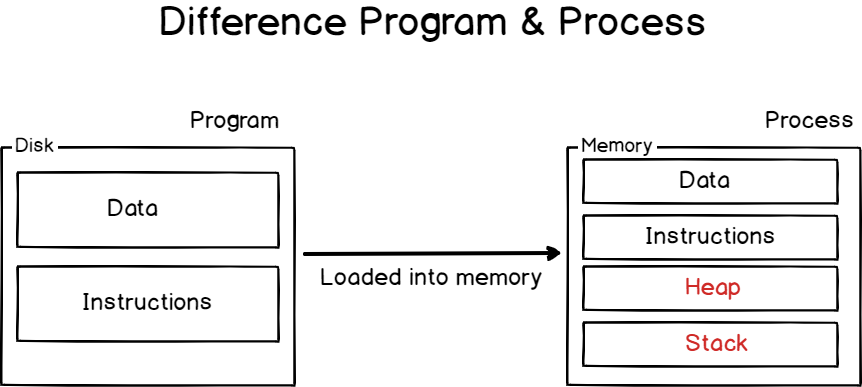
### What is Process ??

An instance of a running program is called a process. Every time you run a shell command, a program is run and a process is created for it. Each process in Linux has a process id (PID) and it is associated with a particular user and group account.

Processes are really at the center of the Linux operating system : created by the Kernel itself, they represent running operations currently happening on your Linux host.

Processes are everywhere, they may run in the background or you may choose to initialize them by yourself for custom operations.

### Program vs Process



P**rograms** are lines or code or lines of machine instructions stored on a persistent data storage. They can just sit on your data storage, or they can be in execution, i.e running as processes.

In order to perform the operations they are assigned to, processes need **resources**: **CPU time**, **memory**(such as **RAM**or **disk space**), but also virtual memory such as **swap space** in case your process gets too greedy.

### Process Initialization on Linux

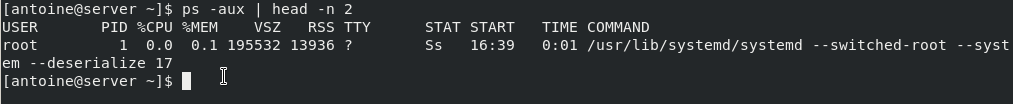
As we already stated, processes are **managed by the Kernel** on Linux.

By default, when you boot a Linux system, your Linux kernel is loaded into memory, it is given a virtual filesystem in the RAM (also called **initramfs**) and the initial commands are executed.

Historically, this process was called the [init process](https://en.wikipedia.org/wiki/Init) but it got replaced by the [systemd initialization process](https://en.wikipedia.org/wiki/Systemd) on many recent Linux distributions.

To prove it, run the following command on your host

$ ps -aux | head -n 2

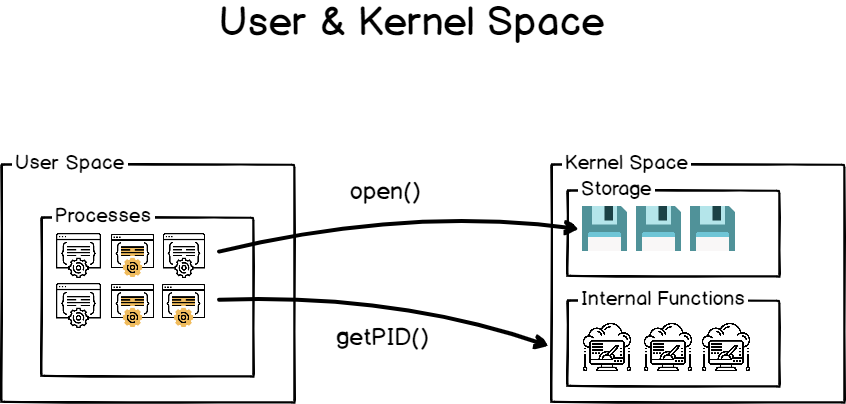


As you can see, the systemd process has**a PID of 1.**

It is noteworthy to underline the fact that all those initialization steps (except for the launch of the initial process) are done in a reserved space called **the kernel space.**

The kernel space is **a space reserved to the Kernel** in order for it to run essential system tools properly and to make sure that your entire host is running in a consistent way.

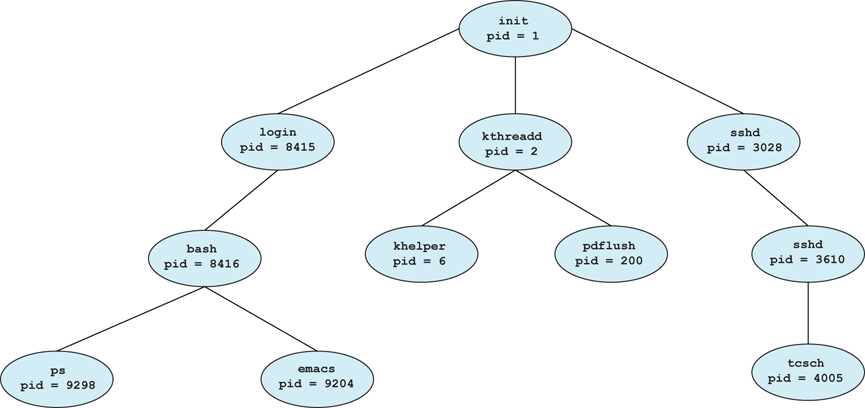
On the other hand, **user space is reserved for processes** launched by the user and managed by the kernel itself.



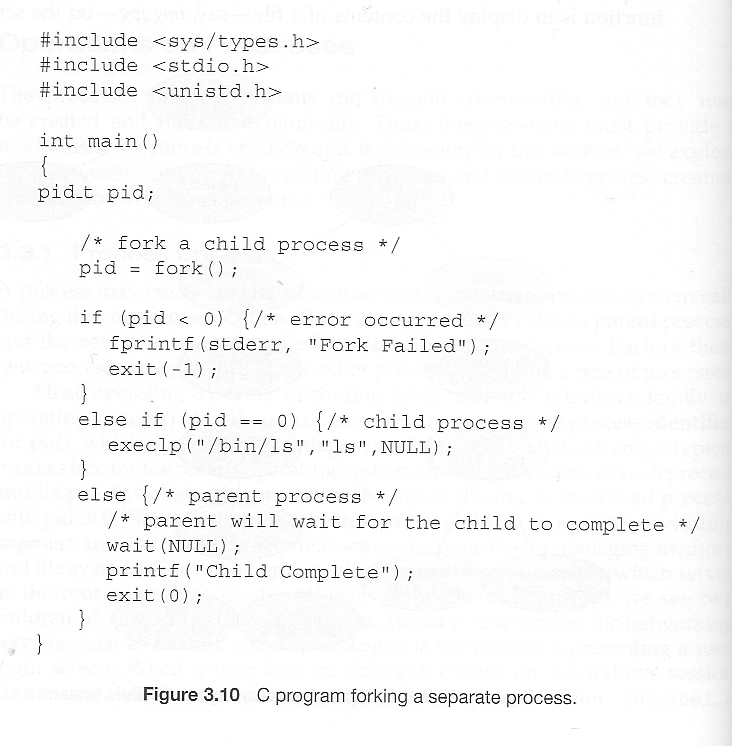
### Process Creation using Fork and Exec

**Method 1**

* Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
* Each process is given an integer identifier, termed its**process identifier**, or PID. The parent PID ( PPID ) is also stored for each process.
* On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.9 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:



* Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.
* There are two options for the parent process after creating the child:
  1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
  2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. ( E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children. )
* Two possibilities for the address space of the child relative to the parent:
  1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the **fork** system call in UNIX.
  2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the **spawn** system calls in Windows. UNIX systems implement this as a second step, using the **exec** system call.
* Figures 3.10 and 3.11 below shows the fork and exec process on a UNIX system. Note that the **fork** system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.



**Method 2**

The systemd process is the very first process launched in the user space.

When you are creating and running a program on Linux, it generally involves two main steps : **fork**and **execute**.

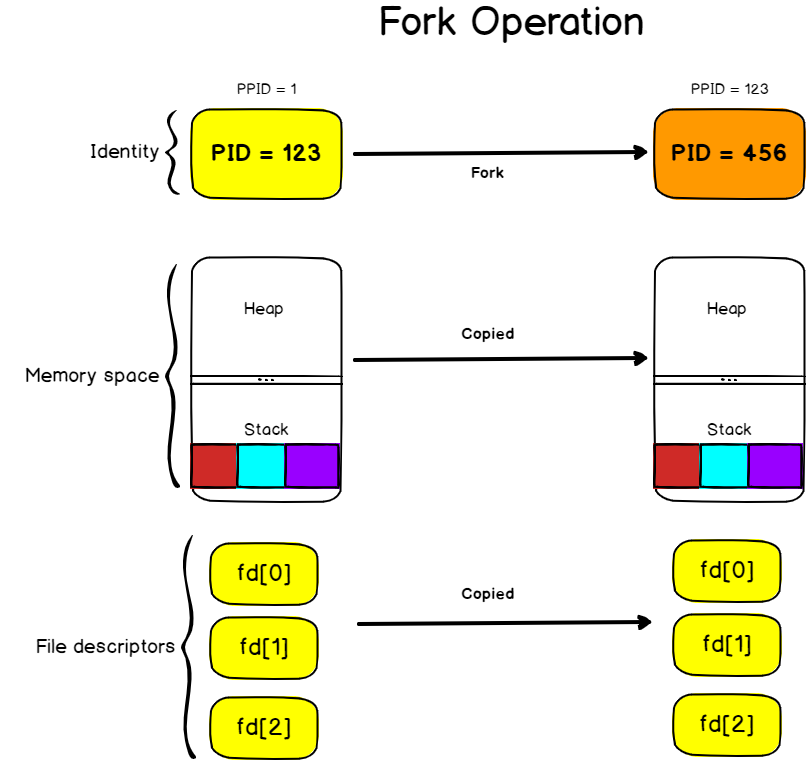
**Fork Operation**

[Fork](http://man7.org/linux/man-pages/man2/fork.2.html)is a clone operation, it takes the current process, also called the parent process, and it clones it in a new process with a brand new process ID.

When forking, everything is copied from the parent process : **the stack**, **the heap**, but also the file descriptors meaning **the standard input, the standard output and the standard error.**

It means that if my parent process was writing to the current shell console, the child process will also write to the shell console.

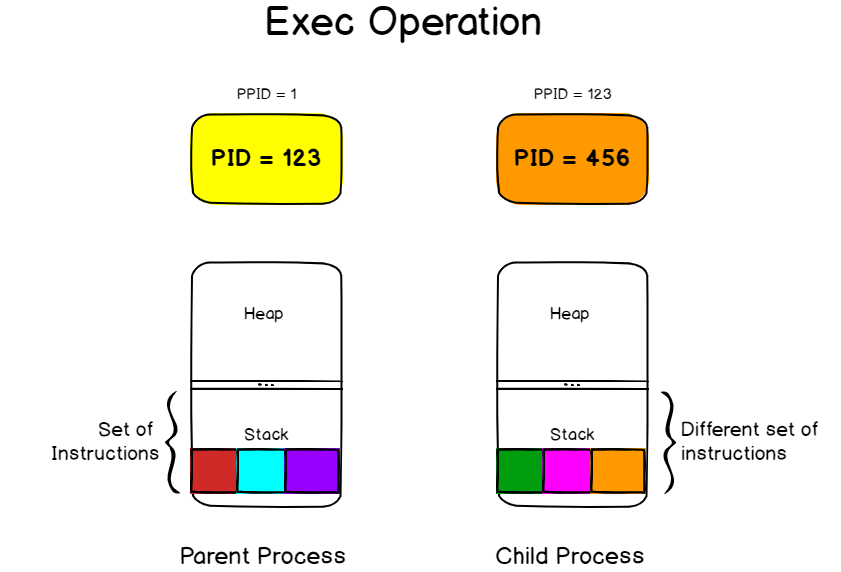
The execution of the cloned process will also start **at the same instruction as the parent process**



#### **Execute operation**

The execute operation is used on Linux **to replace the current process image with the image from another process.**

The [exec](http://man7.org/linux/man-pages/man3/exec.3.html)operation will replace the process image (i.e the set of instructions that need to be executed) by another one.



**Method 3**

Parent process can create several child process, where child can create other child process.

Resource Sharing

Parent process can shaere all or some or never to child.

Resources are CPU, file, memory.

Execution

Parent process & child process occur concurrently. or

Parent process can wait for child process completion

Address space of child process

Gets copy/duplicate of parent process.

Child can/may be replaced with new process

Unix examples

Fork system call >> creates new process

Exec system call >> Corresponding child process replaced with new process.

### Creating processes from a shell environment

When you are launching a shell console, the exact same principles apply when you are launching a command.

A shell console is a process that waits for input from the user.

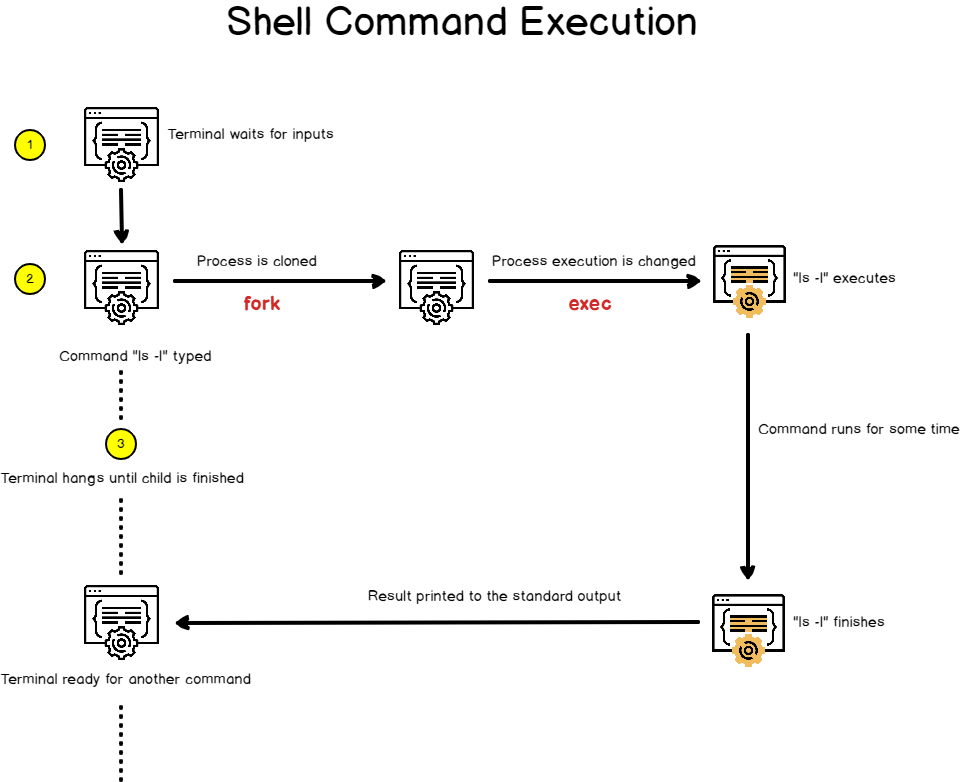
It also launches a bash interpreter when you hit Enter and it provides an environment for your commands to run.

But the shell follows the steps we described earlier.

When you hit enter,**the shell is forked to a child process** that will be responsible for running your command. The shell will wait patiently until the execution of the child process finishes. On the other hand, **the child process is linked to the same file descriptors** and it may share variables that were declared on a global scope.

The child process executes the “**exec**” command in order to replace the current process image (which is the shell process image) in the process image of the command you are trying to run.

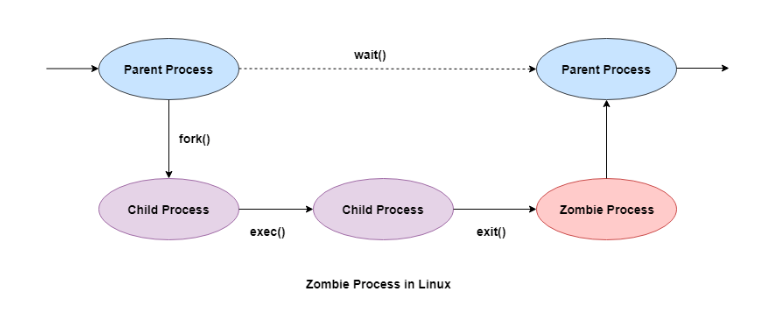
The child process will eventually finish and it will print its result to the standard output it inherited from the parent process, in this case the shell console itself.



### Zombie Process

ps -e -o pid,stat,cmd

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child’s exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.



Some of the salient points related to zombie processes are as follows −

* All the memory and resources allocated to a process are deallocated when the process terminates using the exit() system call. But the process’s entry in the process table is still available. This process is now a zombie process.
* The exit status of the zombie process zombie process can be read by the parent process using the wait() system call. After that, the zombie process is removed from the system. Then the process ID and the process table entry of the zombie process can be reused.
* If the parent process does not use the wait() system call, the zombie process is left in the process table. This creates a resource leak.
* If the parent process is not running anymore, then the presence of a zombie process indicates an operating system bug. This may not be a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system such as running out of process table entries.
* The zombie processes can be removed from the system by sending the SIGCHLD signal to the parent, using the kill command. If the zombie process is still not eliminated from the process table by the parent process, then the parent process is terminated if that is acceptable.

**Dangers of Zombie Processes**

Zombie processes don't use any system resources but they do retain their process ID. If there are a lot of zombie processes, then all the available process ID’s are monopolized by them. This prevents other processes from running as there are no process ID’s available.

The presence of zombie processes also indicates an operating system bug if their parent processes are not running anymore. This is not a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system.

**Getting Rid of Zombie Processes**

**The top command, and the ps command display zombie processes.**

You can’t kill zombie processes as you can kill normal processes with the SIGKILL signal — zombie processes are already dead. Bear in mind that you don’t need to get rid of zombie processes unless you have a large amount on your system – a few zombies are harmless. However, there are a few ways you can get rid of zombie processes.

One way is by sending the SIGCHLD signal to the parent process. This signal tells the parent process to execute the wait() system call and clean up its zombie children. Send the signal with the kill command, replacing pid in the command below with the parent process’s PID:

$ kill -s SIGCHLD pid

However, if the parent process isn’t programmed properly and is ignoring SIGCHLD signals, this won’t help. You’ll have to kill or close the zombies’ parent process. When the process that created the zombies ends, init inherits the zombie processes and becomes their new parent. (init is the first process started on Linux at boot and is assigned PID 1.) init periodically executes the wait() system call to clean up its zombie children, so init will make short work of the zombies. You can restart the parent process after closing it.

If a parent process continues to create zombies, it should be fixed so that it properly calls wait() to reap its zombie children. File a bug report if a program on your system keeps creating zombies.

**Killing Zombies!**

Also known as “defunct” or “dead” process – In simple words, a Zombie process is one that is dead but is present in the system’s process table. Ideally, it should have been cleaned from the process table once it completed its job/execution but for some reason, its parent process didn’t clean it up properly after the execution.

In order to kill a Zombie process, we need to identify it first. The following command can be used to find zombie processes:

$ ps aux | egrep "Z|defunct"

Z in the STAT column and/or [defunct] in the last (COMMAND) column of the output would identify a Zombie process.

Now practically you can’t kill a Zombie because it is already dead! What can be done is to notify its parent process explicitly so that it can retry to read the child (dead) process’s status and eventually clean them from the process table. This can be done by sending a SIGCHLD signal to the parent process. The following command can be used to find the parent process ID (PID):

$ ps -o ppid= <Child PID>

Once you have the Zombie’s parent process ID, you can use the following command to send a SIGCHLD signal to the parent process:

$ kill -s SIGCHLD <Parent PID>

However, if this does not help clearing out the Zombie process, you will have to kill or restart its parent process OR in case of a huge surge in Zombie processes causing or heading towards system outage, you will have no choice but to go for a system reboot. The following command can be used to kill its parent process:

$ kill -9 <Parent PID>

Note that killing a parent process will affect all of its child processes, so a quick double check will be helpful to be safe. Alternatively, if few lying zombie processes are not consuming much CPU/Memory, it’s better to kill the parent process or reboot the system in the next scheduled system maintenance.

### How to Kill the Zombie ??

Identify the zombie processes

$ top -b1 -n1 | grep Z

Find the parent id of our defunct process:

$ ps -A -ostat,pid,ppid | grep -e '[zZ]'

Send signal to read the entry

$kill -s SIGCHLD 103

Kill it If above step not worked

$kill -9 103

### Linux Process States

1. RUNNING & RUNNABLE R  
2. INTERRRUPTABLE\_SLEEP S  
3. UNINTERRUPTABLE\_SLEEP D  
4. STOPPED T  
5. ZOMBIE Z

RUNNING & RUNNABLE

When the CPU executes a process, it will be in a RUNNING state. When the process is not waiting for any resource and ready to be executed by the CPU, it will be in the RUNNABLE state.

**SLEEPING**

SLEEPING state indicates the process is currently waiting on certain resources (like waiting on I/O, waiting on locks, application code making the process to sleep,…). There are two types of SLEEPING processes:

INTERRUPTABLE\_SLEEP: When a process is in INTERRUPTABLE\_SLEEP, it will wake up from the middle of sleep and process new signals sent to it.

UNINTERRUPTABLE\_SLEEP: When a process is in UNINTERRUPTABLE\_SLEEP, it will not wake up from the middle of sleep even though new signals are sent to it.

**How to kill the SLEEPING Process ?**

If the process is in INTERRUPTABLE\_SLEEP state, then issuing SIGKILL signal (i.e. ‘kill -9’) to the process, will terminate the process immediately. On the other hand, if the process is in UNINTERRUPTABLE\_SLEEP state, then the issuing SIGKILL signal will not terminate immediately. Only after the process completes its sleep/waiting operation it will terminate. Thus, if you would like to kill a process which is in UNINTERRUPTABLE\_SLEEP state for a prolonged period then you have to reboot the system, there is no other way.

STOPPED

STOPPED state indicates that the process has been suspended from proceeding further. In Linux when you issue the ‘Ctrl + Z’ command it will issue a SIGSTOP signal to the process. When the process receives this signal it will be suspended/stopped from executing further. When a process is in STOPPED state, it will only handle SIGKILL and SIGCONT signals. SIGKILL signal will terminate the process, but the SIGCONT signal will put the process back into RUNNING/RUNNABLE state.

ZOMBIE

A process will terminate when it calls ‘system exit’ API or when someone else kills the process. When a process terminates, it will release all the data structures and the resources it holds. However, it will not release its slot in the ‘process’ table. Instead, the process will send a SIGCHLD signal to its parent process. Now it’s up to the parent process to release the child process slot in the ‘process’ table. The process will be in ZOMBIE state from the time the child process issues the SIGCHLD signal until the parent process releases the slot in the ‘process’ table.

### Sessions

A session contains a number of process groups, and a process group contains a number of processes, and a process contains a number of threads.

Users normally interact with groups of related processes. Although they initially log in to a single terminal and use a single process (their **shell**, which provides a command-line interface), users end up running many processes as a result of actions such as

* Running noninteractive tasks in the background
* Switching among interactive tasks via **job control**, which is discussed more fully in Chapter 15
* Starting multiple processes that work together through pipes
* Running a windowing system, such as the X Window System, which allows multiple terminal windows to be opened

In order to manage all of these processes, the kernel needs to group the processes in ways more complicated than the simple parent-child relationship we have already discussed. These groupings are called **sessions** and **process groups**

A session can have a controlling tty. At most one process group in a session can be a foreground process group. An interrupt character typed on a tty ("Teletype", i.e., terminal) causes a signal to be sent to all members of the foreground process group in the session (if any) that has that tty as controlling tty.

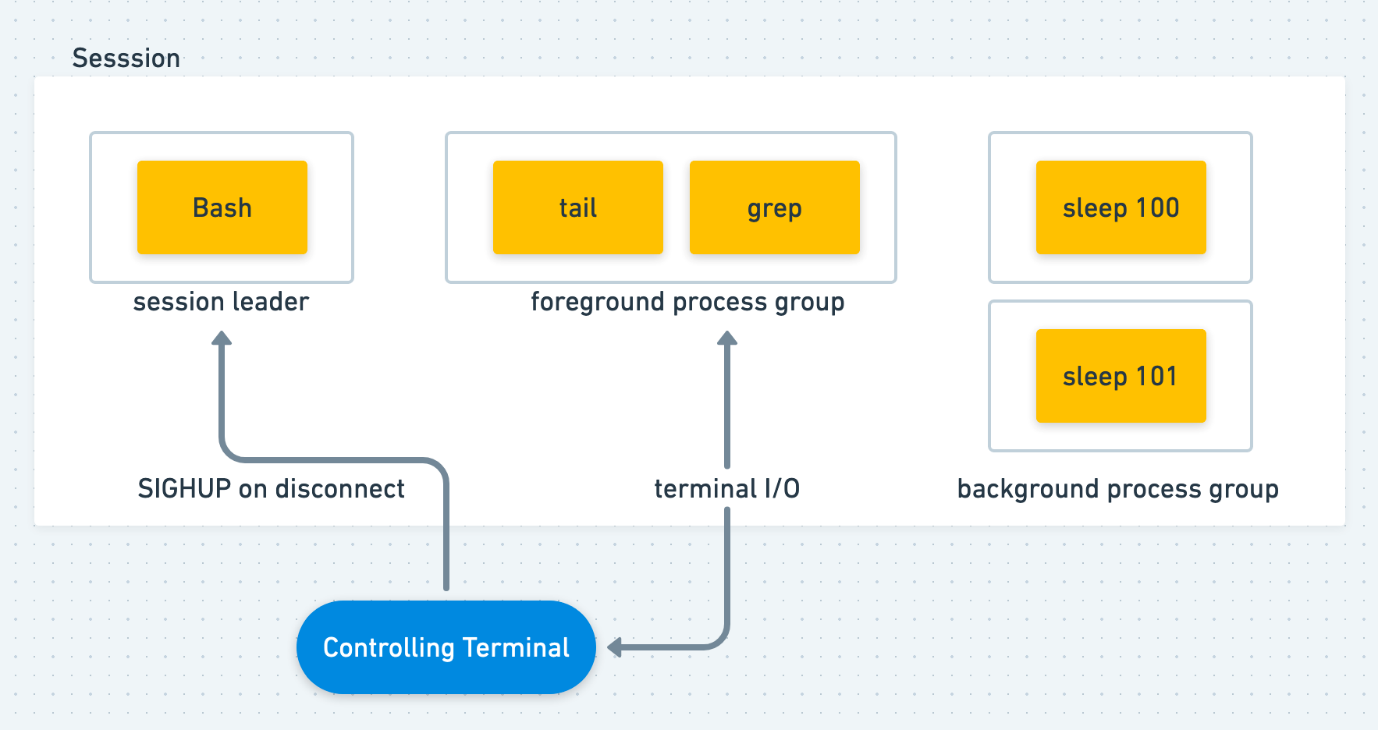
All these objects have numbers, and we have thread IDs, process IDs, process group IDs and session IDs.

When a user logs out of a system, the kernel needs to terminate all the processes the user had running (otherwise, users would leave a slew of old processes sitting around waiting for input that can never arrive). To simplify this task, processes are organized into sets of **sessions**. The session's ID is the same as the pid of the process that created the session through the setsid() system call. That process is known as the **session leader** for that session group. All of that process's descendants are then members of that session unless they specifically remove themselves from it. The setsid() function does not take any arguments and returns the new session ID.

#include <unistd.h>

pid\_t setsid(void);

Every session is tied to a terminal from which processes in the session get their input and to which they send their output. That terminal may be the machine's local console, a terminal connected over a serial line, or a pseudo terminal that maps to an X window or across a network (see Chapter 16 for information on pseudo terminal devices). The terminal to which a session is related is called the **controlling terminal** (or **controlling tty**) of the session. A terminal can be the controlling terminal for only one session at a time.



A session is a collection of process groups, usually associated with one controlling terminals and a session leader process. If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.

### How to Kill the session ?? loginctl

***Getting session details***

*$ loginctl list-sessions*

***View the Session properties***

*$ loginctl show-session <session numer>*

*lock-sessions <id>*

*unlock-sessions <id>*

*terminate-sesison <id>*

*kill-session <id>*

### What is Process Groups ??

In addition to having a process ID, each process belongs to a **process group**.

* A process group is a collection of one or more processes (usually associated with the same job) that can receive signals from the same terminal.
* Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a pid\_t data type.

Process Group Lifetime

The process group life time is the period of time that begins when the group is created and ends when the last remaining process leaves the group. It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. The last remaining process in the process group can either terminate or enter some other process group.

Sessions and process groups have a few other characteristics.

* A session can have a single **controlling terminal**. This is usually the terminal device (in the case of a [terminal login](https://notes.shichao.io/apue/ch9/##terminal-logins)) or pseudo terminal device (in the case of a [network login](https://notes.shichao.io/apue/ch9/#network-logins)) on which we log in.
* The session leader that establishes the connection to the controlling terminal is called the **controlling process**.
* The process groups within a session can be divided into a single **foreground process group** and one or more **background process groups**.
* If a session has a controlling terminal, it has a single foreground process group and all other process groups in the session are background process groups.
* Whenever we press the terminal’s interrupt key (often DELETE or Control-C), the interrupt signal is sent to all processes in the foreground process group.
* Whenever we press the terminal’s quit key (often Control-backslash), the quit signal is sent to all processes in the foreground process group.
* If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

### Job Control

Job control is a feature of the shell which allows a single shell instance to run and manage multiple commands.

**Job control** allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are run in the background.

A job is simply a collection of processes, often a pipeline of processes.

Only one job can read input and keyboard-generated signals from a particular window at at time. Process that are part of that job are foreground process of that controlling terminal.

A Background process of that controlling terminal is a member of any other job associated with that terminal. It cannot read input or recive keyboard-generated interrupts from the terminal.

A job in the background may be stopped [suspended] or it may be running.

**Running jobs in the backround**

* Any command or pipeline can be started in the backround by appending an ampersand (&) to the end of the command line.
* The bash shell displays a job number [unique to the session] and the PID pf the new child process.

***Running jobs in the background***

*$ sleep 10000 &*

***To Display all the jobs***

*$ jobs*

*[1]+ Running sleep 10000 &*

***Background job can be brought Foreground***

*$ fg %jobid $fg %1*

*It can be suspended usng ctrl + Z and killed using ctrl + c*

*# ps j display information related to jobs*

***Start the suspended process running in the background***

*$ bg %job id $gb %1*

### What is Thread ????

How process works ??

* When we start executing the program, the processor begins to process it. It takes the following steps:
* Firstly, the program is loaded into the computer's memory in binary code after translation.
* A program requires memory and other OS resources to run it. The resources such that registers, program counter, and a stack, and these resources are provided by the OS.
* A register can have an instruction, a storage address, or other data that is required by the process.
* The program counter maintains the track of the program sequence.
* The stack has information on the active subroutines of a computer program.
* A program may have different instances of it, and each instance of the running program is knowns as the individual process.

**Features of Process**

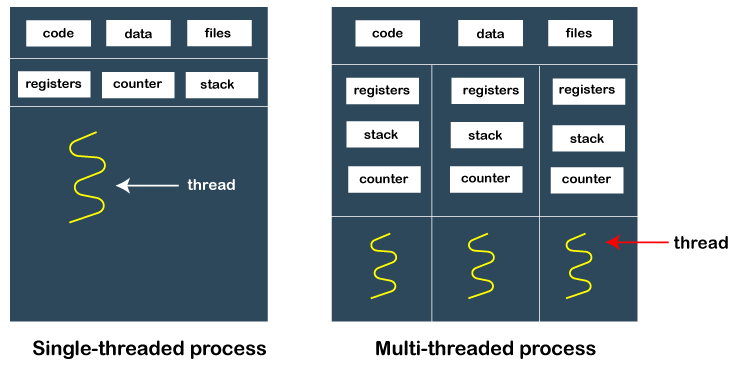
* Each time we create a process, we need to make a separate system call for each process to the OS. The fork() function creates the process.
* Each process exists within its own address or memory space.
* Each process is independent and treated as an isolated process by the OS.
* Processes need IPC (Inter-process Communication) in order to communicate with each other.
* A proper synchronization between processes is not required.

Process = Program + State of all threads executing in the program

What is Thread ???

Light weight process.

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel.  
A thread is a path of execution within a process. A process can contain multiple threads

A thread is the subset of a process and is also known as the lightweight process. A process can have more than one thread, and these threads are managed independently by the scheduler. All the threads within one process are interrelated to each other. Threads have some common information, such as data segment, code segment, files, etc., that is shared to their peer threads. But contains its own registers, stack, and counter.

**How does thread work?**

As we have discussed that a thread is a subprocess or an execution unit within a process. A process can contain a single thread to multiple threads. A thread works as follows:

When a process starts, OS assigns the memory and resources to it. Each thread within a process shares the memory and resources of that process only.

Threads are mainly used to improve the processing of an application. In reality, only a single thread is executed at a time, but due to fast context switching between threads gives an illusion that threads are running parallelly.

If a single thread executes in a process, it is known as a single-threaded And if multiple threads execute simultaneously, then it is known as multithreading.

**Types of Threads**

There are two types of threads, which are:

1. User Level Thread

As the name suggests, the user-level threads are only managed by users, and the kernel does not have its information.

These are faster, easy to create and manage.

The kernel takes all these threads as a single process and handles them as one process only.

The user-level threads are implemented by user-level libraries, not by the system calls.

2. Kernel-Level Thread

The kernel-level threads are handled by the Operating system and managed by its kernel. These threads are slower than user-level threads because context information is managed by the kernel. To create and implement a kernel-level thread, we need to make a system call.

**Features of Thread**

* Threads share data, memory, resources, files, etc., with their peer threads within a process.
* One system call is capable of creating more than one thread.
* Each thread has its own stack and register.
* Threads can directly communicate with each other as they share the same address space.
* Threads need to be synchronized in order to avoid unexpected scenarios.

**Why Multithreading?**

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below.

**Why Threads are Required?**

Now, one would ask why do we need multiple threads in a process?? Why can’t a process with only one (default) main thread be used in every situation.

Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output. Now, if the process is not multi-threaded ie if the process does not involve multiple threads, then the whole processing in the process becomes synchronous. This means that the process takes an input processes it and produces an output

The limitation in the above design is that the process cannot accept an input until its done processing the earlier one and in case processing an input takes longer than expected then accepting further inputs goes on hold.

To consider the impact of the above limitation, if we map the generic example above with a socket server process that can accept input connection, process them and provide the socket client with output. Now, if in processing any input if the server process takes more than expected time and in the meantime another input (connection request) comes to the socket server then the server process would not be able to accept the new input connection as its already stuck in processing the old input connection. This may lead to a connection time out at the socket client which is not at all desired.

|  |  |
| --- | --- |
| **Process** | **Thread** |
| A process is an instance of a program that is being executed or processed. | Thread is a segment of a process or a lightweight process that is managed by the scheduler independently. |
| Processes are independent of each other and hence don't share a memory or other resources. | Threads are interdependent and share memory. |
| Each process is treated as a new process by the operating system. | The operating system takes all the user-level threads as a single process. |
| If one process gets blocked by the operating system, then the other process can continue the execution. | If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process. |
| Context switching between two processes takes much time as they are heavy compared to thread. | Context switching between the threads is fast because they are very lightweight. |
| The data segment and code segment of each process are independent of the other. | Threads share data segment and code segment with their peer threads; hence are the same for other threads also. |
| The operating system takes more time to terminate a process. | Threads can be terminated in very little time. |
| New process creation is more time taking as each new process takes all the resources. | A thread needs less time for creation. |

### Thread Command

***All the thread that a process has.***

*$ ps -e -T | grep <application name or pid>*

*$ ps -e -T | grep clementine T display the thread in SPID column*

*$ ps -o nlwp <pid>*

***To see the Thread count os process***

*On proc pseudo file system, there is a task directory which records thread information:*

*$ cat /proc/1041/status*

*# ls -l /proc/3692/task*

### What is Thread Dump ??

A thread dump is a snapshot of the state of all the threads of a Java process. The state of each thread is presented with a stack trace, showing the content of a thread's stack. A thread dump is useful for diagnosing problems as it displays the thread's activity. Thread dumps are written in plain text, so we can save their contents to a file and look at them later in a text editor.

When the process is executing, we can detect the current state of execution of the threads in the process using **thread dumps**. A thread Dump contains a snapshot of all the threads active at a particular point during the execution of a program. It contains all relevant information about the thread and its current state.

A modern application today involves multiple numbers of threads. Each thread requires certain resources, performs certain activities related to the process. This can boost the performance of an application as threads can utilize available CPU cores.

But there’s are trade-offs, e.g., sometimes multiple threads may not coordinate well with each other and a deadlock situation may arise. So, if something goes wrong, we can use thread dumps to inspect the state of our threads.

What is it useful for ?

Finding Deadlocks

Debugging slow/hung operations in an environment you don’t have direct access to

Finding problems when you cant really reproduce the issue.

A JVM thread Dump is a listing of the state of all threads that are part of the process at that particular point of time. It contains information about the thread’s stack, presented as a stack trace. As it is written in plaintext, the contents can be saved for reviewing later. Analysis of thread dumps can help in

* Optimizing JVM performance
* Optimizing application performance
* Diagnosing problems, e.g. a deadlock, thread contention, etc.

How do I get one ?

Jstack

Spring boot Actuator endpoint

Monitoring tools /JMX