PROCESS - AN INTRODUCTION

Introduction to process:

A process is an instance of a program in execution. A program by itself is not a process; a program is a ***passive entity***, such as a file containing a list of instructions stored on disks. (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Program: A set of instructions a computer can interpret and execute.

Process:

- Dynamic
- Part of a program in execution
- a live entity, it can be created, executed and terminated.
- It goes through different states
  wait
  running
  Ready etc
- Requires resources to be allocated by the OS
- one or more processes may be executing the same code.

Program:

- static
- no states

This example illustrate the difference between a process and a program:

```
main ()
 {
        int i , prod =1;
        for (i=0;i<100;i++)
        prod = pord*i;
 }
```

It is a program containing one multiplication statement (prod = prod * i) but the process will execute

100 multiplication, one at a time through the 'for' loop.
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance several users may be running different copies of mail program, or the same user may invoke many copies of web browser program. Each of these is a separate process, and although the text sections are equivalent, the data, heap and stack section may vary.

The process Model

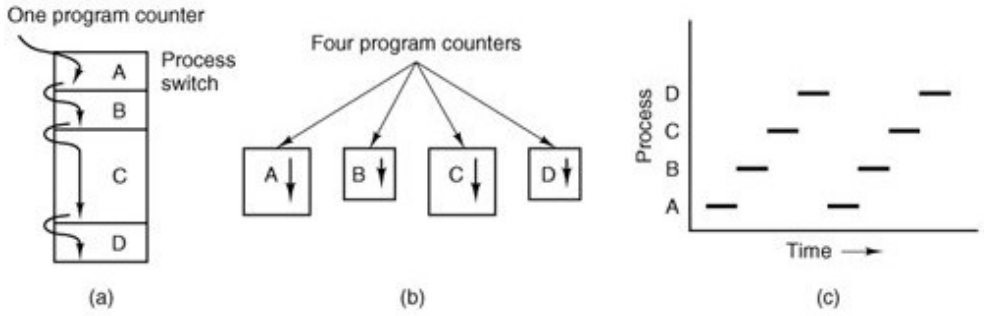

Fig:1.1: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called multiprogramming

Process Creation:

There are four principal events that cause processes to be created:
1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Parent process create children processes, which, in turn create other processes, forming a tree of processes . Generally, process identified and managed via a process identifier (pid)

When an operating system is booted, often several processes are created.

Some of these are foreground processes, that is, processes that interact with (human) users and perform

work for them.

Others are background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as web pages, printing, and so on are called daemons
In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job.

In interactive systems, users can start a program by typing a command or double clicking an icon.
In UNIX there is only one system call to create a new process: **fork.** This call creates an exact clone of the calling process. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes **execve** or a similar system call to change its memory image and run a new program. For example, when a user types a command, say, sort, to the shell, the shell forks off a child process and the child executes sort.

The C program shown below describes the system call, fork and exec used in UNIX. We now have two different process running a copy of the same program. The value of the PID for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command /bin/ls using the execlp() system call. (execlp is version of the exec). The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking the exit()) the parent process resumes from the call to wait, where it completes using the exit() system call. This is also illustrated in the fig. below.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            printf("%d", pid);
              exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
```

```
            wait (NULL);
            printf ("Child Complete\n");
            printf("%d",pid);
              exit(0);
    }
}
```

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Source : http://dayaramb.files.wordpress.com/2012/02/operating-system-pu.pdf