OPERATING SYSTEMS – LAB 2

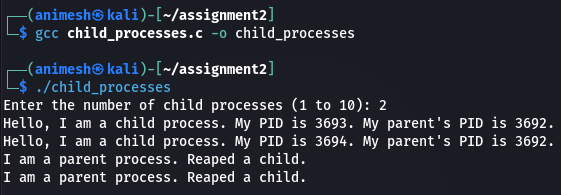
Animesh Mishra 27 January 2024

2210110161

Q1)

This program prompts the user to enter the number of child processes they want to create (between 1 and 10). It then creates the specified number of child processes using the fork system call. Each child process prints its PID and its parent's PID, along with the message "Hello, I am a child process." After printing the message, each child process exits.

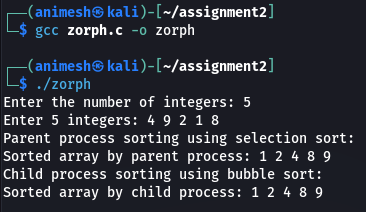
The parent process waits for each child to complete using the wait system call and then prints the message



Q2)

The program sorts integers using a parent-child process model. It prompts the user for integers, forks a child process, and sorts the integers in the parent process. The child process does not participate in sorting but prevents orphaning. After sorting, the parent process waits for the child to terminate, ensuring synchronization and preventing zombies. Finally, it completes execution.

After sorting is complete in the parent process, it waits for the child process to terminate using the wait() system call. Once the child process terminates, the parent process concludes its execution. This entire process showcases the coordination between parent and child processes.

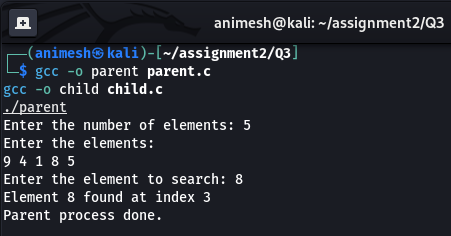


Q3)

In the child process, the sorted array is passed as command-line arguments to the execvp() function, which executes another program (binary\_search) to perform binary search.

In the parent process, the integer array is sorted using any sorting algorithm before waiting for the child process to finish.

After the child process completes its execution, the main program frees the dynamically allocated memory before exiting. The child process, invoked using the exec system call, effectively loads and executes a new program, which in this case is the binary search program.

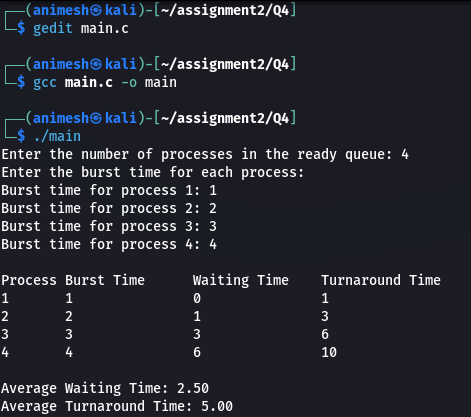


Q4)

The program begins by accepting the number of processes and their corresponding burst times, representing the time each process requires to complete its execution. It then proceeds to calculate the waiting time and turnaround time for each process based on the FCFS principle.

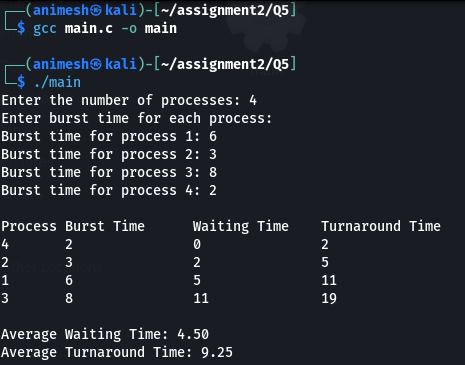
The waiting time for each process is determined as the cumulative sum of the burst times of all preceding processes, reflecting the amount of time a process waits in the ready queue before execution.

Once the waiting and turnaround times are calculated for all processes, the program computes the average waiting and turnaround times, providing insights into the efficiency and performance of the FCFS scheduling algorithm.



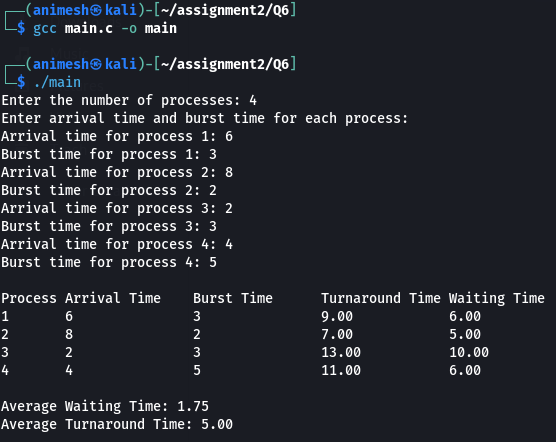
Q5)

This sorting ensures that the process with the shortest burst time is scheduled first, adhering to the SJF algorithm's principle. Through iterations and loops, the program calculates waiting times for each process by maintaining the cumulative burst times of previously executed processes. Similarly, turnaround times reflecting the total time from arrival to completion for each process are determined. Following the completion of calculations, the program computes average waiting and turnaround times offering insights into the efficiency of the SJF algorithm in managing CPU resources. Finally, the program displays the calculated averages, allowing users to evaluate the algorithm's performance and effectiveness in optimizing process scheduling.



Q6)

During CPU scheduling simulation, the program continually selects the process with the shortest remaining time for execution. If a new process arrives with a shorter burst time than the currently executing process, a context switch occurs, and the new process is executed. Following process execution, the waiting time for each process is computed as the difference between its turnaround time and burst time. Turnaround time is calculated as the total time taken by a process from arrival to completion.



Q7)

The program starts by asking the user how many tasks there are and how long each task can run before the CPU switches to another one. Each task is like a small job waiting to be completed, and the program keeps track of their details, like how long each task needs to finish. It then runs through all the tasks, giving each one a turn to use the CPU for a set amount of time. If a task doesn't finish in its turn, it waits until it's its turn again. The program also keeps track of how long each task had to wait and how long it took to finish. After running through all the tasks, the program calculates the average time each task had to wait and how long it took for tasks to complete. This helps users see how well the Round-Robin algorithm manages tasks and uses the CPU.

