**SimpleScheduler Assignment**

Group Members:

**Asa(2022113),**

**Sargun(2022450)**

**GitHub Repository:** https://github.com/sargun4/OS/tree/main/A3

**Project Overview:**

Manages and schedules processes.

Uses custom signals (SIGUSR2, SIGALRM, SIGUSR1) for process control.

Initializes shared memory for process queues.

Suspends and resumes running processes based on signals.

Updates execution and waiting times for processes.

Handles CPU scheduling with a given time slice (TSLICE) and CPU count (NCPU).

**\*\*SimpleScheduler.c\*\***

**1. `initialize\_shared\_queues()`:**

**-** Opens and initializes shared memory for process queues.

- It creates or opens a shared memory segment, sets its size, and maps it to the process's address space.

- Sets up a mutex for process queue synchronization.

**2. `cleanup()`:**

- Unmaps the shared memory and closes the shared memory file descriptor.

- Destroys the mutex.

- Unlinks the shared memory**.**

**3. `custom\_signal\_handler(int signo)`:**

- Handles custom signals (SIGUSR2, SIGALRM, SIGUSR1):

- SIGUSR2: Suspends running processes, moves them to the waiting queue, and updates their execution and waiting times.

- SIGALRM: Sends SIGUSR2 signal to initiate the scheduling cycle.

- SIGUSR1: Lists information about waiting processes.

**4. `main(int argc, char const \*argv[])`:**

- Registers signal handlers for custom signals (SIGUSR2, SIGALRM, SIGUSR1).

- Sets up a signal set for waiting.

- Parses the command-line arguments for the number of CPUs (NCPU) and time slice (TSLICE).

- Calls `initialize\_shared\_queues()` to set up shared memory.

- Creates sample processes and schedules them using the `run\_executable()` function.

- Enters a loop waiting for signals and checks for the completion of all processes.

- Calls `cleanup()` to release shared memory and other resources**.**

**\*\*SimpleShell.c\*\***

**5. `read\_input(char \*input)`:**

**-** Displays the command prompt and reads user input into the input buffer.

**6. `read\_from\_file(char \*input, FILE \*fptr)`:**

- Reads input from a file, stores it in the input buffer, and handles newline characters**.**

**7. `execute\_command(char \*command, int background)`:**

- Executes a command specified by `command`.

- Handles both foreground and background execution:

- In foreground execution:

- Captures the command's output using a pipe.

- Forks a child process to execute the command.

- Redirects the child process's output to the pipe.

- Records information about the command in the command history.

- In background execution:

- Forks a child process to execute the command.

- Records information about the background process**.**

**8. `show\_history()`:**

- Displays the command history, including the command number, command text, timestamp, PID, and execution duration.

**9. `check\_background\_processes()`:**

- Checks the status of background processes and updates their `is\_finished` flag accordingly.

- Displays information about running or completed background processes**.**

**10. `execute\_file()`:**

- Executes commands from a file specified by the user.

- Handles background execution for commands ending with "&".

- Allows the user to execute "exit" to display the command history.

**11. `main(int argc, char const \*argv[])`:**

- Parses command-line arguments for the number of CPUs (NCPU) and time slice (TSLICE).

- Spawns the SimpleScheduler process and sets up shared memory.

- Reads user input or input from a file.

- Executes and manages foreground and background commands.

- Provides options to view command history and background process status.

- Handles exiting the shell and cleaning up resources.

**12. `run\_executable(char name[], char priority)`:**

- Not currently used, appears to be a placeholder for executing processes in the scheduler. The process insertion into queues is present.

**Contributions:**

Asa:

Sargun:

**Conclusion**

In summary, our SimpleScheduler is designed for efficient process scheduling and management. It leverages shared memory and custom signals to ensure that processes are executed fairly and that resources are optimally utilized.