tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.

This tutorial covers a foundational understanding of IPC. Each of the chapters contain related topics with simple and useful examples.

Audience

This tutorial is designed for beginners who seek to understand the basic concepts of inter process communication and how its different components function.

Prerequisites

There are no particular prerequisites for this tutorial, however, a sound knowledge of operating systems and its various concepts will be an added advantage in understanding this tutorial.

Copyright and Disclaimer

Table of Contents

1. IPC - Overview

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types:

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes.

Following are some important terms that we need to know before proceeding further on this topic.

Pipes: Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

FIFO: Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

Message Queues: Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

Shared Memory: Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

Semaphores: Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.

Signals: Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

Note: Almost all the programs in this tutorial are based on system calls under Linux Operating System (executed in Ubuntu).

2. IPC - Process Information

Before we go into process information, we need to know a few things, such as -

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.
```

Let us understand this with a simple example.

Following is a program to know the PID and PPID of the calling process.

```
File name: processinfo.c


#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);

    printf("Cross verification of pid's by executing process commands on
shell\n");
    system("ps -ef");
    return 0;
}
```

On compilation and execution of the above program, following will be the output.

```
sh-4.3$ gcc -o processinfo processinfo.c
sh-4.3$ ./processinfo
Program to know PID and PPID's information
My process ID is 130
My parent process ID is 8
Cross verification of pid's by executing process commands on shell
UID        PID  PPID  C STIME TTY          TIME CMD
cg           1     0  0 11:20 ?        00:00:02 --CODINGGROUND--
cg           8     1  0 11:20 pts/0    00:00:00 sh
cg         130     8  0 11:56 pts/0    00:00:00 ./processinfo
cg         131   130  0 11:56 pts/0    00:00:00 ps -ef
sh-4.3$
```

Note: The "C" library function system() executes a shell command. The arguments passed to system() are commands executed on shell. In the above program, command is "ps", which gives process status.

The complete information about all running processes and other system related information are accessible from proc file system available at /proc location.

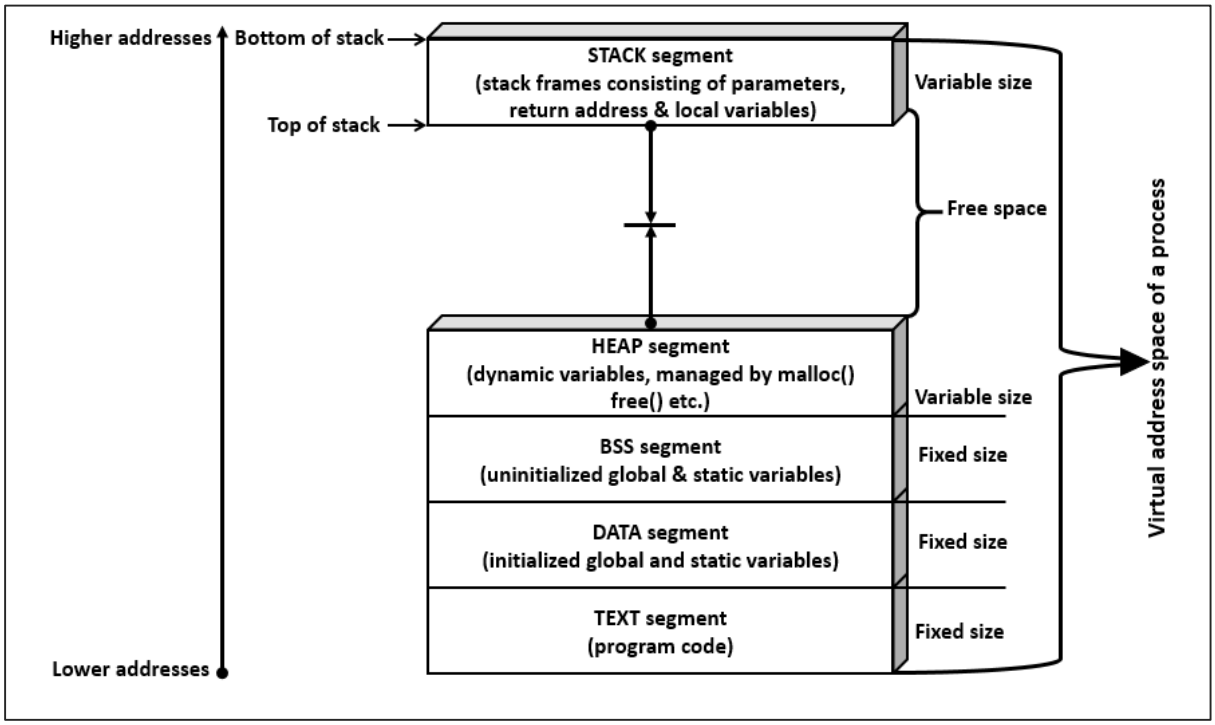
3. IPC - Process Image

Now that we have seen how to get the basic information of process and its parent process, it is time to look into the details of process/program information.

What exactly is 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 vary with a few sample programs. Segment size is known by executing the command "size".

Initial program

File: segment_size1.c

```
#include<stdio.h>
int main()
{
     printf("Hello World\n");
     return 0;
}
```

In the following program, an uninitialized static variable is added. This means uninitialized segment (BSS) size would increase by 4 Bytes. **Note**: In Linux operating system, the size of int is 4 bytes. Size of the integer data type depends on the compiler and operating system support.

File: segment_size2.c

```
#include<stdio.h>
int main()
{
     static int mystaticint1;

     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.

File: segment_size3.c

```
#include<stdio.h>
int main()
{
     static int mystaticint1;
     static int mystaticint2 = 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.

File: segment_size4.c

```
#include<stdio.h>
int myglobalint1 = 500;
int main()
{
     static int mystaticint1;
     static int mystaticint2 = 100;
```

```
     printf("Hello World\n");
     return 0;
}
```

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

File: segment_size5.c

```
#include<stdio.h>
int myglobalint1 = 500;
int myglobalint2;
int main()
{
     static int mystaticint1;
     static int mystaticint2 = 100;

     printf("Hello World\n");
     return 0;
}
```

Execution Steps

Compilation

```
babukrishnam $ gcc segment_size1.c -o segment_size1
babukrishnam $ gcc segment_size2.c -o segment_size2
babukrishnam $ gcc segment_size3.c -o segment_size3
babukrishnam $ gcc segment_size4.c -o segment_size4
babukrishnam $ gcc segment_size5.c -o segment_size5
```

Execution/Output

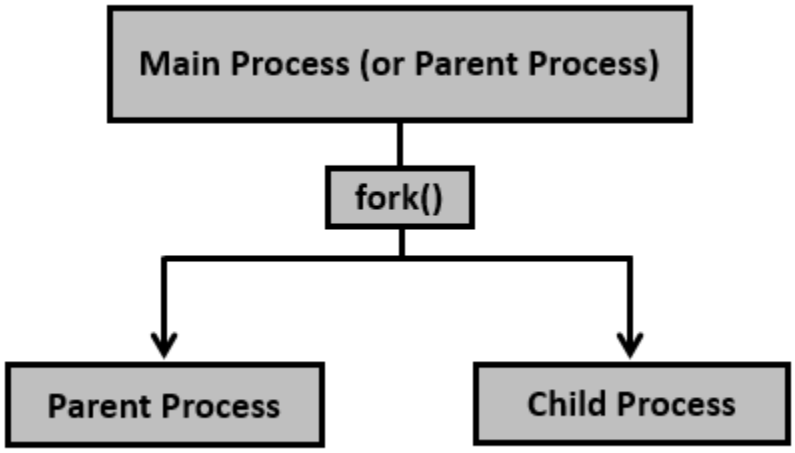
```
babukrishnam $ size segment_size1 segment_size2 segment_size3 segment_size4
segment_size5
   text    data     bss     dec     hex filename
```

```
    878      252        8    1138     472 segment_size1
    878      252       12    1142     476 segment_size2
    878      256       12    1146     47a segment_size3
    878      260       12    1150     47e segment_size4
    878      260       16    1154     482 segment_size5
babukrishnam $
```

4. IPC - Process Creation & Termination

Till now we know that whenever we execute a program then a process is created and would be terminated after the completion of the execution. What if we need to create a process within the program and may be wanted to schedule a different task for it. Can this be achieved? Yes, obviously through process creation. Of course, after the job is done it would get terminated automatically or you can terminate it as needed.

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.



After creation of the child process, let us see the fork() system call details.

```
#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.

Let us consider a simple program.

```
File name: basicfork.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Called fork() system call\n");
    return 0;
}
```

Execution Steps

Compilation

```
$ gcc basicfork.c -o basicfork
```

Execution/Output

```
$ ./basicfork
Called fork() system call
Called fork() system call
$
```

Note: Usually after fork() call, the child process and the parent process would perform different tasks. If the same task needs to be run, then for each fork() call it would run 2 power n times, where **n** is the number of times fork() is invoked.

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.

Having seen fork() create the child process, it is time to see the details of the parent and the child processes.

File name: pids_after_fork.c

```
#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);
    }
    // Parent process
    else
    {
        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;
}
```

Compilation and Execution Steps

```
sh-4.3$ gcc pids_after_fork.c -o pids_after_fork
sh-4.3$ ./pids_after_fork
Before fork: Process id is 28
This is child process
Process id is 29 and PPID is 28
This is parent process
Process id is 28 and PPID is 7
Newly created process id or child pid is 29
sh-4.3$
```

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.

Consider the following example program with exit().

File name: atexit_sample.c

```
#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);
}
```

Compilation and Execution Steps

```
sh-4.3$ gcc atexit_sample.c
sh-4.3$ ./a.out
Hello, World!
Called cleanup function - exitfunc()
sh-4.3$
```

Consider the following example program with _exit().

File name: at_exit_sample.c

```
#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);
}
```

Compilation and Execution Steps

```
sh-4.3$ gcc at_exit_sample.c
sh-4.3$ ./a.out
Hello, World!
sh-4.3$
```