**What is a.out ?**

A executable program such as a.out is a file containing a range of information that describes how to construct a process at run time. This information includes the following:

* Binary format identification : Each program file includes meta information describing the format of the executable file. Most UNIX implementations (including Linux) employ the Executable and Linking Format (ELF). This enables the kernel to interpret information in the file.
* Machine-language instructions: These encode the algorithm of the program.
* Program entry-point address: This identifies the location of the instruction at which execution of the program should commence.
* Data: The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).
* Symbol and relocation tables: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
* Shared-library and dynamic-linking information: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
* Other information: The program file contains various other information that describes how to construct a process

What is a Process:

A process is an instance of an executing program. Several processes can be running the same program. While running the program, kernel appends additional information required to run the program. Information such as kernel data structures that maintain information about the state of the process. The information recorded in the kernel data structures include various identifier numbers (IDs) associated with the process, virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling ( we will talk about Signals soon), process resource usages and limits, the current working directory, and a host of other information. Each running program will have a ID and usually launched by another programs aka parent process. In our system and in all systems, the init process ( aka Kernel ) has ID = 1 and all processes will be child of the his process.

**Process ID and parent process ID**

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process.

#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 |

With the exception of a few system processes such as init (whose process ID is always 1), process ID will very different next time you run the program. The Linux kernel limits process IDs to being less than or equal to 32,767. When a new process is created, it is assigned the next sequentially available process ID. Each time the limit of 32,767 is reached, the kernel resets its process ID counter so that process IDs are assigned starting from low integer values.

**Unix Commands**

**Processes**

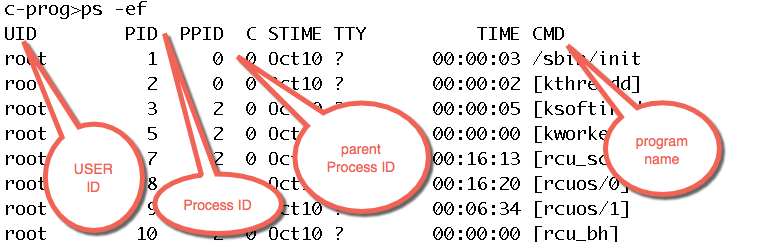
 A program is a set of instructions in passive state stored in a file.  A process executes this program in active state and is executed by a processor.   Every process in Linux gets a process ID.

The process ID of a process can be printed using the system function getpid (  )  and the parent process ID as getppid ( ) .  You can also print the process ID with the command

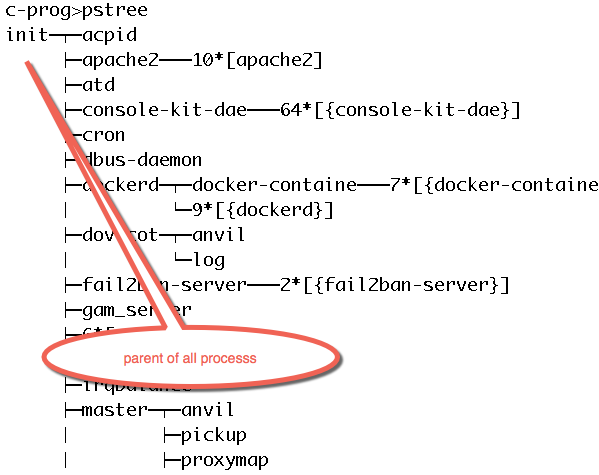
ps -ef

Every process is a child of another process , except the init process.

see the screen shot



You can view the processes in a tree structure using the command pstree command.



You can view the processes created by you,  by typing

c-prog>ps -u <USERID>

  PID TTY          TIME CMD

15113 ?        00:00:00 sshd

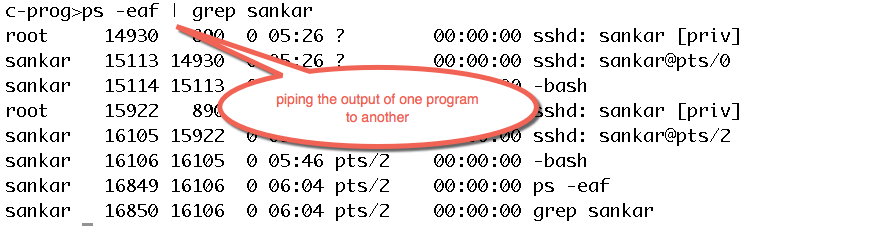
15114 pts/0    00:00:00 bash

16105 ?        00:00:00 sshd

16106 pts/2    00:00:00 bash

16913 pts/2    00:00:00 ps

or you could view in a long format  using the pipe command to ps -eaf



  When a process is executed by the CPU,  the process accesses  all registers and RAM.    Because there are several programs share CPU time,  Operating System uses a scheduling algorithm ( a popular being Round robin) which gives equal time slice to all processes and processes may have different priority to run.  Some processes may have higher priority than others.  When a process is given the CPU time, it might finish execution during the time slice it is allocated or it may be swapped out with another process that is next in the pipeline.  This is called preempted.

Try this command

top

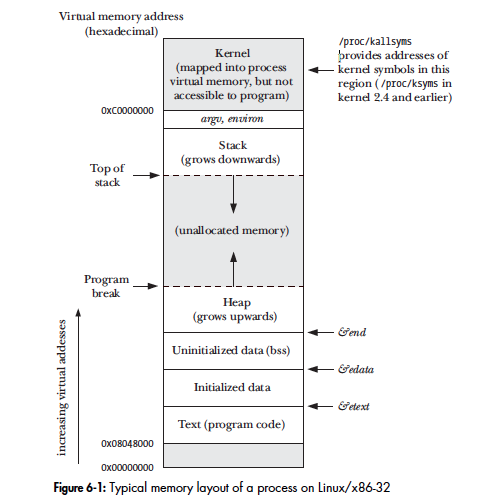
The program that is running is flagged as R and the programs in waiting/sleeping flagged as S.  You quit the top program by typing q .

**Memory Layout of a C Program**

A program may contain global variables (initialized and uninitialized) and functions. When you launch your program, it is loaded into memory . While running, our program can be swapped/switched with another, and we don’t want to lose the status of our program. To store the status of our program, our program is compartmentalized into various sections. We can get the size of each of these sections using the command

size a.out <enter>

Let us examine the sections. The various sections are given below



Text Area: The text segment contains the machine-language instructions of the program run by the process. The text segment is made read-only so that a process doesn’t accidentally modify its own instructions via a bad pointer value. Since many processes may be running the same program, the text segment is made sharable so that a single copy of the program code can be mapped into the virtual address space of all of the processes.

Stack Area: This section is meant for functions. When your program calls functions, the instructions, local variables, return address and other information is loaded in this area.

Heap Area: If your programs allocates dynamic memory, this memory is allocated in the heap area. As your program allocates more memory, all this is allocated in the heap area. Managing this area is very complicated.

Stack vs Heap: Management of this stack area is little easier than Heap area because sometime your program will deallocate the dynamic memory in heap causing gaps resulting in memory gaps. Sometimes, your program may not deallocate the dynamic memory resulting in memory leaks. The stack area grows linearly from top to bottom and management of this memory is little easier.

BSS Area:

This uninitialized data segment contains global and static variables that are not explicitly initialized. Before starting the program, the system initializes all memory in this segment to 0. For historical reasons, this is often called the bss segment, a name derived from an old assembler mnemonic for “block started by symbol.” The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialized data. Instead, the executable merely needs to record the location and size required for the uninitialized data segment, and this space is allocated by the program loader at run time

Data section : This initialized data segment contains global and static variables that are explicitly initialized. The values of these variables are read from the executable file when the program is loaded into memory.

We can more information about these segments using command

objdump –-f -h a.out

We will write several versions of a simple program adding variables in each version and check how the sections of the code varies in size. Here is the table , the left column is our program, the center column summarizes the changes we made, the right column shows the size of Text, Data and BSS sections. The command to get the size of these sections is size a.out

|  |  |  |
| --- | --- | --- |
| Summary of our work | | |
| Program : mem.c | Type of variables in the program | size a.out<enter> |
| #include <stdio.h>  int main ( ) {  } | no variables. | text data bss  1076 496 16 |
| #include <stdio.h>  int age = 20;  int main ( ) {  } | one global variable initialized. | text data bss  1076 500 16 |
| #include <stdio.h>  int age = 20;  int myAge ;  main ( )  {  } | one global variable initialize  one global variable uninitialized | text data bss  1076 500 24 |
| #include <stdio.h>  int age = 20;  int myAge;  main ( )  {  int mainAge = 30;  } | one global variable initialize  one global variable uninitialized  one local variable | text data bss  1092 500 24 |

**Global Variables : etext, edata and end**

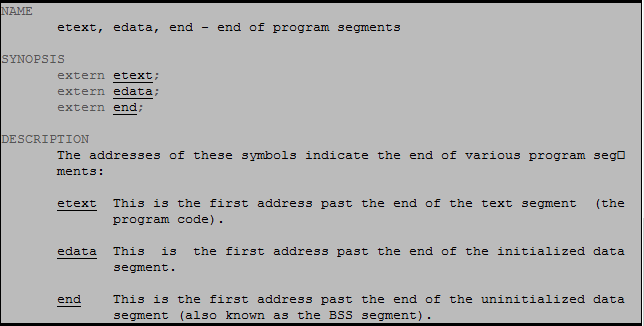
Most UNIX implementations (including Linux) provides three global symbols: etext , edata , and end . These symbols can be used from within a program to obtain the addresses of the next byte past, respectively, the end of the program text, the end of the initialized data segment, and the end of the uninitialized data segment. To make use of these symbols, we must explicitly declare them, as follows:

.

|  |  |
| --- | --- |
| #include <stdio.h>  extern etext, edata, end ;  main ( )  {  printf ("End of Text segment %10p \n", &etext);  printf ("End of Data segment or start of BSS %10p \n", &edata);  printf ("End of BSS data %10p \n", &end);  } | /\* &etext gives the address of the end  of the program text / start of initialized data \*/ |

You can get information about these variables using man command

man end



when you run this program

a.out<enter>

the output is

End of Text segment 0x400626

End of Data Segment or start of BSS 0x601020

End of BSS data 0x601030

**Virtual Memory Management**:

All the addresses we print in our programs are virtual address, ie they are not real physical addresses. Linux employs a technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM (physical memory). A virtual memory scheme splits the memory used by each program into small, fixed-size units called pages . Correspondingly, RAM is divided into a series of page frames of the same size. At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called resident set . Copies of the unused pages of a program are maintained in the swap area —a reserved area of disk space used to supplement the computer’s RAM—and loaded into physical memory only as required. When a process references a page that is not currently resident in physical memory, a page fault occurs, at which point the kernel suspends execution of the process while the page is loaded from disk into memory. On x86-32, pages are 4096 bytes in size. We can get the page size in our system using the program here

#include <stdio.h>

#include <unistd.h>

int main ( )

{

long sz = sysconf(\_SC\_PAGESIZE);

printf ( "page size = %ld \n", sz );

}

c-prog>gcc pagesize.c

c-prog>./a.out

page size = 4096

In our system, the page size is 4K.

How are virtual addresses mapped to physical addresses then ?

The kernel maintains a page table for each process (see the Figure below). The page table describes the location of each page in the process’s virtual address space (the set of all virtual memory pages available to the process). Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.

Virtual memory management separates the virtual address space of a process from the physical address space of RAM. This provides many advantages :

* Processes are isolated from one another and from the kernel, so that one process can’t read or modify the memory of another process or the kernel. This is accomplished by having the page-table entries for each process point to distinct sets of physical pages in RAM (or in the swap area).
* Where appropriate, two or more processes can share memory. The kernel makes this possible by having page-table entries in different processes refer to the same pages of RAM. This could happen when we fork a process, the child and parent may share the same pages until one of them updates the memory, then the copy of the memory is created , this is copy-on-write concept.
* The implementation of memory protection schemes is facilitated; that is, pagetable entries can be marked to indicate that the contents of the corresponding page are readable, writable, executable, or some combination of these protections.
* Where multiple processes share pages of RAM, it is possible to specify that each process has different protections on the memory; for example, one process might have read-only access to a page, while another has read-write access.
* Programmers, and tools such as the compiler and linker, don’t need to be concerned with the physical layout of the program in RAM.
* Because only a part of a program needs to reside in memory, the program loads and runs faster.
* One final advantage of virtual memory management is that since each process uses less RAM, more processes can simultaneously be held in RAM. This typically leads to better CPU utilization, since it increases the likelihood that, at any moment intime, there is at least one process that the CPU can execute.

