LINUX PROCESS

STUDENT NAME:

Abdelrahman Mohamed Nagah

921230076

Mohamed Ashraf Mohamed Attia

921230114

Omar Mohamed Morsi Imam

921230095

Mazen Ahmed Kamel Ahmed

921230108

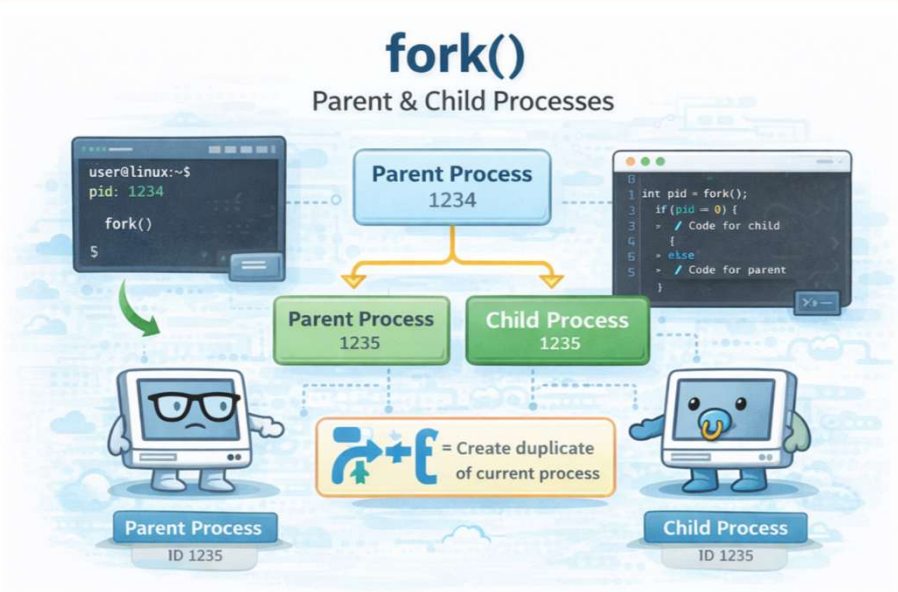
Saif Alden Ahmed Hesen

921230055

Linux Process

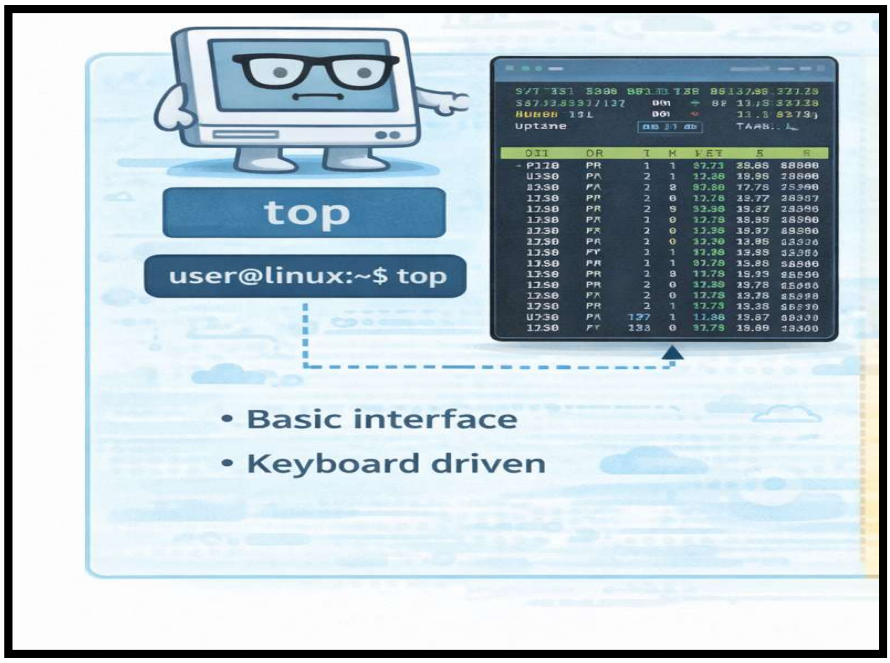
Linux is a multitasking operating system that allows multiple programs to run at the same time. To do this efficiently, Linux uses processes and threads, which are managed by the operating system kernel. Understanding how processes are created, scheduled, monitored, and terminated is an important topic in operating systems.

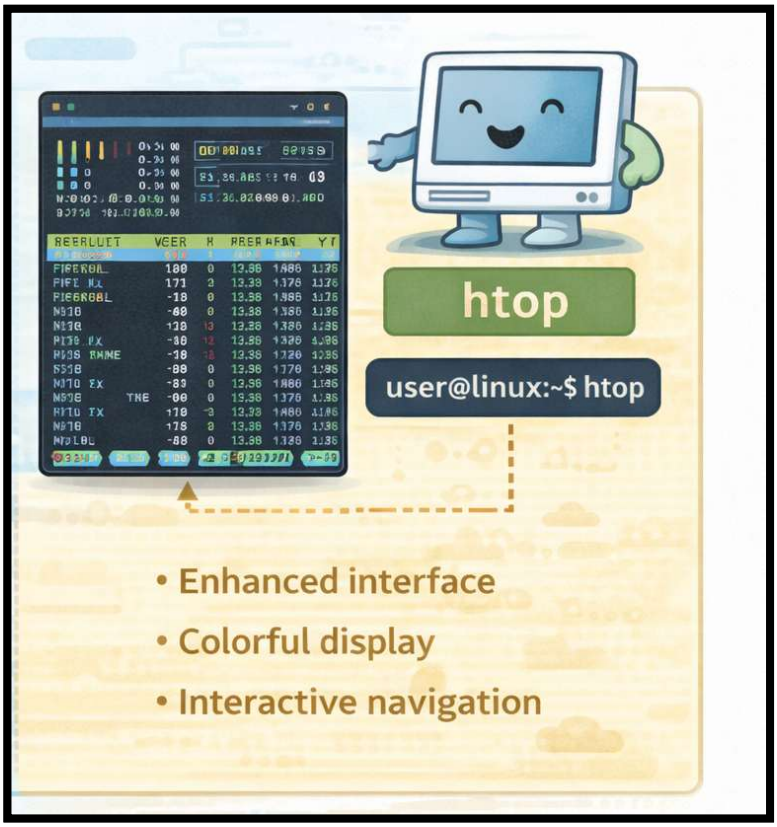
In this project, we study Linux process management through practical examples. We use the fork() system call to create parent and child processes and observe how they run and interact. Multithreading is also explored using POSIX threads (pthreads) to show how multiple threads can execute within the same process.



The project also examines CPU scheduling using the Linux Completely Fair Scheduler (CFS) and shows how changing process priorities with nice values affects CPU usage. Signal handling is covered by comparing SIGTERM and SIGKILL and understanding how processes are terminated. In addition, zombie and orphan processes are created to observe different process states

To monitor system behavior, Linux tools such as ps, top, and htop are used. A Bash script is implemented to automate process creation, monitoring, priority adjustment, and termination. Finally, a simple visualization is used to display process execution and CPU usage over time.





PID (Process ID) is a unique number assigned by the operating system to identify each running process. It allows the system to manage and control processes individually.
PPID (Parent Process ID) is the PID of the process that created the current process. It shows the parent–child relationship between processes in Linux.

CPU-Intensive Process

A CPU-intensive process was implemented to consume the maximum possible CPU resources in order to observe Linux scheduling behavior. The process runs in a continuous loop without performing any sleep operations, making it CPU-bound.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

void handle_sigterm(int sig) {
    printf("\nReceived SIGTERM. Cleaning up and exiting...\n");
    exit(0);
}

int main() {

pid_t pid = fork();

if (pid == 0) {

        signal(SIGTERM, handle_sigterm);
printf("\nChild\n");
printf("\nCPU Process is Looping Time...\n");
printf("\nCPU Process is Doesn't Stop...\n");
printf("\n PID = %d\n", getpid());
printf("\n PPID = %d\n", getppid());
printf("\n");
```

```
Parent

CPU Parent is Sleeping...

 PID = 5871

 PPID = 5099


Child

CPU Process is Looping Time...

CPU Process is Doesn't Stop...

 PID = 5872

 PPID = 5871

Terminated
nagah@nagah-VirtualBox:~/project_os$
```

Code Explanation

This program demonstrates process creation using fork() and handling the SIGTERM signal in Linux.
A signal handler function is defined to handle SIGTERM. When the signal is received, the process prints a message and exits gracefully.
In the main() function, fork() is used to create a child process. The child process registers the SIGTERM handler, prints its PID and PPID, and continues running.
When SIGTERM is sent to the child process, it terminates cleanly using the signal handler. This shows the difference between graceful termination (SIGTERM) and forceful termination (SIGKILL).

Slow/idle process

A slow (idle) process is a process that consumes very little CPU time. It spends most of its execution time in a sleep state using functions such as sleep().

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handle_sigterm(int sig) {
    printf("\nKill With SIGTERM.\nExiting Successfully...\n");
    _exit(0);
}

int main() {
    signal(SIGTERM, handle_sigterm);



pid_t pid = fork();

if (pid == 0) {
printf("\nSlow process started...\n");
printf("\n PID = %d\n", getpid());
printf("\n PPID = %d\n", getppid());
while (1) {
printf("\nSleeping...\n");
sleep(2);
    }
    }
    else {
        sleep(60);
    }

    return 0;
}
```

```
Slow process started...

 PID = 5884

 PPID = 5883

Sleeping...

Sleeping...

Sleeping...

Sleeping...

Sleeping...

Sleeping...

Kill With SIGTERM.
Exiting Successfully...
```

This program implements a slow process that consumes minimal CPU resources. After creating a child process using fork(), the child repeatedly enters a sleep state, causing it to remain mostly idle. A signal handler is used to handle SIGTERM, allowing the process to terminate gracefully when the signal is received. This demonstrates how Linux schedules low-CPU processes and handles graceful termination.

The parent process sleeps for a long period, allowing the child process to continue running. When sigterm is sent, the slow process exits cleanly using the signal handler.

Zombie Process

A zombie process is a process that has finished execution but still remains in the process table because its parent has not yet collected its exit status.
In this implementation, a parent process creates a child process using fork(). The child process terminates immediately, while the parent process continues running without calling wait(). As a result, the child becomes a zombie process. This behavior is observed using Linux monitoring tools, where the process state appears as Z.

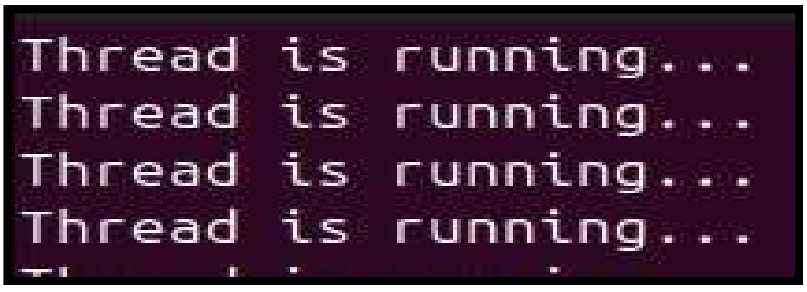
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

void handle_sigterm(int sig) {
    printf("\nReceived SIGTERM. Cleaning up and exiting...\n");
    exit(0);
}

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

    if (pid == 0) {

     signal(SIGTERM, handle_sigterm);

        printf("\nChild zombie...\n");
        exit(0);
    } else {
        printf("\nParent sleeping\n");
        printf("\n PID = %d\n", getpid());
        printf("\n PPID = %d\n", getppid());
        printf("\nchild becomes zombie\n");
        sleep(60);
    }

    return 0;
}
```

```
Parent sleeping

 PID = 5902

 PPID = 5099

child becomes zombie
Child zombie...
Terminated
nagah@nagah-VirtualBox:~/project_os$
```

4-Threaded Process Code (POSIX Threads)

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void* thread_function(void* arg) {
    int id = *(int*)arg;
    printf("Thread %d is running (TID: %lu)\n", id, pthread_self());

    while (1) {
        printf("Thread %d working...\n", id);
        sleep(1);
    }

    return NULL;
}

int main() {
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

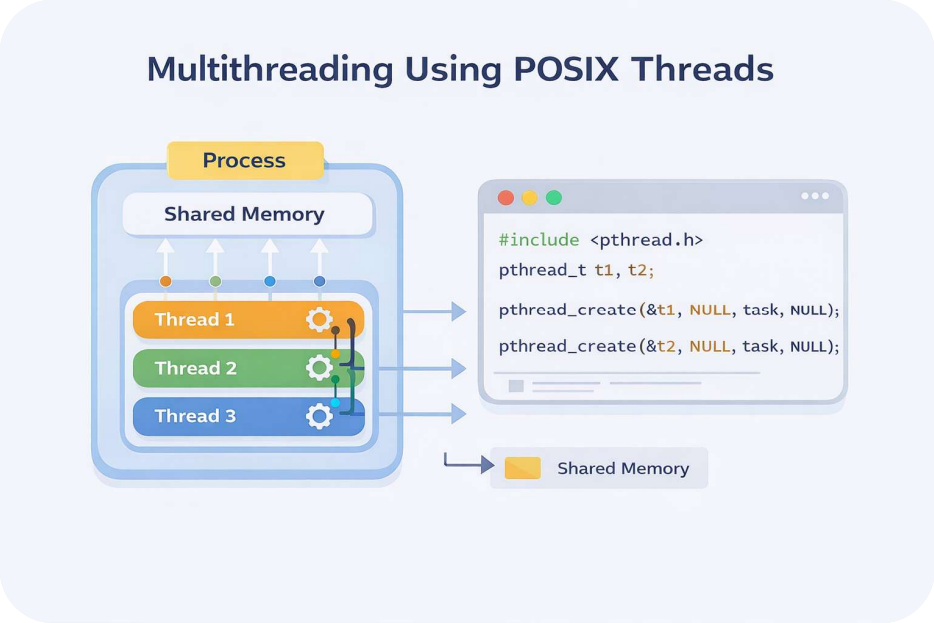
This program demonstrates multithreading using POSIX threads (pthreads). Two threads are created within a single process using pthread_create(). Each thread runs independently and repeatedly executes a task while sharing the same process memory. The threads are scheduled by the operating system.



Multithreading Using POSIX Threads

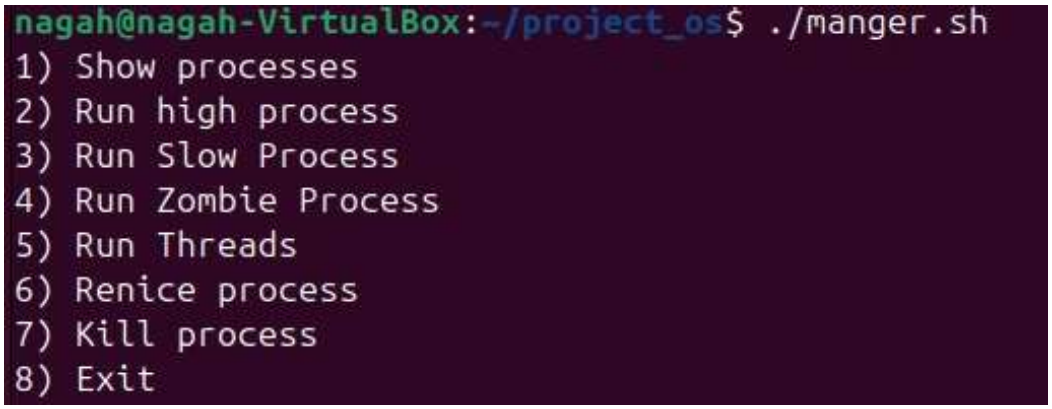
Multithreading was implemented using POSIX threads (pthreads). Multiple threads were created within a single process, each executing concurrently.

Threads share the same memory space and resources of the process, making them more lightweight than processes. This part of the project demonstrated parallel execution and thread scheduling in Linux.



MANAGER SHELL

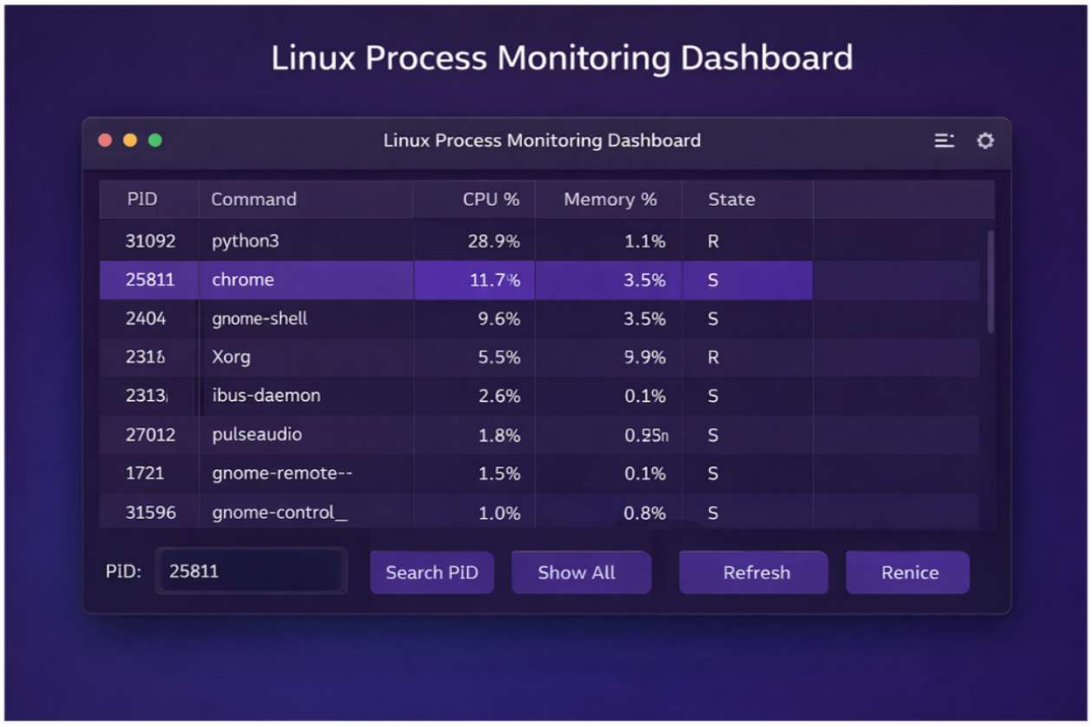
A manager shell was developed to automate and simplify the execution and management of project programs. The shell script acts as a control interface that allows compiling, running, and managing different process-related programs without manually invoking multiple command



GUI JAVA

A professional graphical user interface was developed using Java Swing to visualize Linux processes. The GUI displays key process information including PID, command name, CPU usage, memory usage, and process state.

The interface includes interactive buttons to refresh the process list, terminate processes using SIGTERM, change process priority using renice, and search for processes by PID. This GUI provides a visual alternative to command-line tools and enhances usability.



CONCLUSION

This project provided hands-on experience with Linux process management and scheduling. Through practical implementation, core concepts such as process creation, threading, signal handling, priority control, and monitoring were explored and analyzed. By combining command-line tools with a graphical interface, the project successfully bridges theoretical operating system concepts with real-world Linux behavior.