

- Balancing can be achieved through either push migration or pull migration:
- ✓ Push migration involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.
- ✓ Pull migration involves idle processors taking processes from the ready queues of other processors.
- ✓ Push and pull migration are not mutually exclusive.

Operations on Processes:

There are two major operations:

1. Process Creation
2. Process Termination

How Does Linux Identify Processes?

- Because Linux is a multi-user system, meaning different users can be running various programs on the system, each running instance of a program must be identified uniquely by the kernel.
- And a program is identified by its process ID (PID) as well as it's parent processes ID (PPID).

The Init Process

- ❖ **Init** process is the mother (parent) of all processes on the system, it's the first program that is executed when the Linux system boots up; it manages all other processes on the system. It is started by the kernel itself, so in principle it does not have a parent process.
- ❖ The init process always has process ID of 1. It functions as an adoptive parent for all orphaned processes.

- You can use the pidof command to find the ID of a process:

```
[root@tecmint ~]# pidof systemd
1
[root@tecmint ~]# pidof top
2160
[root@tecmint ~]# pidof httpd
2103 2102 2101 2100 2099 1076
[root@tecmint ~]#
```

- To find the process ID and parent process ID of the current shell, run:

```
[root@tecmint ~]# echo $$
2109
[root@tecmint ~]# echo $PPID
2106
[root@tecmint ~]#
```

1. Process creation:

The fork () Function:

- ❖ The fork () function is used to create a new process by duplicating the existing process from which it is called.
- ❖ The existing process from which this function is called becomes the parent process and the newly created process becomes the child process. As already stated that child is a duplicate copy of the parent but there are some exceptions to it.
 - The child has a unique PID like any other process running in the operating system.
 - The child has a parent process ID which is same as the PID of the process that created it.
 - Resource utilization and CPU time counters are reset to zero in child process.
 - Set of pending signals in child is empty.
 - Child does not inherit any timers from its parent

Return Type:

- **Negative Value:** creation of a child process was unsuccessful.
- **Zero:** Returned to the newly created child process.
- **Positive value:** Returned to parent or caller. The value contains process ID of newly created child process.

Example:

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int main( )
{
    pid_t child_pid;
    child_pid = fork ( );           // Create a new child process;
    if (child_pid >= 0)
    {
        if (child_pid == 0)
        {
            printf ("child process successfully created!!\n");
            printf ("child PID = %d, parent PID = %d\n", getpid( ), getppid( ) );
            exit(0);
        }
    }
    else
    {
        perror("fork");
        exit(0);
    }
}
```

Output:

```

terminal
Mon 09:55
student@localhost:~/lp
File Edit View Search Terminal Help
[student@localhost lp]$ cc fork.c
[student@localhost lp]$ ./a.out
child process successfully created!!
child PID = 2743, parent PID = 1
[student@localhost lp]$

```

2. Process termination:

- The `exit ()` system call terminates the process normally.
- Status: Status value returned to the parent process. Generally, a status value of 0 or `EXIT_SUCCESS` indicates success, and any other value or the constant `EXIT_FAILURE` is used to indicate an error.

Example:

```

/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    FILE * fd;
    fd = fopen ("myfile.txt", "r");
    if (fd == NULL)
    {
        printf ("Error opening file");
        exit (1);
    }
    else
    {
        /* file operations here */
    }
    return 0;
}

```

- ❖ When `exit ()` is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, `init`, and the process parent is sent a `SIGCHLD` signal.

3. exec() system call - Replacing a process image:

“The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program”

- ❖ More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.
- ❖ The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().
- ❖ The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.
- ❖ If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process.
- ❖ PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

➤ **exec system** call is a collection of functions as follows:

1. execl ✓
2. execv.... etc. ✓

Synopsis:

```
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
```

```
int execv(const char *path, char *const argv[]);
```

Description:

- The **exec()** family of functions replaces the current process image with a new process image.
- The initial argument for these functions is the name of a file that is to be executed.
- The *const char *arg* and subsequent ellipses in the **execl()** function can be thought of as *arg0*, *arg1*, ..., *argn*. It describes a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.
- The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer.
- The **execv** function provides an array of pointers to null-terminated string that represent the argument list available to the new program.

- The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

Return Value:

The **exec ()** functions return only if an error has occurred. The return value is -1, and [*errno*](#) is set to indicate the error.

Example:

Consider the following example in which we have used exec system call in C programming in Linux. We have two c files here example.c and hello.c:

example.c

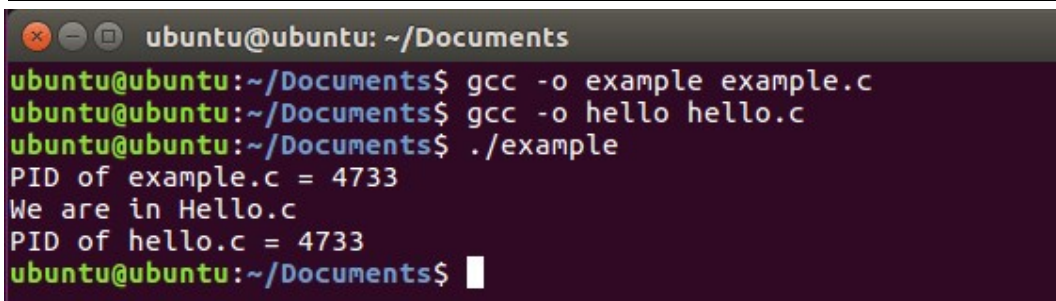
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

hello.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

OUTPUT:

```
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
```



```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

- In the above example we have an example.c file and hello.c file. In the example .c file first of all we have printed the ID of the current process (file example.c is running in current process). Then in the next line we have created an array of character pointers. The last element of this array should be NULL as the terminating point.
- Then we have used the function `execv()` which takes the file name and the character pointer array as its argument. It should be noted here that we have used `./` with the name of file, it specifies the path of the file. As the file is in the folder where example.c resides so there is no need to specify the full path.
- When `execv()` function is called, our process image will be replaced now the file example.c is not in the process but the file hello.c is in the process. It can be seen that the process ID is same whether hello.c is process image or example.c is process image because process is same and process image is only replaced.
- Then we have another thing to note here which is the `printf()` statement after `execv()` is not executed. This is because control is never returned back to old process image once new process image replaces it. The control only comes back to calling function when replacing process image is unsuccessful. (The return value is -1 in this case).

Difference between fork() and exec() system calls:

- The `fork()` system call is used to create an exact copy of a running process and the created copy is the child process and the running process is the parent process. Whereas, `exec()` system call is used to replace a process image with a new process image. Hence there is no concept of parent and child processes in `exec()` system call.
- In `fork()` system call the parent and child processes are executed at the same time. But in `exec()` system call, if the replacement of process image is successful, the control does not return to where the `exec` function was called rather it will execute the new process. The control will only be transferred back if there is any error.

Example: Combining fork() and exec() system calls

Consider the following example in which we have used both `fork()` and `exec()` system calls in the same program:

example.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    pid_t p;
```

```

p = fork();
if(p == -1)
{
    printf("There is an error while calling fork()");
}
if(p == 0)
{
    printf("We are in the child process\n");
    printf("Calling hello.c from child process\n");
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
}
else
{
    printf("We are in the parent process");
}
return 0;
}

```

hello.c:

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}

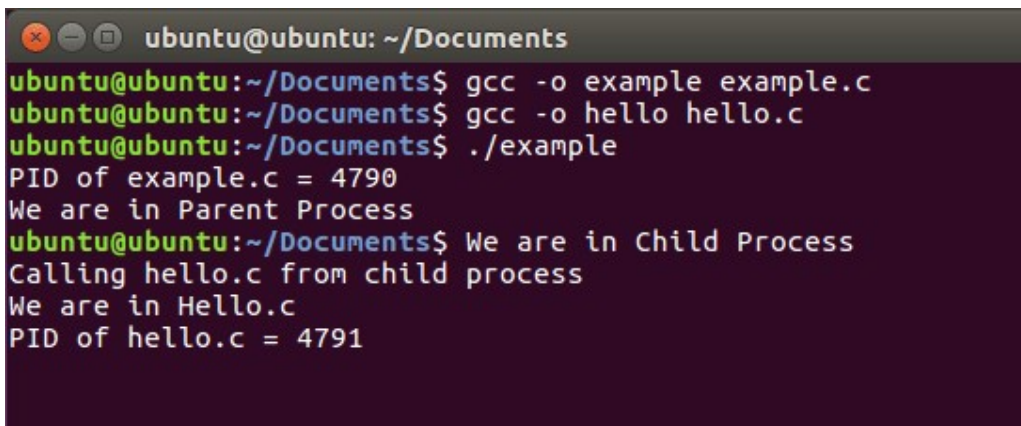
```

OUTPUT:

```

PID of example.c = 4790
We are in Parent Process
We are in Child Process
Calling hello.c from child process
We are in hello.c
PID of hello.c = 4791

```



```

ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4790
We are in Parent Process
ubuntu@ubuntu:~/Documents$ We are in Child Process
Calling hello.c from child process
We are in Hello.c
PID of hello.c = 4791

```

- ❖ In this example we have used `fork()` system call. When the child process is created 0 will be assigned to `p` and then we will move to the child process. Now the block of statements with `if(p == 0)` will be executed. A message is displayed and we have used `execv ()` system call and the current child process image which is `example.c` will be replaced with `hello.c`. Before `execv()` call child and parent processes were same.
- ❖ It can be seen that the PID of `example.c` and `hello.c` is different now. This is because `example.c` is the parent process image and `hello.c` is the child process image.

Waiting for a process:

`wait()` and `waitpid()`

`wait`, `waitpid` - wait for process to change state

Synopsis:

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Description:

- ❖ All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

A state change is considered to be:

- ✓ the child terminated;
 - ✓ the child was stopped by a signal;
 - ✓ or the child was resumed by a signal.
- ❖ In the case of a terminated child, performing a `wait` allows the system to release the resources associated with the child; if a `wait` is not performed, then terminated the child remains in a "zombie" state.
 - ❖ The `wait()` system call suspends execution of the current process until one of its children terminates. The call `wait(&status)` is equivalent to: `waitpid(-1, &status, 0)`;

- ❖ The **waitpid()** system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid()** waits only for terminated children, but this behaviour is modifiable via the *options* argument, as described below.

The value of *pid* can be:

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of <i>pid</i> .
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of <i>pid</i> .

Return Value:

- **wait()**: on success, returns the process ID of the terminated child; on error, -1 is returned.
- **waitpid()**: on success, returns the process ID of the child whose state has changed; on error, -1 is returned; if **WNOHANG** was specified and no child(ren) specified by *pid* has yet changed state, then 0 is returned.

Errors:

Tag	Description
ECHILD	(for wait()) The calling process does not have any unwaited-for children.
ECHILD	(for waitpid() or waitid()) The process specified by <i>pid</i> (waitpid()) or <i>idtype</i> and <i>id</i> (waitid()) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for SIGCHLD is set to SIG_IGN . See also the LINUX NOTES section about threads.)
EINTR	WNOHANG was not set and an unblocked signal or a SIGCHLD was caught.
EINVAL	The <i>options</i> argument was invalid.