Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavrepalanchowk

A Lab Report 2 and 3

on

"Understanding fork( ) and process creation"

[Code No: COMP307]

Submitted by:

Salina Nakarmi

CS III-I (60)

Roll-No: 34

Submitted to:

Ms. Rabina Shrestha

Department of Computer Science and Engineering

Submission Date:

2026/01/04

Objective

To study how fork() creates new processes and understand why multiple fork() calls increase the number of processes.

Theory: Understanding fork()

The fork() system call is fundamental to process creation in Unix/Linux systems. It creates a new process by duplicating the calling process.

Key Characteristics of fork():

- Takes no arguments and returns a process ID
- Creates a child process that is an exact copy of the parent
- Both parent and child execute the next instruction after fork()
- Returns different values to parent and child:
    - **Negative value:** Process creation failed
    - **Zero (0):** Returned to the newly created child process
    - **Positive value:** Child's PID returned to the parent process

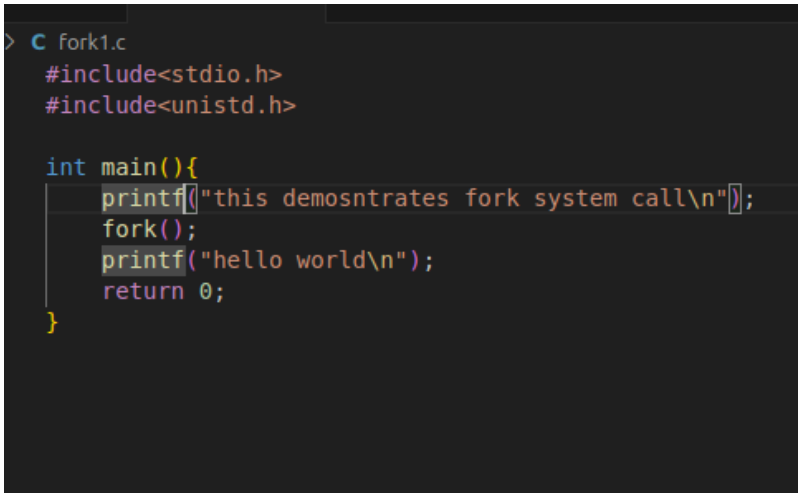
When fork() executes successfully, Unix creates two identical copies of the address space; one for the parent and one for the child. Both processes have separate address spaces and continue execution from the statement following the fork() call.

Lab_Report_2

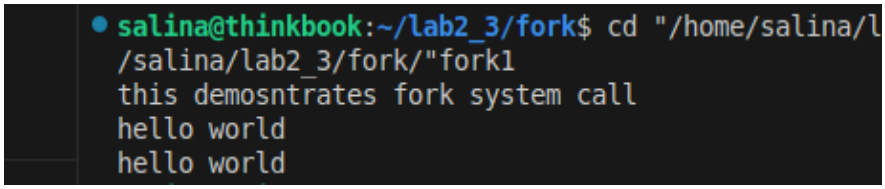
Program 1: Basic Process Creation

Objective: Demonstrate basic fork() usage and understand process duplication.

Source Code:



Output:



Analysis:

In this program, the fork() system call divides the parent process into two identical processes. The parent process prints "This demonstrates the fork" first, then calls fork(). At this point, a child process is created. Both the parent and child processes continue execution from the line after fork(), which is why "Hello world" is printed twice—once by the parent and once by the child.

Part A: Two fork() Calls

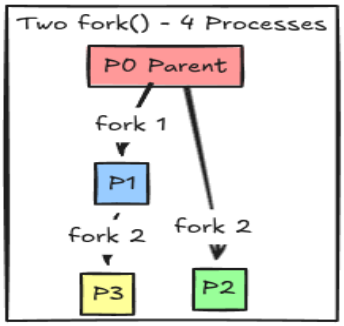
Modified Source Code:

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

int main()
{
    printf("This demonstrates the fork\n");
    fork();
    fork();
    printf("Hello world\n");
    return 0;
}
```

Output:

```
salina@thinkbook:~/lab2_3/fork$ cd "/home/sali
me/salina/lab2_3/fork/"fork1a
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
```

Process Tree Diagram:



Analysis:

With two fork() calls, the number of processes increases exponentially. Here's what happens:

1. **First fork():** The original process (P0) creates one child (P1). Now there are 2 processes.
2. **Second fork():** Both P0 and P1 execute this fork(), each creating their own child (P2 and P3). Now there are 4 processes.

Part B: Three fork() Calls

Modified Source Code:

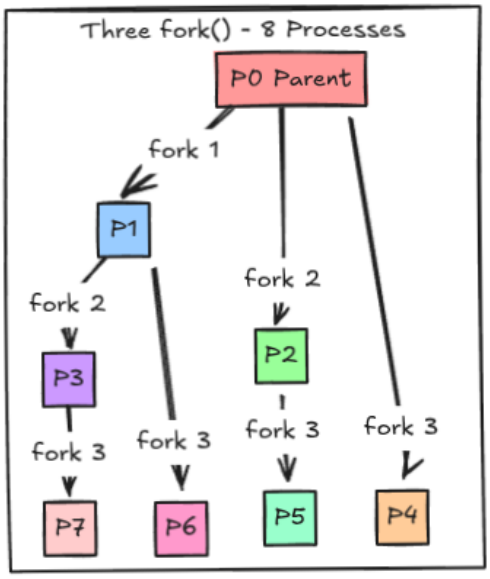
> C fork1a.c

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

int main()
{
    printf("This demonstrates the fork\n");
    fork();
    fork();
    fork();
    printf("Hello world\n");
    return 0;
}
```

Output:

```
salina@thinkbook:~/lab2_3/fork$ cd "/home/salina/lab
me/salina/lab2_3/fork/"fork1a
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

Process Tree Diagram:



Analysis:

With three fork() calls, we observe an exponential increase in the number of processes:

1. **First fork():** 1 parent (P0) creates 1 child (P1) → 2 processes total
2. **Second fork():** 2 processes each create a child → 4 processes total (P0, P1, P2, P3)
3. **Third fork():** 4 processes each create a child → 8 processes total (P0-P7)

Formula: Total processes = **2^n**, where n = number of fork() calls**.**
Therefore: 2^3 = 8 processes, resulting in 8 "Hello world" prints.

Why does the number increase exponentially?

Each fork() call is executed by ALL existing processes at that point. When the first fork() creates 2 processes, both of those processes execute the second fork(), creating 4 total. Then all 4 processes execute the third fork(), creating 8 total. This cascading effect causes exponential growth.

Program 2: Understanding fork() with Process IDs

Objective: Demonstrate how to distinguish between parent and child processes using PIDs.

Source Code:

process.c

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

int main() {
    int pid;
    printf("I am the parent process with ID %d\n", getpid());
    printf("Here I am before the use of forking\n");

    pid = fork();

    printf("Here I am just after forking\n");

    if (pid == 0)
        printf("I am a child process with ID %d\n", getpid());
    else
        printf("I am the parent process with PID %d\n", getpid());

    return 0;
}
```

Output:

```
salina@thinkbook:~/lab2_3/fork$ cd "/home/salina/lab2_3/" && gcc process
s
I am the parent process with ID 5788
Here I am before the use of forking
Here I am just after forking
I am the parent process with PID 5788
Here I am just after forking
I am a child process with ID 5789
```

Question 1: Explain the difference between pid == 0 and pid > 0. Which one corresponds to the child and the parent?

→ The variable pid stores the return value of fork():

- pid == 0: This condition is TRUE in the child process. When fork() successfully creates a child, it returns 0 to the child process. Therefore, only the child process executes the code inside this if block.

- pid > 0: This condition is TRUE in the parent process. When fork() creates a child, it returns the child's process ID (a positive integer) to the parent. The parent uses this value to identify which child it created.

  Summary:

- pid == 0 → Child process
- pid > 0 → Parent process (the value is the child's PID)
- pid < 0 → Fork failed (error condition)

Question 2: Why do the parent and child processes print their own PID differently?

→ Although the child is a copy of the parent, each process has its own unique Process ID (PID) assigned by the operating system. When we call getpid():

- In the parent process: getpid() returns the parent's PID
- In the child process: getpid() returns the child's PID (different from parent)

Question 3: Explain why the output order is parent lines first, child lines after. Can the order change? Why?

→ In many cases, the parent process executes first because:

1. The parent process was already running when fork() was called
2. The parent may complete its execution before the child is fully scheduled

3. The operating system may give priority to the parent process

However, the order CAN change because:

- Process scheduling is non-deterministic: The OS scheduler decides which process runs when, based on scheduling algorithms, system load, and priorities.
- No guaranteed execution order: There is no guarantee that the parent will always execute before the child or vice versa.
- Race condition: Both processes are ready to run, and whichever gets CPU time first will execute first.

In practice: We might see outputs interleaved differently in different runs. For example:

- Sometimes parent completes all prints before child
- Sometimes child prints before parent
- Sometimes outputs are interleaved

This demonstrates the concurrent nature of process execution in multitasking operating systems.

Lab 3: Process Scheduling Algorithms

Objective

To understand and implement various CPU scheduling algorithms including SJF, SRTF, and Round Robin.

Question 1: Short Notes on Preemptive and Non-Preemptive Scheduling

Non-Preemptive Scheduling

In non-preemptive scheduling, once a process starts executing, it continues until it completes its execution or voluntarily yields the CPU (e.g., for I/O operations). The CPU cannot be taken away from the process.

Characteristics:

- Simple to implement
- Low overhead as no context switching during execution
- Processes run to completion once started
- May cause poor response time for short processes if a long process is executing

Examples: FCFS (First Come First Serve), SJF (Shortest Job First - non-preemptive version)

Advantages:

- No context switching overhead during execution
- Predictable and simple scheduling decisions

Disadvantages:

- Can lead to convoy effect (short processes waiting for long ones)
- Poor average waiting time
- Not suitable for time-sharing systems

Preemptive Scheduling

In preemptive scheduling, the CPU can be taken away from a running process before it completes execution. The scheduler can interrupt a running process and allocate the CPU to another process based on priority, time quantum, or other criteria.

Characteristics:

- More complex to implement
- Higher overhead due to frequent context switching
- Better response time for short processes
- Prevents monopolization of CPU by long processes

Examples: SRTF (Shortest Remaining Time First), Round Robin, Priority Scheduling (preemptive)

Advantages:

- Better response time
- Fair CPU allocation among processes
- Prevents starvation of short processes
- Suitable for time-sharing and real-time systems

Disadvantages:

- Context switching overhead
- More complex implementation
- Requires hardware support (timer interrupts)

Question 2: SJF (Shortest Job First) Scheduling Algorithm

SJF is a non-preemptive scheduling algorithm that selects the process with the smallest burst time for execution next. It aims to minimize average waiting time.

Algorithm Steps:

1. Sort all processes by their burst time in ascending order
2. The first process (shortest burst time) executes first
3. Calculate waiting time: For each process, waiting time = sum of burst times of all previous

processes

4. Calculate turnaround time: Turnaround Time = Waiting Time + Burst Time
5. Compute averages

Characteristics:

- Optimal for minimizing average waiting time
- Non-preemptive (process runs to completion)
- Can cause starvation for longer processes
- Requires knowledge of burst time in advance (often not practical)

Source Code:

```c
#include <stdio.h>

int main() {
    int n, i, j, temp;
    int bt[20], wt[20], tat[20], p[20];
    int total_wt = 0, total_tat = 0;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    // Input burst times
    for (i = 0; i < n; i++) {
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1; // Store process numbers
    }

    // Sort processes by burst time (SJF)
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (bt[i] > bt[j]) {
                // Swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap process numbers
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    // Waiting time for first process is 0
    wt[0] = 0;

    // Calculate waiting time
    for (i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    // Calculate turnaround time
    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
        total_wt += wt[i];
        total_tat += tat[i];
    }
```

```
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time = %.2f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", (float)total_tat / n);

    // Gantt Chart
    printf("\nGantt Chart:\n");
    printf(" ");
    for (i = 0; i < n; i++) {
        printf("-------");
    }
    printf("\n|");
    for (i = 0; i < n; i++) {
        printf("  P%d  |", p[i]);
    }
    printf("\n ");
    for (i = 0; i < n; i++) {
        printf("-------");
    }
    printf("\n0");
    for (i = 0; i < n; i++) {
        printf("      %d", wt[i] + bt[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
salina@thinkbook:~/lab2_3$ cd "/home/salina/lab2_3/scheduling/" && gcc sjf.
Enter number of processes: 3
Enter Burst Time for Process 1: 3
Enter Burst Time for Process 2: 2
Enter Burst Time for Process 3: 5

Process Burst Time      Waiting Time    Turnaround Time
P2              2               0               2
P1              3               2               5
P3              5               5               10

Average Waiting Time = 2.33
Average Turnaround Time = 5.67

Gantt Chart:
 ---------------------
|  P2  |  P1  |  P3  |
 ---------------------
0       2       5       10
```

Question 3: SRTF (Shortest Remaining Time First) Scheduling Algorithm

SRTF is the preemptive version of SJF. At any given time, the process with the shortest remaining burst time is executed. If a new process arrives with a shorter burst time than the remaining time of the current process, the current process is preempted.

Algorithm Steps:

1. At each time unit, check for newly arrived processes
2. Select the process with the shortest remaining time
3. If a new process arrives with shorter remaining time, preempt current process

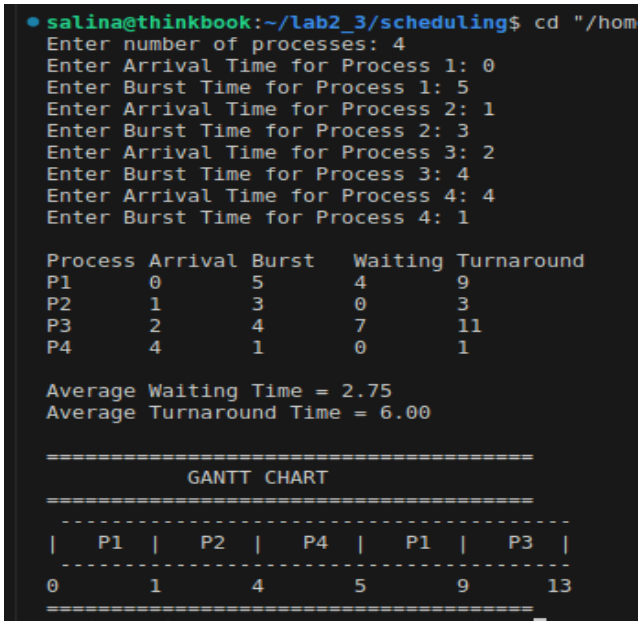
4. Continue until all processes complete
5. Calculate waiting time and turnaround time

Characteristics:

- Preemptive scheduling
- Optimal for minimizing average waiting time
- Requires frequent context switching
- Can cause starvation for longer processes

Source Code:

Output:



Question 4: Round Robin Scheduling Algorithm

Round Robin is a preemptive scheduling algorithm designed for time-sharing systems. Each process is assigned a fixed time quantum. The CPU switches between processes in a circular queue fashion.

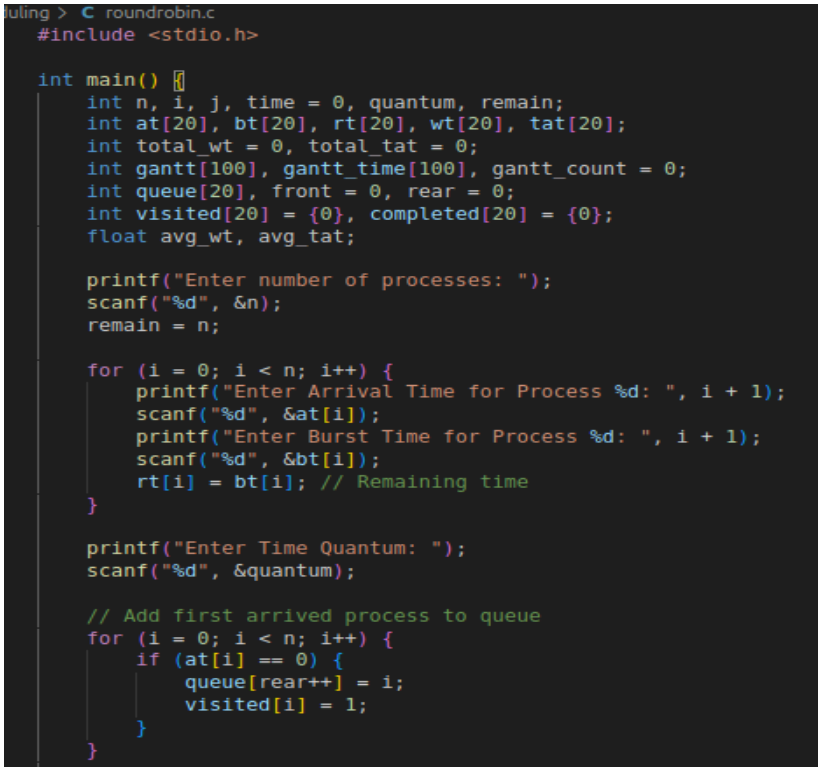
Algorithm Steps:

1. Set a time quantum (time slice)
2. Place all processes in a circular queue
3. Execute each process for the time quantum
4. If process completes within quantum, remove from queue
5. If process doesn't complete, move to end of queue
6. Continue until all processes complete

Characteristics:

- Preemptive scheduling
- Fair allocation of CPU time
- No starvation (every process gets CPU time)
- Performance depends on time quantum selection
- Context switching overhead

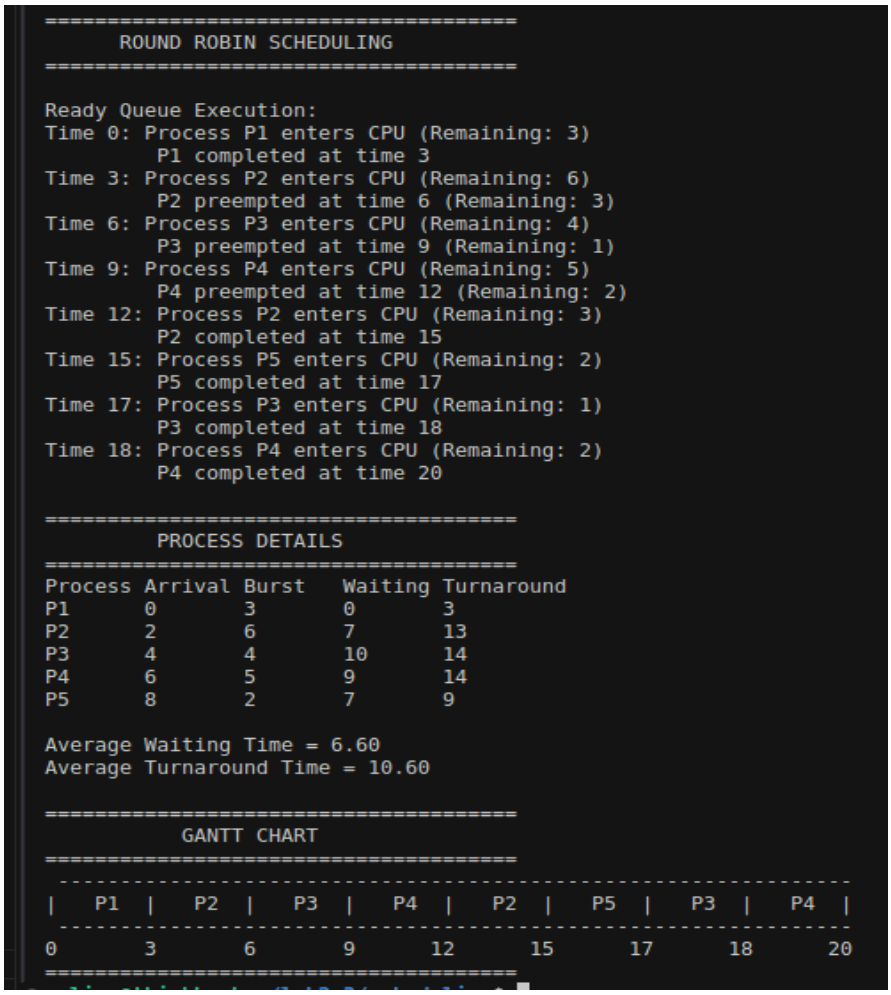
Source Code:



```c
printf("\n====================================");
printf("\n      ROUND ROBIN SCHEDULING");
printf("\n====================================\n");
printf("\nReady Queue Execution:\n");

while (remain > 0) {
    if (front == rear) {
        // No process in queue, advance time
        time++;
        // Check for newly arrived processes
        for (i = 0; i < n; i++) {
            if (at[i] == time && visited[i] == 0) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
        continue;
    }

    i = queue[front++];

    printf("Time %d: Process P%d enters CPU (Remaining: %d)\n", time, i + 1, rt[i]);

    // Record for Gantt chart
    gantt[gantt_count] = i;
    gantt_time[gantt_count] = time;
    gantt_count++;

    if (rt[i] <= quantum && rt[i] > 0) {
        // Process will complete
        time += rt[i];
        rt[i] = 0;
        completed[i] = 1;
        remain--;

        // Calculate times
        tat[i] = time - at[i];
        wt[i] = tat[i] - bt[i];
        total_wt += wt[i];
        total_tat += tat[i];

        printf("         P%d completed at time %d\n", i + 1, time);

        // Add newly arrived processes to queue
        for (j = 0; j < n; j++) {
            if (at[j] <= time && visited[j] == 0 && completed[j] == 0) {
                queue[rear++] = j;
                visited[j] = 1;
            }
        }
    } else if (rt[i] > 0) {
        // Process needs more time
        time += quantum;
        rt[i] -= quantum;

        printf("         P%d preempted at time %d (Remaining: %d)\n", i + 1, time, rt[i]);

        // Add newly arrived processes to queue first
        for (j = 0; j < n; j++) {
            if (at[j] <= time && visited[j] == 0 && completed[j] == 0) {
                queue[rear++] = j;
                visited[j] = 1;
            }
        }

        // Add current process back to queue
        queue[rear++] = i;
    }
}
```

```c
    // Add final time to Gantt chart
    gantt_time[gantt_count] = time;

    // Display process details
    printf("\n=====================================");
    printf("\n         PROCESS DETAILS");
    printf("\n=====================================\n");
    printf("Process\tArrival\tBurst\tWaiting\tTurnaround\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], wt[i], tat[i]);
    }

    avg_wt = (float)total_wt / n;
    avg_tat = (float)total_tat / n;

    printf("\nAverage Waiting Time = %.2f", avg_wt);
    printf("\nAverage Turnaround Time = %.2f\n", avg_tat);

    // Display Gantt Chart
    printf("\n=====================================");
    printf("\n           GANTT CHART");
    printf("\n=====================================\n");

    // Top border
    printf(" ");
    for (i = 0; i < gantt_count; i++) {
        printf("--------");
    }
    printf("\n");

    // Process names
    printf("|");
    for (i = 0; i < gantt_count; i++) {
        printf("   P%d  |", gantt[i] + 1);
    }
    printf("\n");

    // Bottom border
    printf(" ");
    for (i = 0; i < gantt_count; i++) {
        printf("--------");
    }
    printf("\n");

    // Time markers
    printf("%d", gantt_time[0]);
    for (i = 1; i <= gantt_count; i++) {
        if (gantt_time[i] < 10) {
            printf("       %d", gantt_time[i]);
        } else {
            printf("      %d", gantt_time[i]);
        }
    }
    printf("\n");
    printf("=====================================\n");

    return 0;
}
```

Output:



```
====================================
      ROUND ROBIN SCHEDULING
====================================

Ready Queue Execution:
Time 0: Process P1 enters CPU (Remaining: 3)
         P1 completed at time 3
Time 3: Process P2 enters CPU (Remaining: 6)
         P2 preempted at time 6 (Remaining: 3)
Time 6: Process P3 enters CPU (Remaining: 4)
         P3 preempted at time 9 (Remaining: 1)
Time 9: Process P4 enters CPU (Remaining: 5)
         P4 preempted at time 12 (Remaining: 2)
Time 12: Process P2 enters CPU (Remaining: 3)
         P2 completed at time 15
Time 15: Process P5 enters CPU (Remaining: 2)
         P5 completed at time 17
Time 17: Process P3 enters CPU (Remaining: 1)
         P3 completed at time 18
Time 18: Process P4 enters CPU (Remaining: 2)
         P4 completed at time 20

====================================
         PROCESS DETAILS
====================================
Process Arrival Burst   Waiting Turnaround
P1      0       3       0       3
P2      2       6       7       13
P3      4       4       10      14
P4      6       5       9       14
P5      8       2       7       9

Average Waiting Time = 6.60
Average Turnaround Time = 10.60

====================================
           GANTT CHART
====================================
 ------------------------------------------------------------------
|   P1  |   P2  |   P3  |   P4  |   P2  |   P5  |   P3  |   P4  |
 ------------------------------------------------------------------
0       3       6       9       12      15      17      18      20
====================================
```

Time Quantum Selection:

- Small quantum: Better response time but higher context switching overhead
- Large quantum: Approaches FCFS, less overhead but poor response time
- Optimal quantum: Usually 10-100 milliseconds, depends on system requirements

Conclusion

Through these lab exercises, we have:

1. **Lab 2 Learnings:**

    - Understood the fork() system call and process creation mechanism
    - Observed how multiple fork() calls create exponential numbers of processes (2^n)
    - Learned to distinguish between parent and child processes using PIDs
    - Recognized that process execution order is non-deterministic

2. **Lab 3 Learnings:**

    - Distinguished between preemptive and non-preemptive scheduling
    - Implemented SJF for optimal average waiting time (non-preemptive)
    - Implemented SRTF for optimal performance with preemption
    - Implemented Round Robin for fair time-sharing with configurable quantum

These concepts are fundamental to understanding operating system process management and CPU scheduling, which are crucial for system performance optimization.