait for a specific child it is not possible using wait(), however, it is possible using waitpid() system call. Run the following program and check the output.

**#include<stdio.h>**

**#include<unistd.h>**

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

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

**int main() {**

**int pid;**

**int pids[3];**

**int status;**

**int numprocesses = 0;**

**int total\_processes = 3;**

**while (numprocesses < total\_processes) {**

**pid = fork();**

**// Child process**

**if (pid == 0) {**

**printf("In child process: process id is %d\n", getpid());**

**sleep(5);**

**return 4;**

**} else {**

**pids[numprocesses] = pid;**

**numprocesses++;**

**printf("In parent process: created process number: %d\n", pid);**

**}**

**}**

**// Waiting for 3rd child process**

**waitpid(pids[total\_processes - 1], &status, 0);**

**if (WIFEXITED(status) != 0) {**

**printf("process %d exited normally\n", pids[total\_processes - 1]);**

**printf("exit status from child is %d\n", WEXITSTATUS(status));**

**} else {**

**printf("process %d not exited normally\n", pids[total\_processes - 1]);**

**}**

**return 0;**

**}**