Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

"Lab Report 02 & 03"

[Code No: COMP 307]

Submitted by:

Krishtina Bhatta

Roll no: 09

Submitted to:

Rabina Shrestha

Department of Computer Science and Engineering

Date: 11th January, 2026

Lab Report 2

Fork()

Fork() is a system call in UNIX/Linux used to create a new process. The existing process is called the parent process. The newly created process is called the child process. After fork() both parent and child run concurrently and execute the same program, but they are different processes.

Program 1: Process creation

Objective: A program that uses the fork.

Source code:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        printf("This demonstrates the fork\n");
        fork();
        printf("Hello world\n");
        return 0;
}
```

Output:

```
Hello world
Hello world
```

Analysis:

In the main program, we can see that a function fork() is called. Here, fork() is the name of the system call that the parent process uses to "divide" itself ("fork" into two identical processes). After calling fork(), the created child process is actually an exact copy of the parent, which would probably be of limited use, so it replaces itself with another process.

Questions:

Part A – Two fork() calls

1. Modify the above program and add a total **of two fork()** on it

```
fork();
fork();
```

Ans→

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("This demonstrates the fork\n");
    fork();
    fork();
    printf("Hello world\n");
    return 0;
}
```

2. Run the program and check how many times **"Hello world"** will be printed?

Ans→

```
#include <stdio.h>
#include <unistd.h>
int main()
{
printf("This demonstrates the fork\n");
        fork();
        fork();
    printf("Hello world\n");
    return 0;
}
```

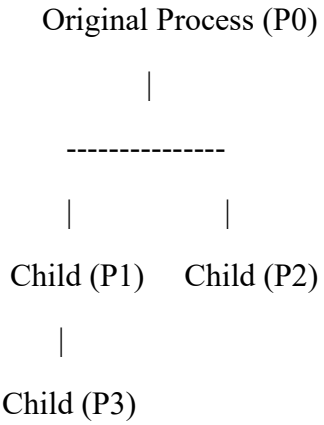
```
krishtina_15@Krishtina1510:~$ ls
file.txt  final_report.txt  fork  fork.c  graphics.txt  my_config.txt.save  name.txt  new.txt  status.log
krishtina_15@Krishtina1510:~$ gcc fork.c -o fork
krishtina_15@Krishtina1510:~$ ./fork
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
krishtina_15@Krishtina1510:~$
```

From the output above , we see **Hello world** is printed **4 times**.

3. Draw a **process tree diagram** showing which processes are created at each fork.

Write a brief **analysis** of what happens and why?

Ans→ **Process Tree Diagram**



Analysis:

When the program starts, there is only one parent process. The first fork() call creates a child process, so there are now two processes running. Both of these processes execute the second fork() call, and each creates another child process. As a result, a total of four processes are created. Each process continues execution independently and reaches the printf("Hello world") statement. Therefore, “Hello world” is printed four times. This happens because each fork() duplicates the currently running processes.

Part B – Three fork() calls

1. Modify the above program to add **three fork()** calls on separate lines:

```
fork();

fork();

fork();
```

Ans→

```
#include <stdio.h>
#include <unistd.h>
int main()
 {
        printf("This demonstrates the fork\n");
        fork();
        fork();
        fork();
        printf("Hello world\n");
        return 0;
}
```

2. Run the program and check how many times **“Hello world”** will be printed?

Ans→

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

int main() {
    printf("This demonstrates the fork\n");

    fork();
    fork();
    fork();

    printf("Hello world\n");
    return 0;
}
```

```
krishtina_15@Krishtina1510:~$ ls
file.txt  final_report.txt  fork  fork.c  graphics.txt  my_config.txt.save  name.txt  new.txt  status.log
krishtina_15@Krishtina1510:~$ gcc fork.c -o fork
krishtina_15@Krishtina1510:~$ ./fork
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
krishtina_15@Krishtina1510:~$
```

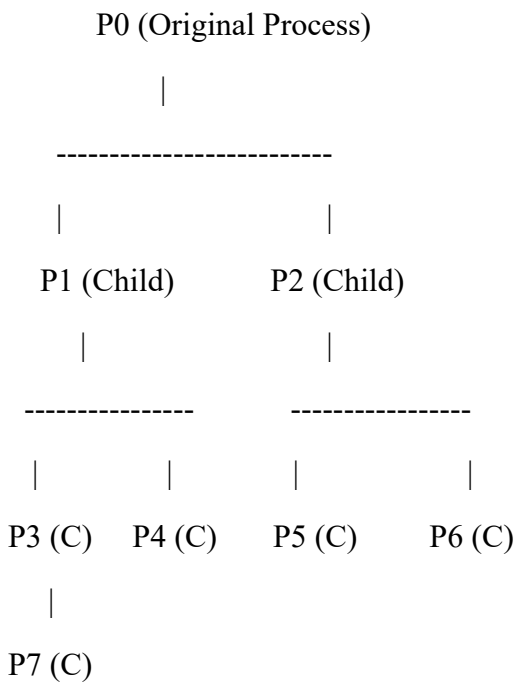
Each fork() doubles the number of running processes.Number of processes after n forks = 2^n

Here, n = 3, so total processes = $2^3 = 8$

Hence , **Hello world** is printed **8 times**.

3. Draw a **process tree diagram** showing all processes created?

Ans→**Process Tree Diagram**



4. Analyze the output and explain why the number of prints increases with each fork.

Ans→

When the program executes, the first fork() creates a new child process, resulting in two processes: the original parent (P0) and the child (P1). Both processes continue executing the program independently. The second fork() duplicates both existing processes, creating two more processes (P2 and P3), so now there are four processes running simultaneously. Each process executes the code after the fork independently.The third fork() duplicates all four existing processes, creating four additional processes (P4, P5, P6, and P7), bringing the total number of processes to eight. Each of these eight processes executes the printf("Hello world") statement independently, which is why the message is printed eight times.

The number of prints increases with each fork because each fork doubles the number of running processes and every process executes the remaining code independently. This exponential growth follows the formula:

Number of prints = 2^n,

where n is the number of fork calls.

Program 2: Understanding fork() with PID

Objective: Another program that shows the use of forking

Source Code:

```
#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:

Questions:

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

and the parent?

Ans→

In the above program, the variable pid stores the return value of the fork() system call. After fork() is executed, the return value behaves differently in the parent and child processes. When pid == 0, the code is running in the child process because fork() returns 0 to the newly created child. When pid > 0, the code is running in the parent process and the value of pid is the processID (PID) of the child. This difference allows the program to distinguish between parent and child and execute different instructions for each.

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

Ans→

The parent and child processes have different PIDs assigned by the operating system. When the parent executes getpid(), it prints its own PID, which is unique to that process. The child, being a newly created process, has a different PID, so when it calls getpid(), it prints a different number. This distinction is important because, despite having identical code, the processes are independent and can be identified uniquely by their PIDs.

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

Ans→

In this program, the lines before the fork() (e.g., the first two printf() statements) are executed only once by the parent, so they appear first in the output. After the fork(), both the parent and child execute the code following it independently. The operating system's process scheduler decides which process runs first, so typically the parent lines appear before the child lines, but this is not guaranteed. Depending on the OS scheduling, sometimes the child may execute first, so the output order can vary between runs.

Lab Report 3

Simulation of Process Scheduling Algorithm

1. Write short notes on Preemptive and Non preemptive scheduling

Ans→

Preemptive Scheduling

Preemptive scheduling is a CPU scheduling method in which the operating system has the authority to forcibly remove a running process from the CPU in order to allocate it to another process, typically based on priority or time-slice allocation. This ensures that higher-priority or time-sensitive processes receive the CPU promptly, which improves system responsiveness and allows multiple processes to share the CPU fairly. Each time a process is preempted, a context switch occurs, saving the state of the current process and loading the state of the next process. Preemptive scheduling is widely used in modern multitasking operating systems and is implemented in algorithms such as Round Robin, Shortest Remaining Time First (SRTF), and Preemptive Priority Scheduling. While it provides better CPU utilization and responsiveness, it also introduces complexity due to frequent context switching and the need to handle concurrency carefully.

Non Preemptive Scheduling

Non-preemptive scheduling is a CPU scheduling method in which a process, once allocated the CPU, continues its execution until it either completes or voluntarily relinquishes control of the CPU, such as when it performs an I/O operation. In this approach, the CPU is not taken away forcibly, so higher-priority processes may have to wait until the running process finishes. Non-preemptive scheduling is simpler to implement and does not involve as many context switches, making it easier to manage. However, it can lead to longer waiting times for important processes and may result in inefficient CPU utilization if long processes block the CPU. Common examples of non-preemptive algorithms include First-Come-First-Serve (FCFS), Shortest Job First (SJF) and Non-Preemptive Priority Scheduling. Although it is less responsive compared to preemptive

scheduling, it is suitable for systems where process execution times are predictable and fairness is less critical.

Q 1) WAP in C to simulate FCFS CPU Scheduling Algorithm.

Ans→

FCFS (First Come First Serve) Scheduling

First-Come First-Serve (FCFS) is one of the simplest CPU scheduling algorithms. In this non-preemptive method, processes are executed in the order they arrive in the ready queue. The first process that arrives gets the CPU first and runs until completion before the next process starts.It is useful in batch systems where all processes are ready at the same time and in Simple scheduling in small systems without real-time requirements.

Applied algorithm below:

1.The waiting time of the first process is zero

2. Waiting time of each process is the sum of the burst times of all previous processes

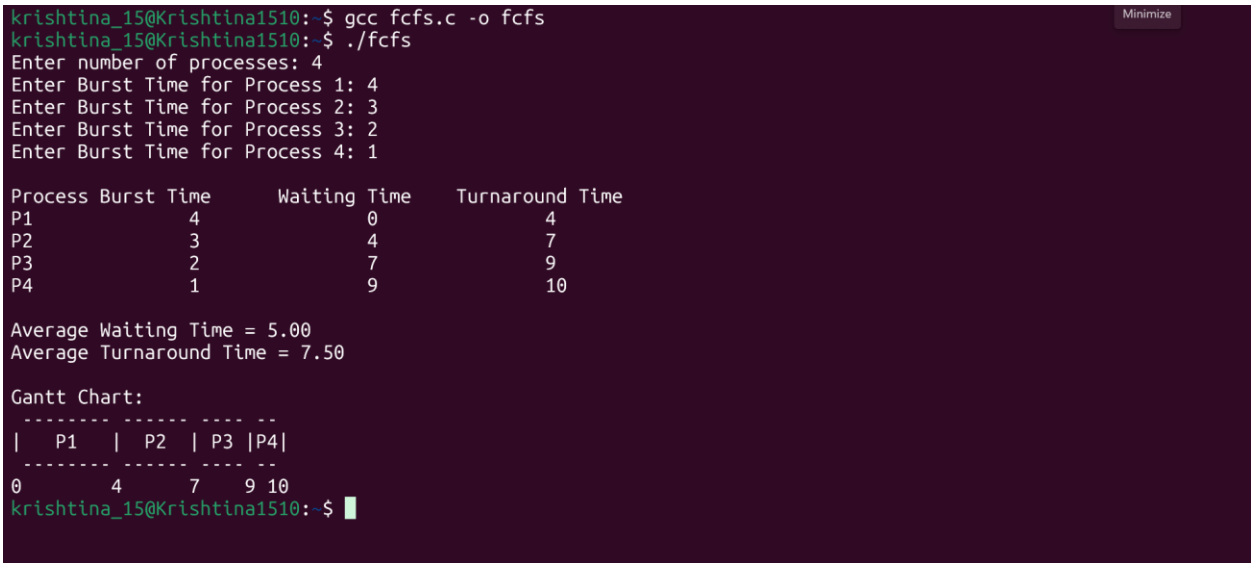
3. Turnaround time = Waiting Time + Burst Time

FCFS is simple to implement, but it may cause the convoy effect, where short processes wait for a long process to finish.

Code:

```c
#include <stdio.h>

int main() {
    int n, i, j;
    int bt[20], wt[20], tat[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]);
    }

    // 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 and totals
    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
        total_wt += wt[i];
        total_tat += tat[i];
    }

    // Print Process Table
    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", i + 1, 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);

    // Print Gantt Chart
    printf("\nGantt Chart:\n");

    // Top bar
    for (i = 0; i < n; i++) {
        printf(" ");
        for (j = 0; j < bt[i]; j++) printf("--");
    }
    printf("\n|");

    // Process IDs
    for (i = 0; i < n; i++) {
        for (j = 0; j < bt[i]-1; j++) printf(" ");
        printf("P%d", i+1);
        for (j = 0; j < bt[i]-1; j++) printf(" ");
        printf("|");
    }
    printf("\n");

    // Bottom bar
    for (i = 0; i < n; i++) {
        printf(" ");
        for (j = 0; j < bt[i]; j++) printf("--");
    }
    printf("\n");

    // Timeline
    printf("0");
    for (i = 0; i < n; i++) {
        for (j = 0; j < bt[i]; j++) printf("  ");
        if (tat[i] > 9) // adjust spacing for 2-digit numbers
            printf("\b");
        printf("%d", tat[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



Q 2) WAP in C to simulate the SJF Process Scheduling Algorithm and Gantt Chart.

Ans→

SJS (Shortest Job First) Scheduling

Shortest Job First (SJF) Scheduling is a non-preemptive CPU scheduling algorithm where the process with the shortest burst time is executed first. The waiting time of each process is calculated as the sum of burst times of all previous processes, and turnaround time is the sum of waiting time and burst time. A Gantt Chart can be used to visualize the execution order and timings. SJF reduces the average waiting time compared to FCFS, but longer processes may wait if many short processes arrive first.

Applied algorithm below:

1. Non-preemptive SJF: The CPU always executes the process with the shortest burst time among the ready processes. Once a process starts, it runs to completion.
2. The program sorts processes based on burst time before scheduling.
3. Waiting time (WT) of a process = sum of burst times of all previous processes.
4. Turnaround time (TAT) = Waiting Time + Burst Time.
5. The program also prints a Gantt Chart to visualize the order of execution.

Code:

```c
#include <stdio.h>

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

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

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

    // 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;
            }
        }
    }

    // Calculate waiting time
    wt[0] = 0; // first process has 0 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];
    }

    // Print Process Table
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\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");
    // Top bar
    for (i = 0; i < n; i++) {
        printf(" ");
        for (j = 0; j < bt[i]; j++) printf("--");
    }
    printf("\n|");
    // Process IDs
    for (i = 0; i < n; i++) {
        for (j = 0; j < bt[i]-1; j++) printf(" ");
        printf("P%d", p[i]);
        for (j = 0; j < bt[i]-1; j++) printf(" ");
        printf("|");
    }
    printf("\n");
    // Bottom bar
    for (i = 0; i < n; i++) {
        printf(" ");
        for (j = 0; j < bt[i]; j++) printf("--");
    }
    printf("\n");
    // Timeline
    printf("0");
```

```
    for (i = 0; i < n; i++) {
        for (j = 0; j < bt[i]; j++) printf("  ");
        printf("%d", tat[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
krishtina_15@Krishtina1510:~$ gcc sjf.c -o sjf
krishtina_15@Krishtina1510:~$ ./sjf
Enter number of processes: 4
Enter Burst Time for Process 1: 6
Enter Burst Time for Process 2: 4
Enter Burst Time for Process 3: 8
Enter Burst Time for Process 4: 3

Process Burst Time      Waiting Time    Turnaround Time
P4      3               0               3
P2      4               3               7
P1      6               7               13
P3      8               13              21

Average Waiting Time = 5.75
Average Turnaround Time = 11.00

Gantt Chart:
 ------ -------- ------------ ----------------
|  P4  |   P2   |     P1     |       P3       |
 ------ -------- ------------ ----------------
0       3        7            13               21
krishtina_15@Krishtina1510:~$
```

Q 3) WAP in C to simulate the SRTF Process Scheduling Algorithm and Gantt Chart.

Ans→

SRTF (Shortest Remaining Time First) Scheduling

Shortest Remaining Time First (SRTF) Scheduling is a preemptive CPU scheduling algorithm where the process with the shortest remaining burst time is executed next. In this implementation, at every unit of time, the program checks which process has the minimum remaining time among all processes that have arrived and executes it. If a new process arrives with a shorter remaining time, the CPU switches to that process, making SRTF preemptive. Waiting time and turnaround time are calculated when a process completes, and a Gantt Chart is used to visualize the execution order. SRTF generally reduces the average waiting time compared to FCFS and SJF (non-preemptive), but frequent switching may slightly increase overhead.

Applied algorithm below:

1. SRTF is the preemptive version of SJF.
2. The CPU always executes the process with the shortest remaining burst time among all ready processes.

3. If a new process arrives with a shorter burst time than the remaining time of the current process, the CPU preempts the running process.
4. Waiting time for each process is calculated based on the total time the process spends in the ready queue.
5. Turnaround time = Waiting Time + Burst Time.

Code:

```c
#include <stdio.h>

int main() {
    int n, i, j, t = 0, completed = 0;
    int at[20], bt[20], rt[20], wt[20], tat[20];
    int total_wt = 0, total_tat = 0;

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

    for (i = 0; i < n; i++) {
        printf("Enter Arrival Time for Process %d: ", i + 1);
        scanf("%d", &at[i]);
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i]; // initialize remaining time
    }

    printf("\nGantt Chart:\n");

    int prev = -1;
    while (completed < n) {
        int min_index = -1;
        int min_rt = 1e9;

        // Find process with minimum remaining time at time t
        for (i = 0; i < n; i++) {
            if (at[i] <= t && rt[i] > 0 && rt[i] < min_rt) {
                min_rt = rt[i];
                min_index = i;
            }
        }

        if (min_index != -1) {
            // Print process for Gantt Chart if switching occurs
            if (prev != min_index) {
                printf("| P%d ", min_index + 1);
                prev = min_index;
            }

            rt[min_index]--; // execute process for 1 unit
            t++;

            // If process is finished
            if (rt[min_index] == 0) {
                completed++;
                tat[min_index] = t - at[min_index];
                wt[min_index] = tat[min_index] - bt[min_index];
                total_wt += wt[min_index];
                total_tat += tat[min_index];
            }
        } else {
            t++; // CPU idle
        }
```

```
    }
    printf("|\n\n");

    // Print Process Table
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[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);

    return 0;
}
```

Output:

```
krishtina_15@Krishtina1510:~$ gcc srtf.c -o srtf
krishtina_15@Krishtina1510:~$ ./srtf
Enter number of processes: 4
Enter Arrival Time for Process 1: 2
Enter Burst Time for Process 1: 3
Enter Arrival Time for Process 2: 4
Enter Burst Time for Process 2: 5
Enter Arrival Time for Process 3: 6
Enter Burst Time for Process 3: 7
Enter Arrival Time for Process 4: 8
Enter Burst Time for Process 4: 9

Gantt Chart:
| P1 | P2 | P3 | P4 |

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
P1      2               3               0               3
P2      4               5               1               6
P3      6               7               4               11
P4      8               9               9               18

Average Waiting Time = 3.50
Average Turnaround Time = 9.50
krishtina_15@Krishtina1510:~$
```

Q 4) WAP in C to simulate the Round Robin Process Scheduling Algorithm and Gantt Chart.

Ans→

RR (Round Robin) Process Scheduling

Round Robin (RR) Scheduling is a preemptive CPU scheduling algorithm where each process is executed for a fixed time quantum in a cyclic order. If a process does not finish within its time slice, it is moved to the end of the ready queue and waits for its next turn. In this implementation, the program keeps track of remaining burst times, updates waiting and turnaround times when a process finishes and prints a Gantt Chart to visualize the order of execution. RR improves response time for shorter processes and ensures fair CPU sharing, but performance depends on the chosen time quantum: too small increases context switching, too large behaves like FCFS.

Applied algorithm below:

1. Input arrival time and burst time of all processes.
2. Input the time quantum.
3. Maintain remaining burst times for all processes.
4. At each time unit, execute the next process in the ready queue for a time slice equal to the time quantum or until the process finishes, whichever is smaller.
5. Move the process to the end of the queue if it is not finished.
6. Repeat until all processes are complete.
7. Calculate waiting time = turnaround time – burst time, and turnaround time = completion time – arrival time.
8. Print the Gantt Chart to show the order of execution.

Code:

```c
#include <stdio.h>

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

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

    for (i = 0; i < n; i++) {
        printf("Enter Arrival Time for Process %d: ", i+1);
        scanf("%d", &at[i]);
        printf("Enter Burst Time for Process %d: ", i+1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i]; // initialize remaining time
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &tq);

    int t = 0, completed = 0;
    printf("\nGantt Chart:\n");

    while (completed < n) {
        int done = 1; // check if all processes finished
        for (i = 0; i < n; i++) {
            if (rt[i] > 0 && at[i] <= t) {
                done = 0;
                int exec = (rt[i] < tq) ? rt[i] : tq;
                printf("| P%d ", i+1); // print for Gantt chart
                rt[i] -= exec;
                t += exec;
                if (rt[i] == 0) {
                    tat[i] = t - at[i];
                    wt[i] = tat[i] - bt[i];
                    total_wt += wt[i];
                    total_tat += tat[i];
                    completed++;
                }
            }
        }
        if (done) t++; // idle time if no process has arrived
    }
    printf("|\n\n");

    // Print Process Table
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[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);

    return 0;
}
```

Output:

```
krishtina_15@Krishtina1510:~$ gcc roundrobin.c -o roundrobin
krishtina_15@Krishtina1510:~$ ./roundrobin
Enter number of processes: 3
Enter Arrival Time for Process 1: 0
Enter Burst Time for Process 1: 5
Enter Arrival Time for Process 2: 1
Enter Burst Time for Process 2: 4
Enter Arrival Time for Process 3: 2
Enter Burst Time for Process 3: 2
Enter Time Quantum: 2

Gantt Chart:
| P1 | P2 | P3 | P1 | P2 | P1 |

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
P1      0               5               6               11
P2      1               4               5               9
P3      2               2               2               4

Average Waiting Time = 4.33
Average Turnaround Time = 8.00
krishtina_15@Krishtina1510:~$
```