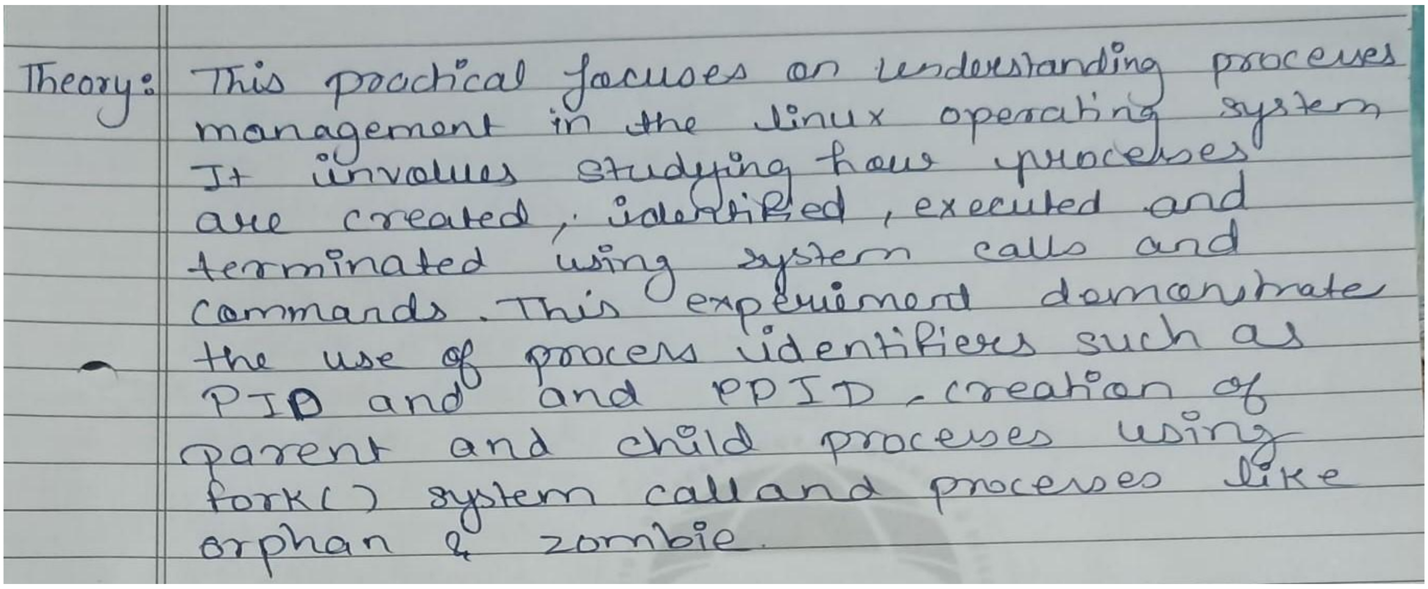
AIM: Adam is working in an IT company. He has been given a task to reduce the load of a system by killing some of the processes running in the LINUX operating system. Which commands will he use to complete the given task with the help of the following operation?

- Kill processes by name
- Kill a process based on the process name
- Kill a single process at a time with the given process ID

2.Write a program for process creation using C
- Orphan Process
- Zombie Process

3. Create the process using fork () system call.
- Child Process creation
- Parent process creation
- PPID and PID

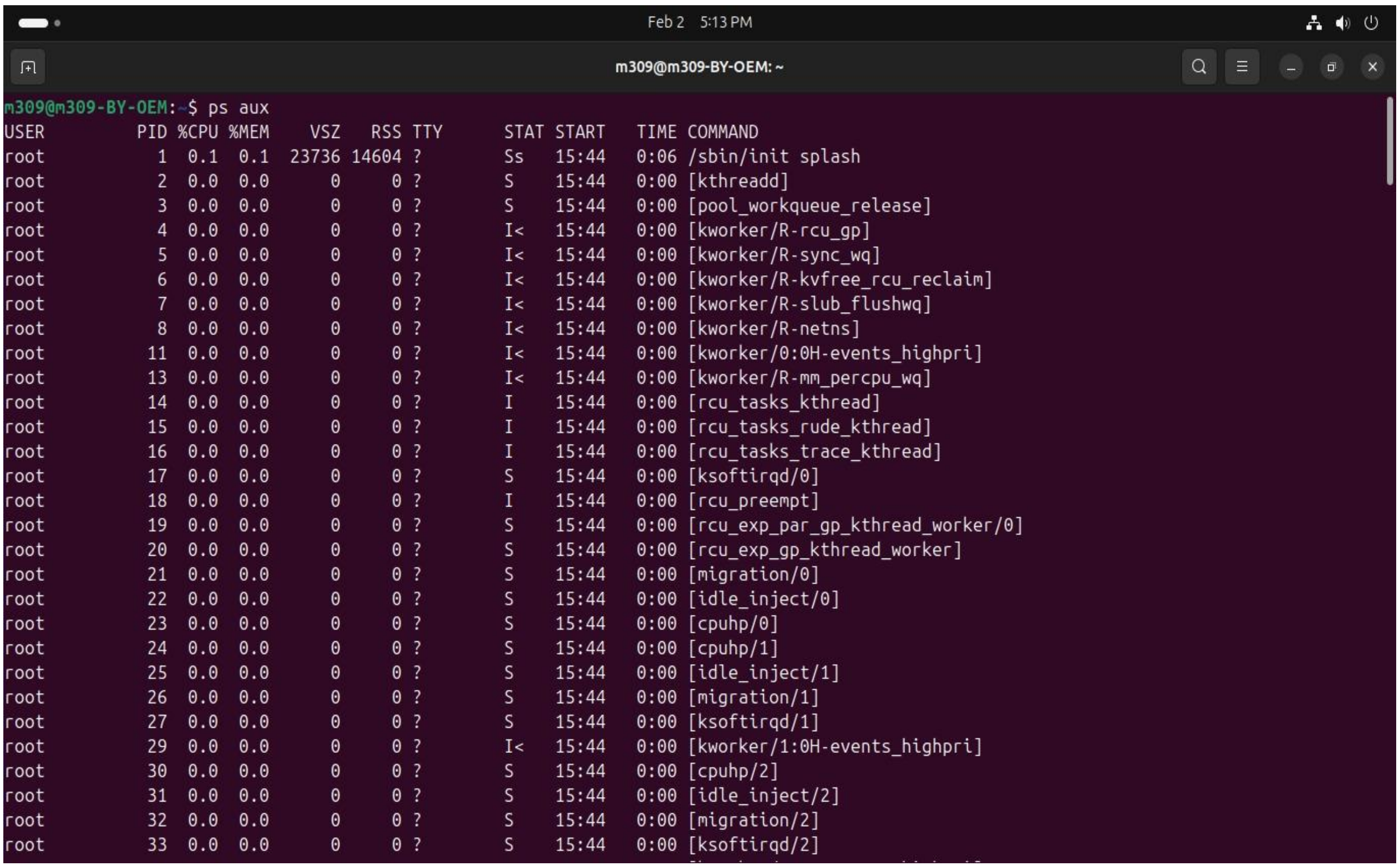
THEORY:



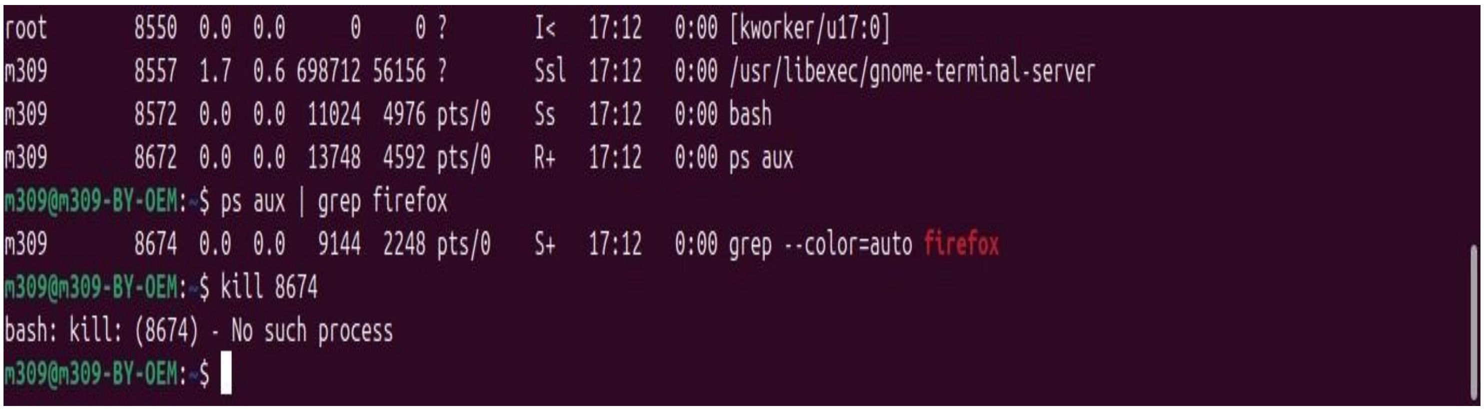
Theory: This practical focuses on understanding processes management in the linux operating system. It involves studying how processes are created, identified, executed and terminated using system calls and commands. This experiment demonstrate the use of process identifiers such as PID and and PPID, creation of parent and child processes using fork() system call and processes like orphan & zombie.

- Kill processes by name
- Kill a process based on the process name
- Kill a single process at a time with the given process ID

COMMANDS:



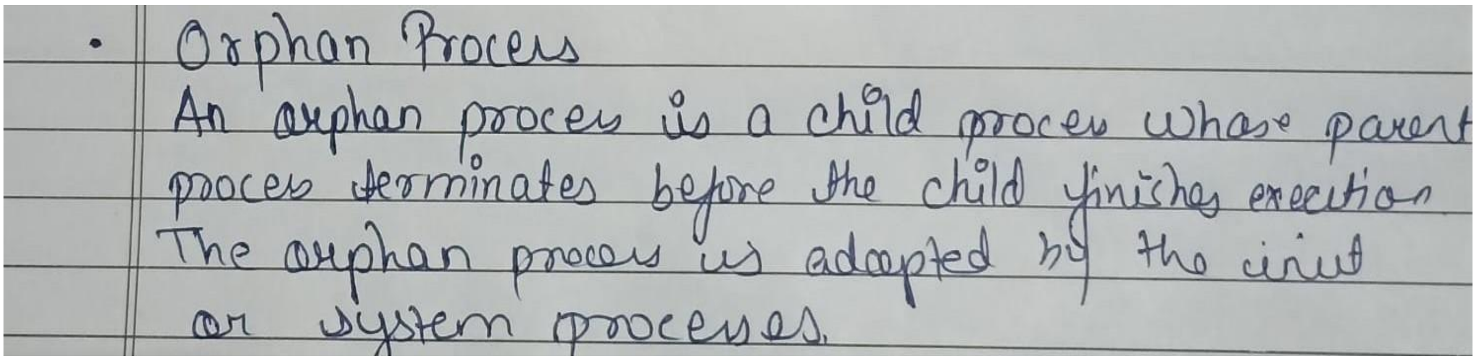
```
m309@m309-BY-OEM:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.1  23736 14604 ?        Ss   15:44   0:06 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    15:44   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    15:44   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-sync_wq]
root           6  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-kvfree_rcu_reclaim]
root           7  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-netns]
root          11  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/0:0H-events_highpri]
root          13  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        I    15:44   0:00 [rcu_tasks_kthread]
root          15  0.0  0.0      0     0 ?        I    15:44   0:00 [rcu_tasks_rude_kthread]
root          16  0.0  0.0      0     0 ?        I    15:44   0:00 [rcu_tasks_trace_kthread]
root          17  0.0  0.0      0     0 ?        S    15:44   0:00 [ksoftirqd/0]
root          18  0.0  0.0      0     0 ?        I    15:44   0:00 [rcu_preempt]
root          19  0.0  0.0      0     0 ?        S    15:44   0:00 [rcu_exp_par_gp_kthread_worker/0]
root          20  0.0  0.0      0     0 ?        S    15:44   0:00 [rcu_exp_gp_kthread_worker]
root          21  0.0  0.0      0     0 ?        S    15:44   0:00 [migration/0]
root          22  0.0  0.0      0     0 ?        S    15:44   0:00 [idle_inject/0]
root          23  0.0  0.0      0     0 ?        S    15:44   0:00 [cpuhp/0]
root          24  0.0  0.0      0     0 ?        S    15:44   0:00 [cpuhp/1]
root          25  0.0  0.0      0     0 ?        S    15:44   0:00 [idle_inject/1]
root          26  0.0  0.0      0     0 ?        S    15:44   0:00 [migration/1]
root          27  0.0  0.0      0     0 ?        S    15:44   0:00 [ksoftirqd/1]
root          29  0.0  0.0      0     0 ?        I<   15:44   0:00 [kworker/1:0H-events_highpri]
root          30  0.0  0.0      0     0 ?        S    15:44   0:00 [cpuhp/2]
root          31  0.0  0.0      0     0 ?        S    15:44   0:00 [idle_inject/2]
root          32  0.0  0.0      0     0 ?        S    15:44   0:00 [migration/2]
root          33  0.0  0.0      0     0 ?        S    15:44   0:00 [ksoftirqd/2]
```



```
root        8550  0.0  0.0      0     0 ?        I<   17:12   0:00 [kworker/u17:0]
m309        8557  1.7  0.6 698712 56156 ?        Ssl  17:12   0:00 /usr/libexec/gnome-terminal-server
m309        8572  0.0  0.0  11024  4976 pts/0    Ss   17:12   0:00 bash
m309        8672  0.0  0.0  13748  4592 pts/0    R+   17:12   0:00 ps aux
m309@m309-BY-OEM:~$ ps aux | grep firefox
m309        8674  0.0  0.0   9144  2248 pts/0    S+   17:12   0:00 grep --color=auto firefox
m309@m309-BY-OEM:~$ kill 8674
bash: kill: (8674) - No such process
m309@m309-BY-OEM:~$ 
```

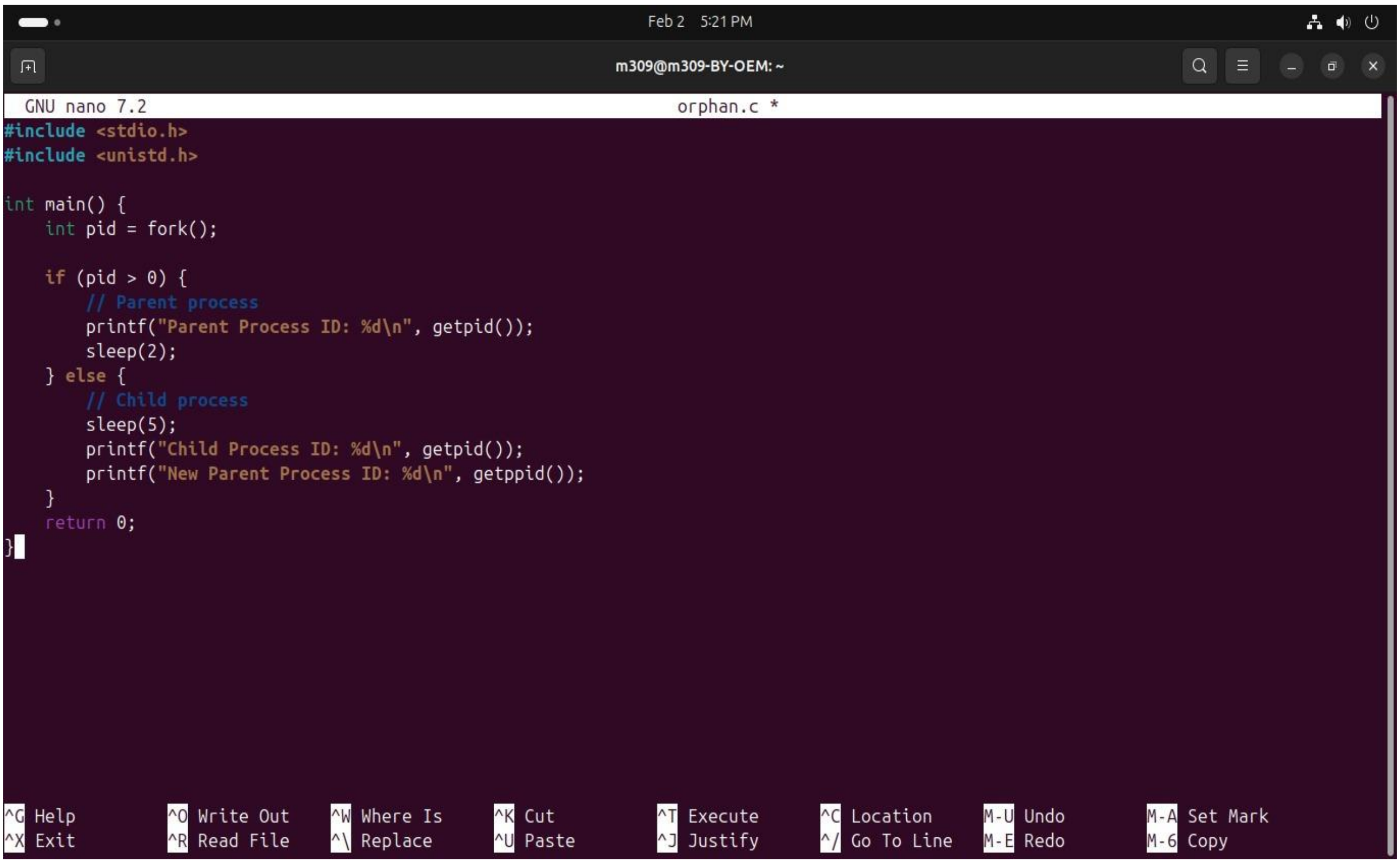
2.Write a program for process creation using C

- Orphan Process

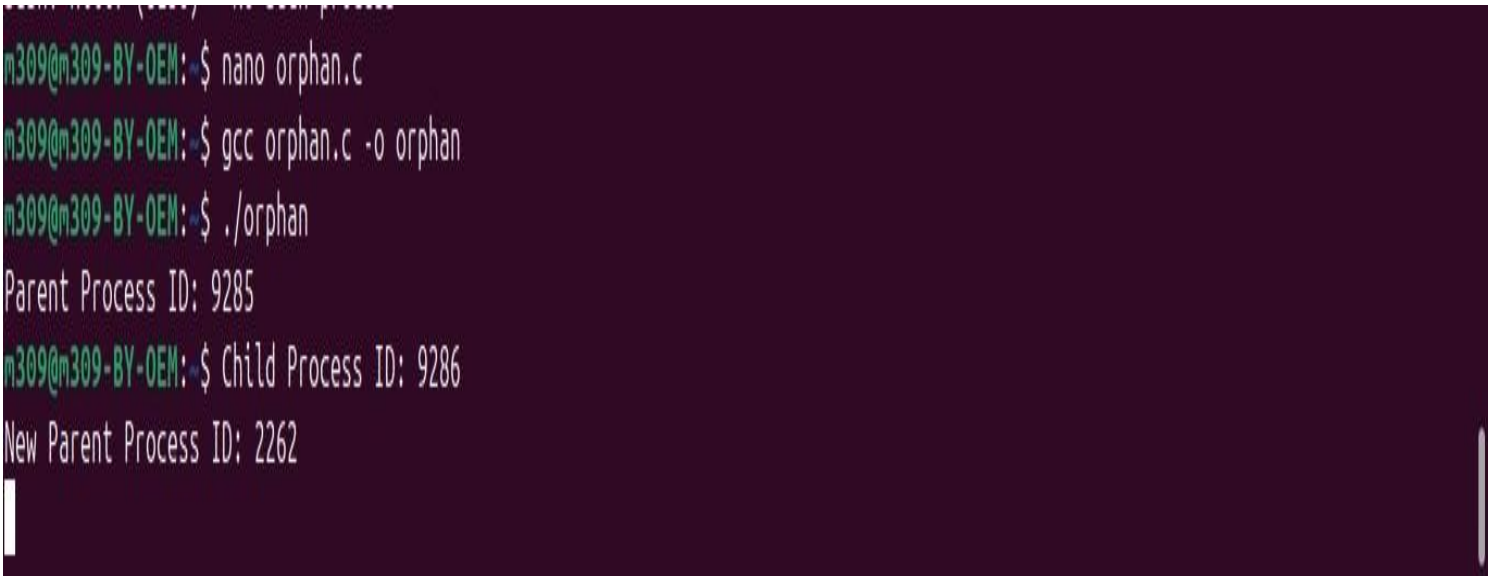


- Orphan Process

An orphan process is a child process whose parent process terminates before the child finishes execution. The orphan process is adopted by the init or system processes.

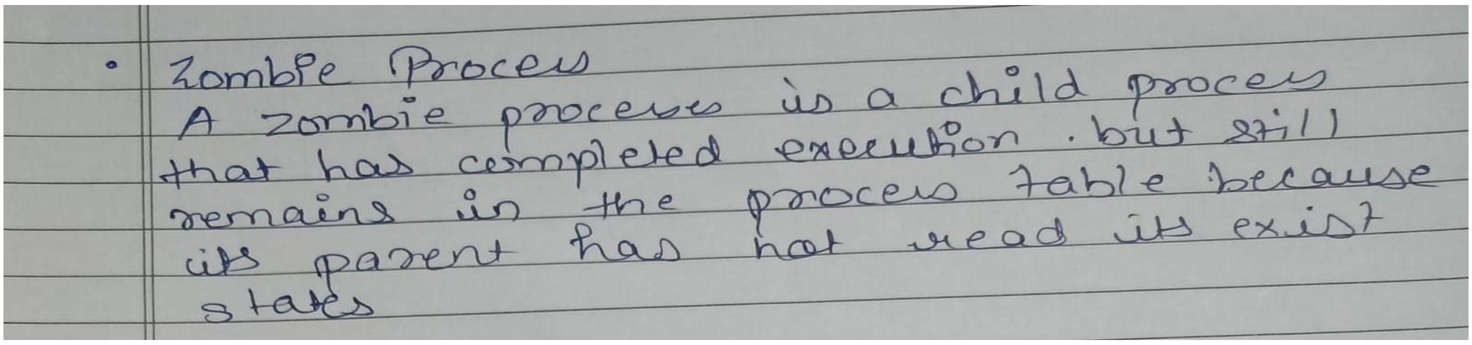


```
Feb 2   5:21 PM
m309@m309-BY-OEM: ~
  GNU nano 7.2                                      orphan.c *
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid = fork();

    if (pid > 0) {
        // Parent process
        printf("Parent Process ID: %d\n", getpid());
        sleep(2);
    } else {
        // Child process
        sleep(5);
        printf("Child Process ID: %d\n", getpid());
        printf("New Parent Process ID: %d\n", getppid());
    }
    return 0;
}
```

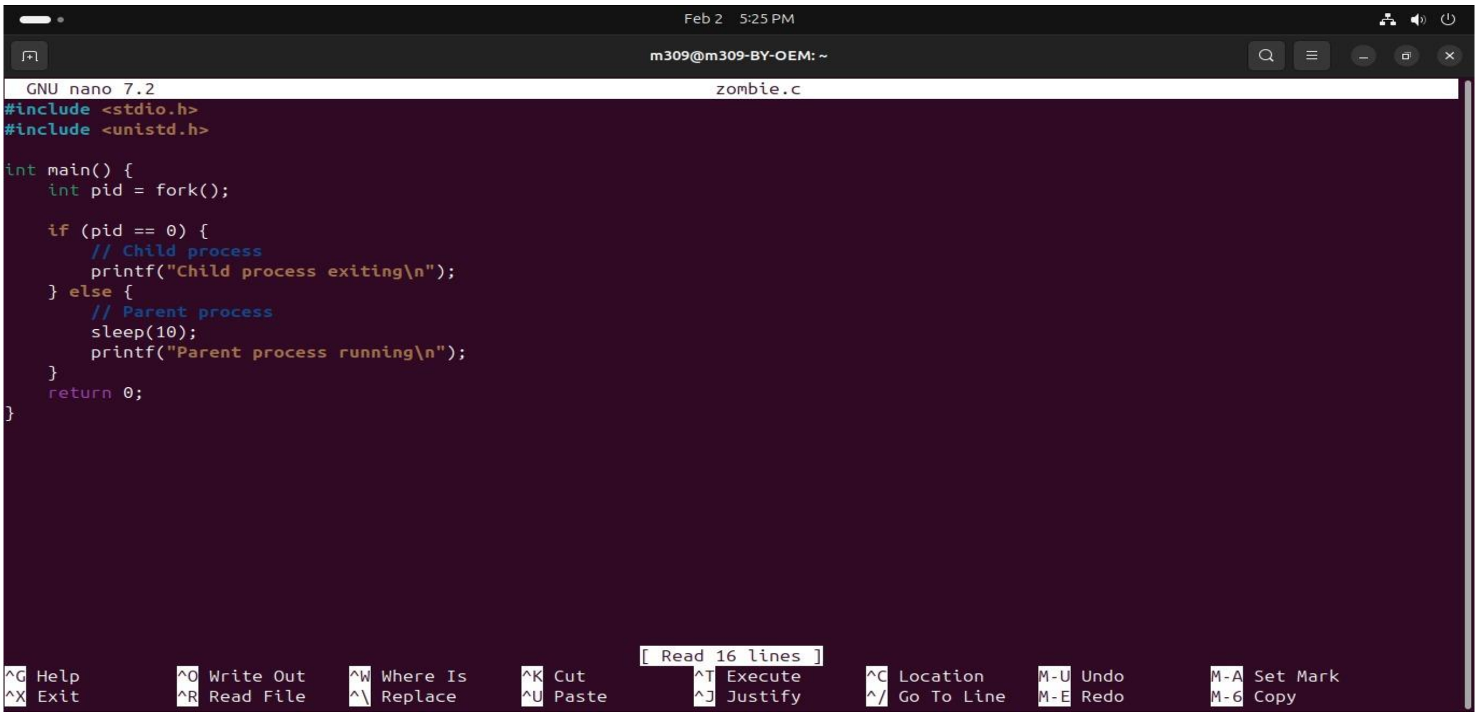


```
m309@m309-BY-OEM:~$ nano orphan.c
m309@m309-BY-OEM:~$ gcc orphan.c -o orphan
m309@m309-BY-OEM:~$ ./orphan
Parent Process ID: 9285
m309@m309-BY-OEM:~$ Child Process ID: 9286
New Parent Process ID: 2262
```

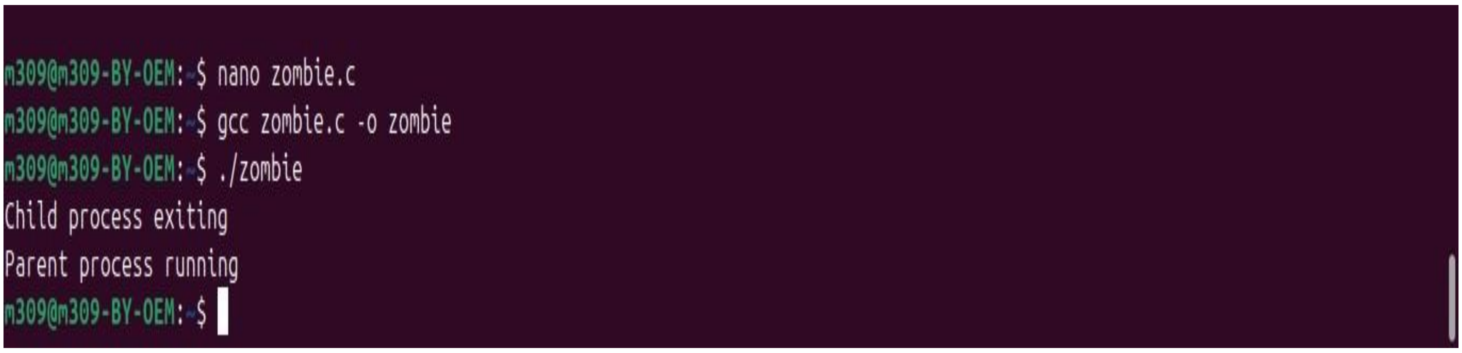
- Zombie Process



• Zombie Process
A zombie process is a child process that has completed execution. but still remains in the process table because its parent has not read its exist states

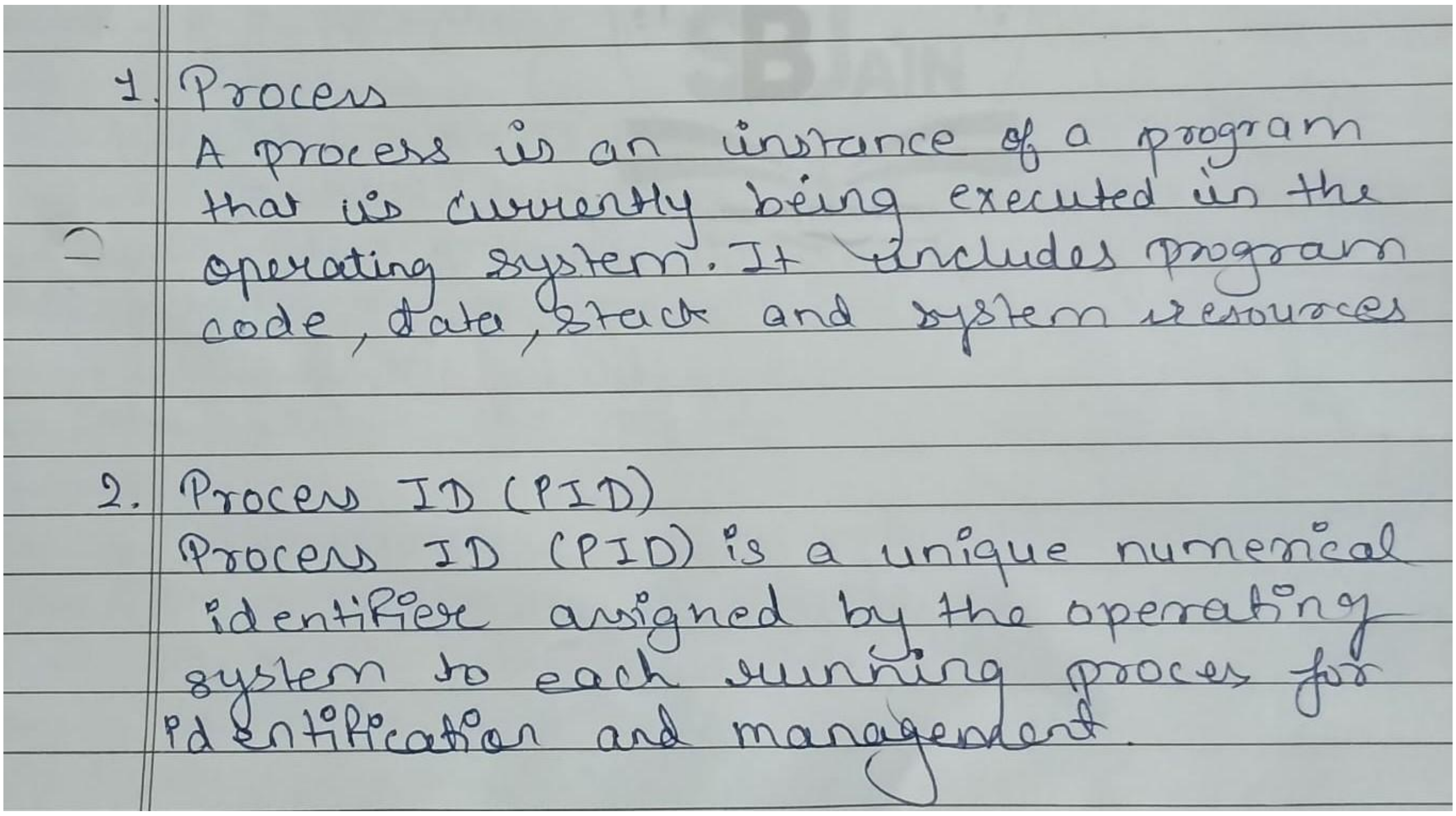


```
  GNU nano 7.2                                zombie.c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid = fork();

    if (pid == 0) {
        // Child process
        printf("Child process exiting\n");
    } else {
        // Parent process
        sleep(10);
        printf("Parent process running\n");
    }
    return 0;
}

                                  [ Read 16 lines ]
^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute     ^C Location    M-U Undo       M-A Set Mark
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify     ^/ Go To Line  M-E Redo       M-6 Copy
```



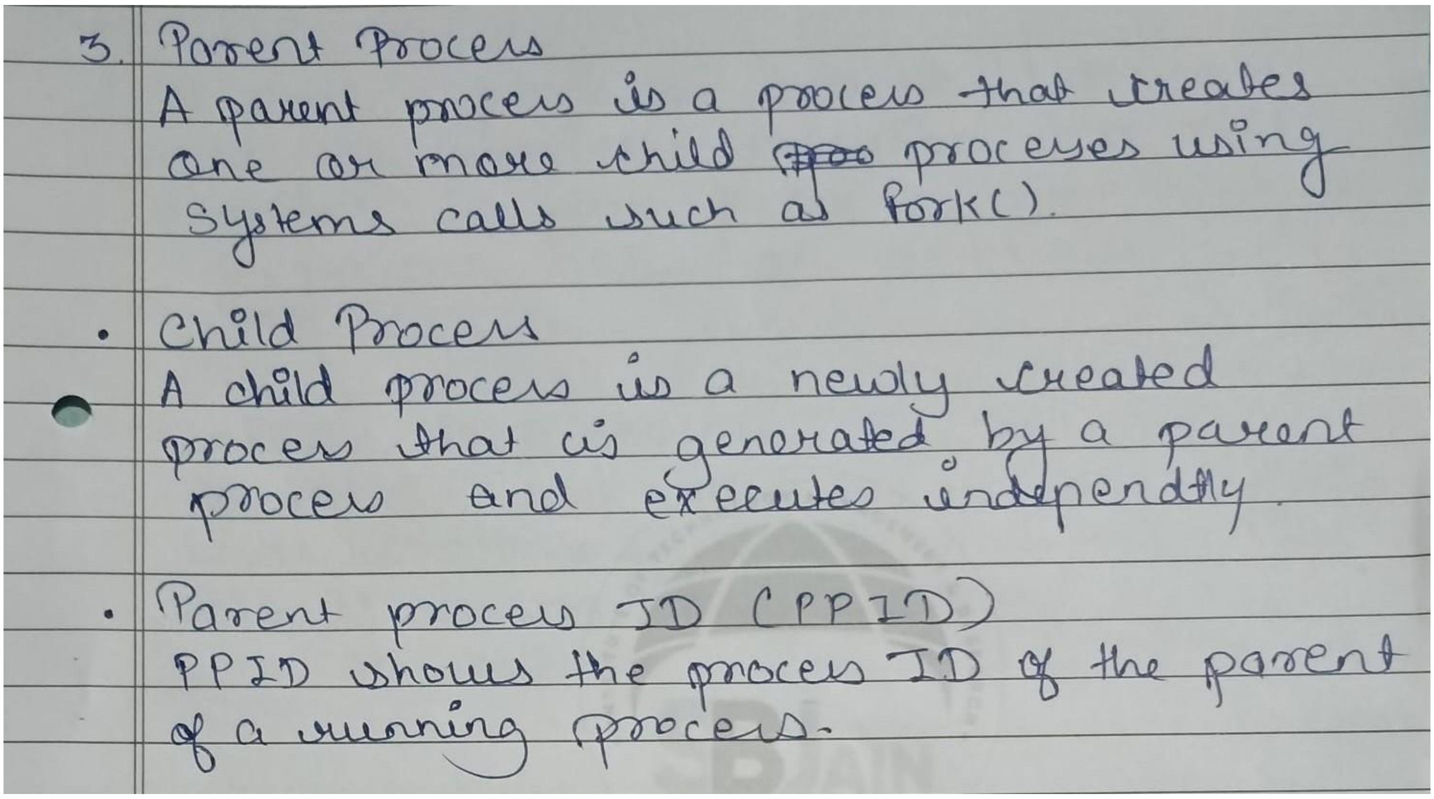
```
m309@m309-BY-OEM:~$ nano zombie.c
m309@m309-BY-OEM:~$ gcc zombie.c -o zombie
m309@m309-BY-OEM:~$ ./zombie
Child process exiting
Parent process running
m309@m309-BY-OEM:~$
```

3. Create the process using fork () system call.
- Child Process creation
- Parent process creation
- PPID and PID



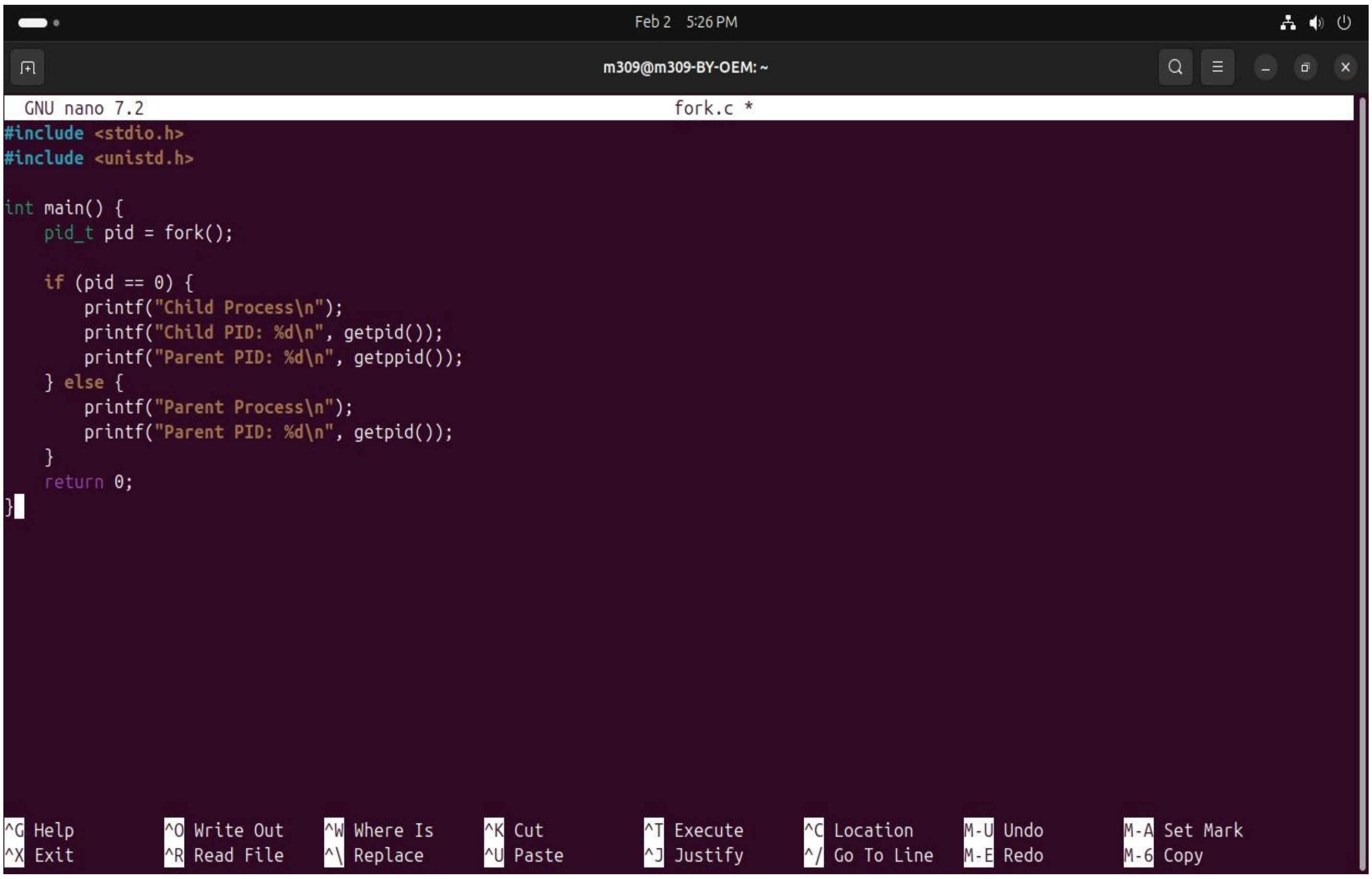
1. Procers
A process is an instance of a program that is currently being executed in the operating system. It includes program code, data, stack and system resources

2. Procers ID (PID)
Procers ID (PID) is a unique numerical identifier assigned by the operating system to each running proces for identification and management.


3. Parent Process
A parent process is a process that creates one or more child ~~proc~~ processes using systems calls such as fork().

• Child Process
A child process is a newly created process that is generated by a parent process and executes independtly.

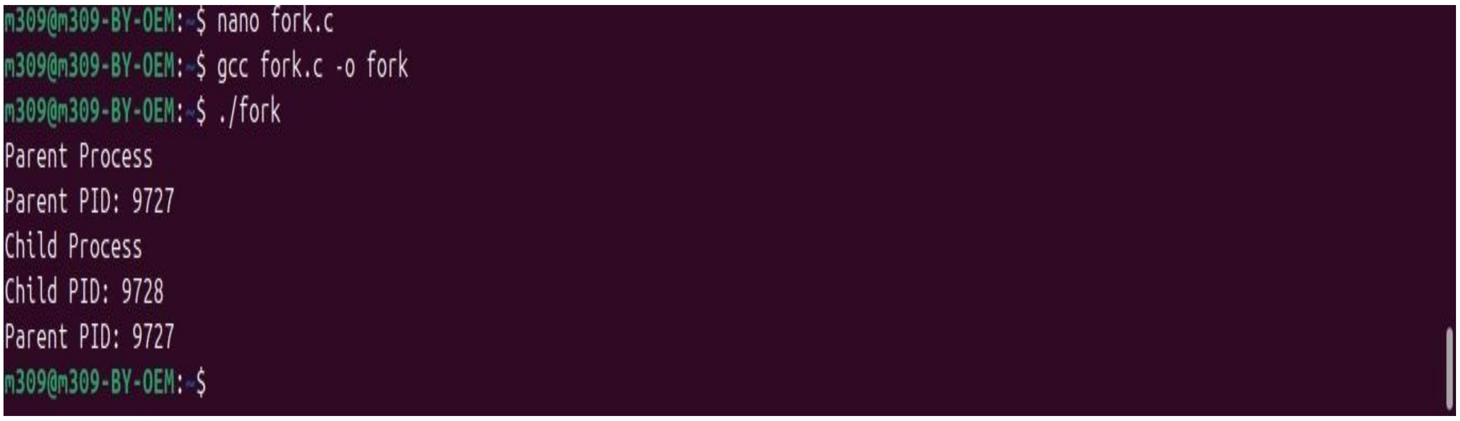
• Parent process ID (PPID)
PPID shows the process ID of the parent of a running process.


Feb 2 5:26 PM

m309@m309-BY-OEM: ~

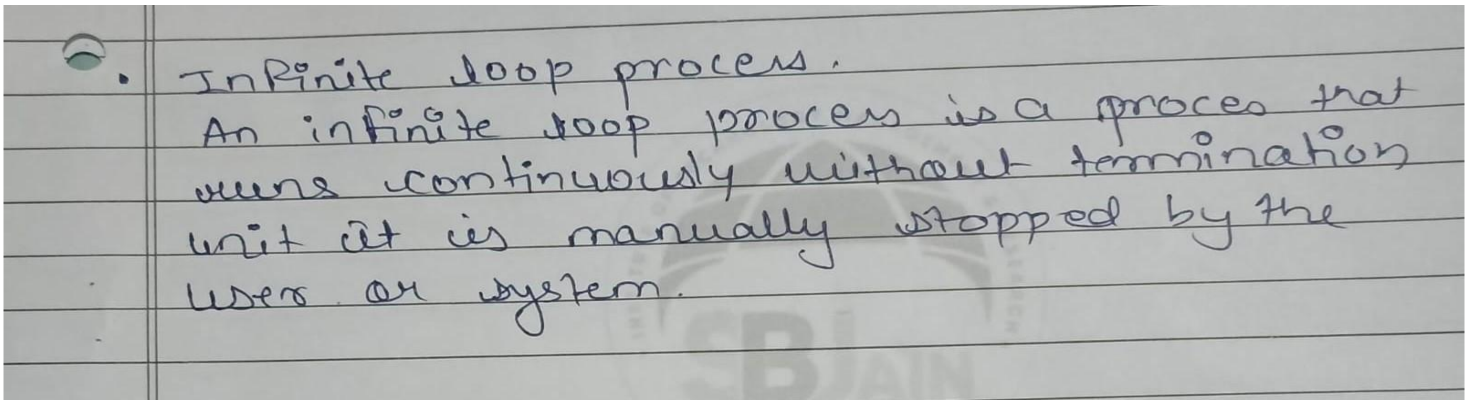
```
 GNU nano 7.2                                   fork.c *
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child Process\n");
        printf("Child PID: %d\n", getpid());
        printf("Parent PID: %d\n", getppid());
    } else {
        printf("Parent Process\n");
        printf("Parent PID: %d\n", getpid());
    }
    return 0;
}
```

```
^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute     ^C Location    M-U Undo       M-A Set Mark
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify     ^/ Go To Line  M-E Redo       M-6 Copy
```

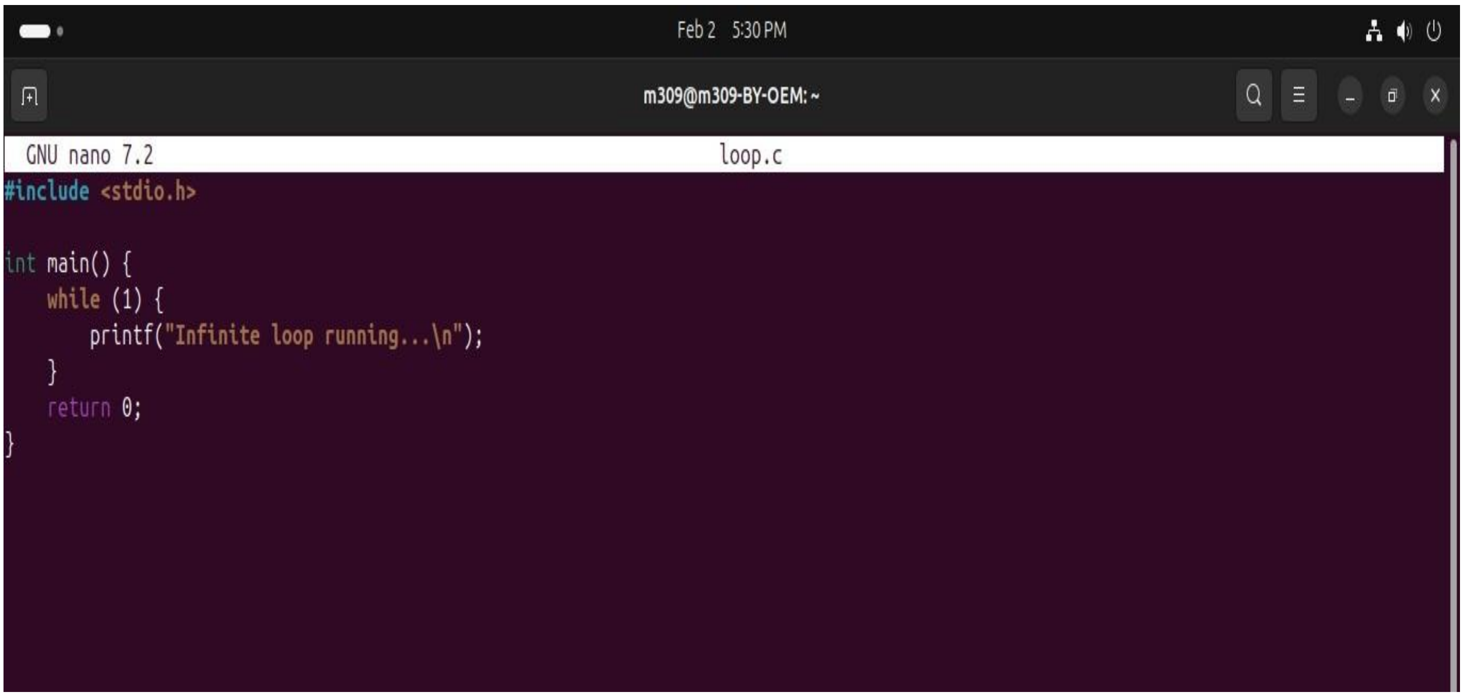


```
m309@m309-BY-OEM:~$ nano fork.c
m309@m309-BY-OEM:~$ gcc fork.c -o fork
m309@m309-BY-OEM:~$ ./fork
Parent Process
Parent PID: 9727
Child Process
Child PID: 9728
Parent PID: 9727
m309@m309-BY-OEM:~$
```

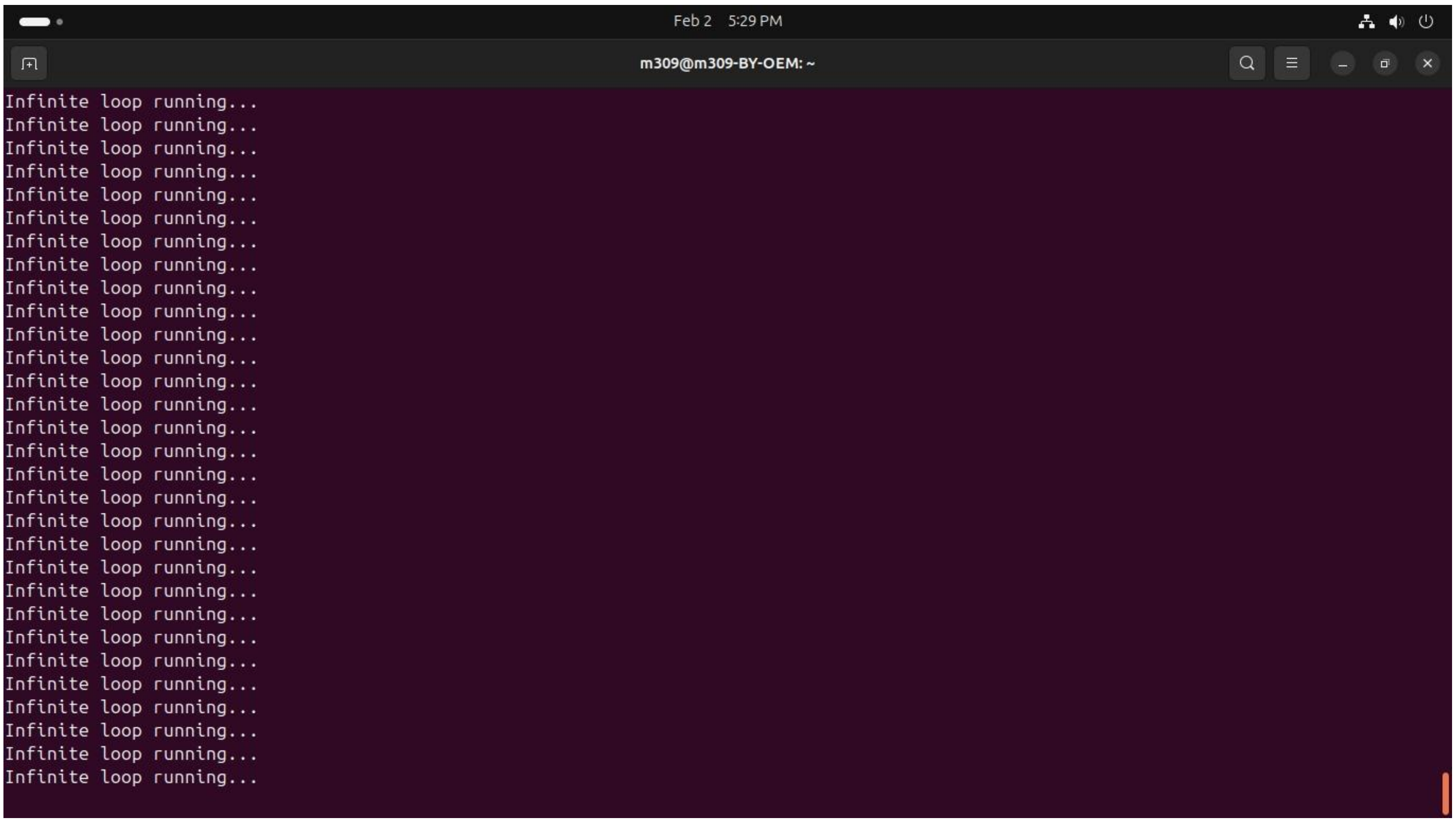
- Infinite Loop on Fork



• Infinite loop process.
An infinite loop process is a proces that runs continuously without termination unit it is manually stopped by the user or system.



```
GNU nano 7.2                                loop.c
#include <stdio.h>

int main() {
    while (1) {
        printf("Infinite loop running...\n");
    }
    return 0;
}
```



```
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
Infinite loop running...
```