Project Description: Process Management Simulator

Overview

This project introduces **basic Unix process management** through a simple C program. Students will use `fork()`, `execvp()`, and `waitpid()` to create multiple child processes, execute external commands, and synchronize with a parent process.

The project emphasizes **understanding process creation, execution replacement, and termination status**, rather than advanced inter-process communication or scheduling.

Project Objectives

By completing this project, students will be able to:

- Understand and implement **process creation** using `fork()`.
- Use **exec-family functions** (e.g., `execvp()`) to replace a child process with another program.
- Manage **process synchronization** using `waitpid()`.
- Interpret **process termination results**, including:
  - normal exit with an exit code
  - termination by a signal
- Understand the difference between **process creation order** and **process completion order**.

Task Description

1. Parent Process Creation

- Write a C program where the main process (parent) creates **at least 15 unique child processes** using `fork()` inside a loop. (A unique process means different Linux commands.)
- The parent process must **record the PID of each child** in an array in the order the children are created.
- The parent should print its own PID at the start of the program.

2. Child Process Execution

- Each child process must:
  - Print its **child index**, **PID**, and the **command it will execute**.
  - Use an `exec` function (such as `execvp`) to execute a simple command (e.g., `ls`, `date`, `echo`, etc.).
- At least one child process must execute:
- `echo "Hello <Your Name>"`

- At least two child process must intentionally run an **invalid command**, causing `execvp()` to fail and the child to exit with a non-zero exit code (e.g., `127`).
- At least two child process must intentionally **terminate abnormally** using `abort()` to demonstrate signal-based termination.

3. Process Synchronization (Controlled Execution Order)

- After creating all child processes, the parent process must wait for them to finish using:
- `waitpid(childPids[i], &status, 0);`
- The parent must wait for child processes **in creation order**, not completion order.
- For each child process, the parent must report:
    - Whether the child exited normally or was terminated by a signal.
    - The exit code (if exited normally).
    - The signal number (if terminated by a signal).

4. Reporting and Summary

- After all child processes have terminated, the parent process must print a **summary**, including:
    - Number of child processes that exited normally with exit code `0`
    - Number of child processes that exited normally with a non-zero exit code
    - Number of child processes that were terminated by a signal
- Output should clearly show how child processes were created, executed, and terminated.

5. Makefile

- Provide a simple **Makefile** that compiles the program using `gcc`.
- The Makefile should support a command such as:
- `make`

Project Report (1–2 pages)

Submit a short report that combines theory with your implementation. The report must include:

1. **Introduction**
    - Briefly explain what process management is and why it is important in operating systems.
2. **Implementation Summary**
    - Describe the overall structure of your program.
    - Explain how `fork()`, `execvp()`, and `waitpid()` are used.
3. **Results and Observations**
    - Explain how child processes were created and how their PIDs were managed.
    - Describe the difference between:
        - process creation order
        - process termination order

    - Explain how the parent determines whether a child exited normally or was terminated by a signal.
4. **Conclusion**
    - Summarize what you learned about Unix process management.

Implementation Hints

1. Setup

- Start with a simple C program structure.
- Include the required headers:
- `#include <unistd.h>`
- `#include <sys/wait.h>`
- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <signal.h>`

2. Process Creation

- Use `fork()` in a loop.
- Correctly handle the return value of `fork()`:
    - `0` → child process
    - `>0` → parent process (child PID)
    - `<0` → error

3. Executing Commands

- In each child process, use `execvp()` to run a command.
- Remember: `execvp()` only returns if an error occurs.

4. Waiting and Status Checking

- Use `waitpid()` in the parent to wait for each child.
- Use macros such as:
    - `WIFEXITED(status)`
    - `WEXITSTATUS(status)`
    - `WIFSIGNALED(status)`
    - `WTERMSIG(status)`

5. Error Handling

- Include basic error handling for `fork()`, `execvp()`, and `waitpid()`.

Submission Requirements

1. **Source Code**
    - C source file
    - Makefile

    - Clear comments explaining major steps
      (Submit to GitHub)
2. **Execution Results**
    - Screenshots showing program output
      (Submit to GitHub)
3. **Project Report**
    - 1–2 page PDF or DOC file
      (Submit to GitHub)
4. **GitHub Repository Link**
    - Submit the repository link on Canvas