Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 2&3

[Code No: COMP 307]

Submitted by:

Nischal Subedi (53)

Group: CS60 (III/I)

.

Submitted to:

Ms. Rabina Shrestha

Department of Computer Science and Engineering

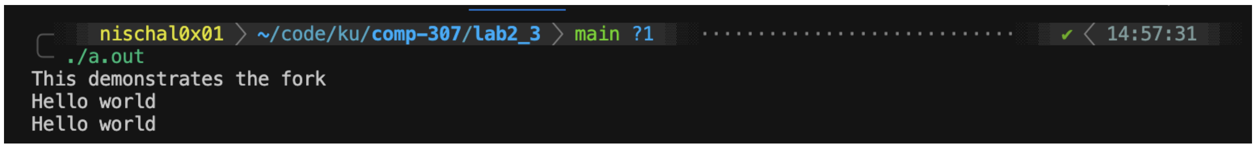
Submission Date: 11/01/2026

Lab-2

Program 1: Process creation
Objective: A program that uses the fork.

Source code:

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



```
nischal0x01 ❯ ~/code/ku/comp-307/lab2_3 ❯ main ?1 ····························· ✔ ❮ 14:57:31
./a.out
This demonstrates the fork
Hello world
Hello world
```

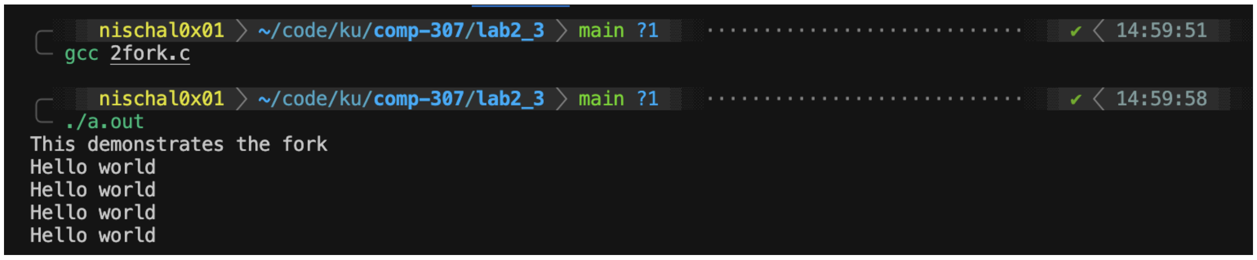
The `fork()` system call creates a new child process which is an exact copy of the parent process. After `fork()` executes, **both parent and child continue execution from the next statement**, which causes `"Hello world"` to be printed twice.

<u>Part-1</u>

Q1. Modify the above program and add a total of two fork() on it
fork();
fork();

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

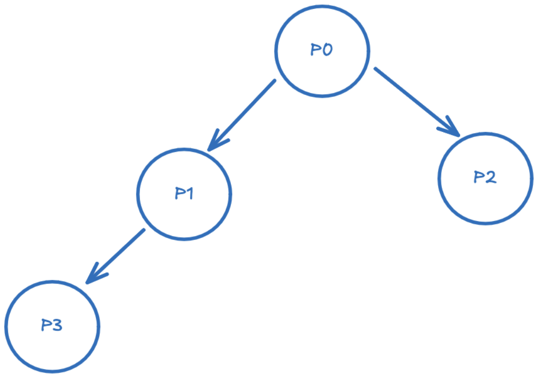
Q2. Run the program and check how many times “Hello world” will be printed?



```
  nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ······························ ✔ < 14:59:51
  gcc 2fork.c

  nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ······························ ✔ < 14:59:58
  ./a.out
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
```

Here “Hello world” is printed 4 times

Q3. Draw a process tree diagram showing which processes are created at each fork. Write a brief



Analysis

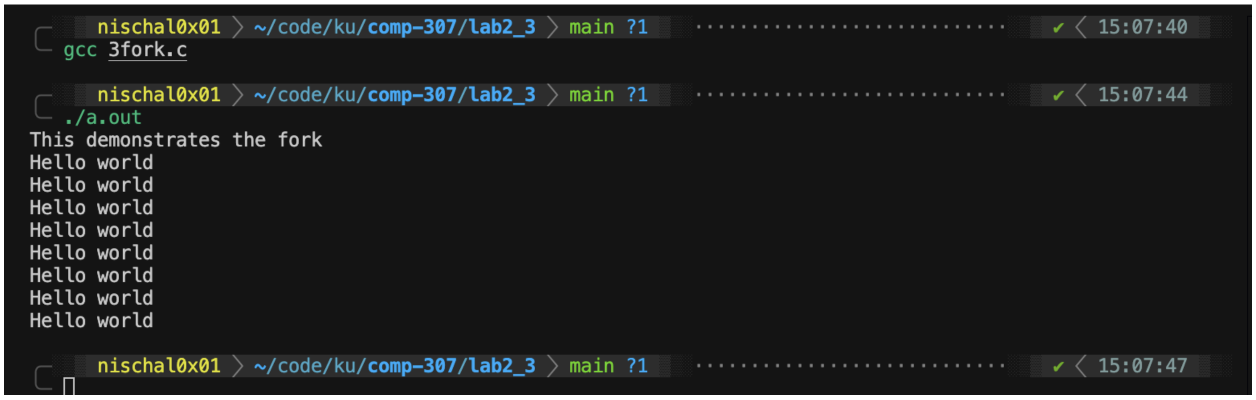
- First `fork()` creates 2 processes
- Second `fork()` is executed by both processes
- Total processes = $2^2 = 4$
- Each process executes `printf()`, so `"Hello world"` prints 4 times

<u>Part-2</u>

Q1. Modify the above program and add a total of three fork() on it
fork();
fork();
fork();

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

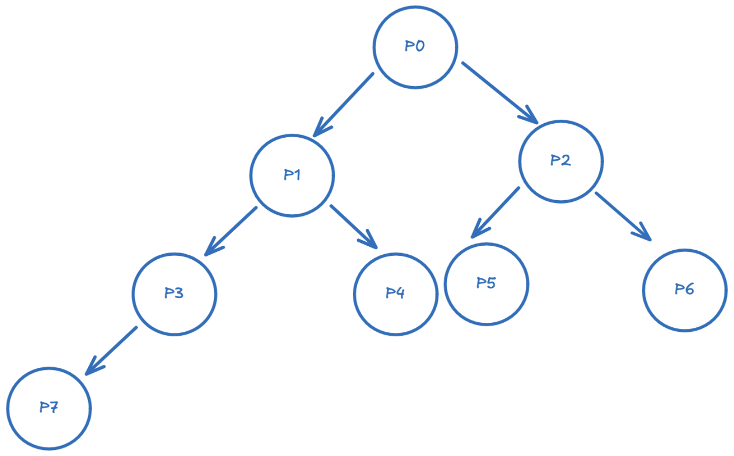
Q2. Run the program and check how many times “Hello world” will be printed?



```
 nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:07:40
 gcc 3fork.c

 nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:07:44
 ./a.out
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world

 nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:07:47
```

Here “Hello world” is printed 8 times

Q3. Draw a process tree diagram showing all processes created? Analyze the output and explain why the number of prints increases with each fork.



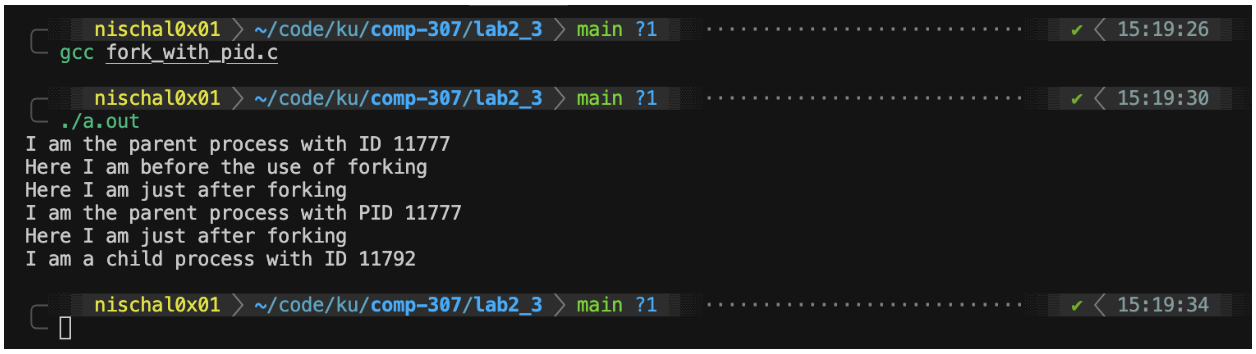
Analysis

- Each `fork()` doubles the number of running processes
- Total processes = $2^3 = 8$
- Each process prints `"Hello world"`
- Hence, output appears 8 times

Program 2: Understanding fork() with PID
Objective: Another program that shows the use of forking

```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:



```
nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:19:26
gcc fork_with_pid.c

nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:19:30
./a.out
I am the parent process with ID 11777
Here I am before the use of forking
Here I am just after forking
I am the parent process with PID 11777
Here I am just after forking
I am a child process with ID 11792

nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ............................ ✔ < 15:19:34
```

Q1. Explain the difference between pid == 0 and pid > 0. Which one corresponds to the child and the parent?

When the `fork()` system call is executed, it returns different values to the parent and child processes.

- If `pid == 0`, it means the process is the child process.
- If `pid > 0`, it means the process is the parent process, and the value returned is the Process ID (PID) of the child.

This return value helps the program distinguish between parent and child, even though both execute the same code after the `fork()` call. The operating system uses this mechanism so that both processes can take different execution paths if required.

Q2. Why do the parent and child processes print their own PID differently?

Although the child process is created as an exact copy of the parent, the operating system assigns a unique PID to each process.
The function `getpid()` returns the PID of the currently executing process, not the parent's PID.

Therefore:

- The parent prints its original PID
- The child prints a new PID assigned by the OS

This difference proves that parent and child are separate and independent processes, even though they share the same code and data at the time of creation.

Q3. Explain why the output order is parent lines first, child lines after. Can the order change? Why?

In most executions, the parent process prints its output first because it was already running before the child was created. After the `fork()` call, the operating system scheduler usually allows the parent to continue execution briefly before switching to the child process.

However, this order is not guaranteed. The operating system uses CPU scheduling, and depending on system load, priority, and timing, the child process may execute before the parent.

So yes, the output order can change, because:

- Parent and child run concurrently
- Process scheduling is controlled by the OS
- There is no synchronization between parent and child in this program

Lab-3

1. Write short notes on Preemptive and Non preemptive scheduling.

Preemptive Scheduling

Preemptive scheduling is a CPU scheduling technique in which the operating system can interrupt a currently running process and assign the CPU to another process with higher priority or shorter remaining execution time. This type of scheduling improves system responsiveness and is especially useful in time-sharing and real-time systems. Algorithms such as Shortest Remaining Time First (SRTF) and Round Robin use preemption to ensure fair CPU allocation and better response time. However, frequent context switching may increase overhead and reduce CPU efficiency.

Non-Preemptive Scheduling

Non-preemptive scheduling is a CPU scheduling method where once a process starts executing, it continues until it finishes or enters a waiting state. The CPU cannot be taken away forcibly from the running process. This approach is simple to implement and has minimal context switching overhead. Algorithms like First Come First Serve (FCFS) and non-preemptive Shortest Job First (SJF) follow this method. However, it may lead to poor response time and problems like the convoy effect, where short processes wait behind long processes.

Preemptive	**Non-Preemptive**
CPU can be taken away from a running process	CPU is allocated until the process finishes execution
Improves response time	No interruption once execution starts
Example: SRTF, Round Robin	Example: FCFS, SJF (Non-Preemptive)

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

Shortest Job First (SJF) is a non-preemptive CPU scheduling algorithm in which the process with the smallest burst time is selected for execution first. Once a process starts executing, it runs until completion without interruption. SJF minimizes the average waiting time but requires prior knowledge of burst times, which may not always be practical.

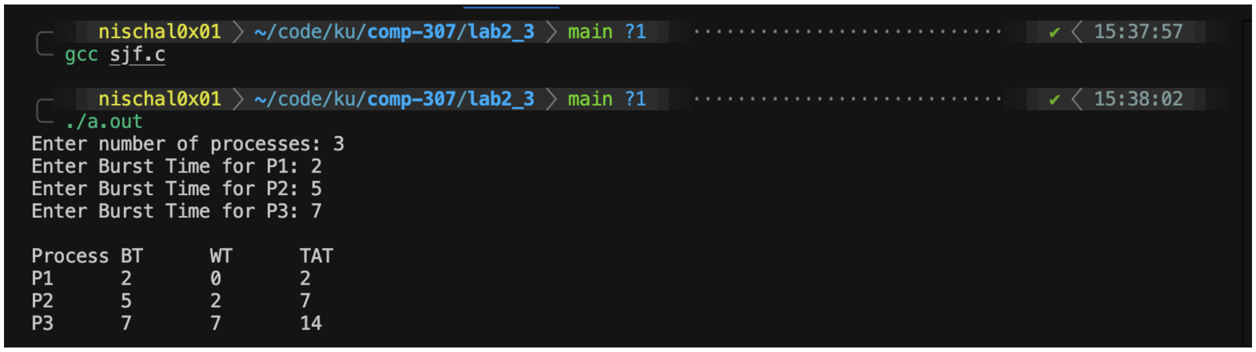
Algorithm

1. Input number of processes and their burst times
2. Sort processes in ascending order of burst time
3. Set waiting time of first process to zero
4. Calculate waiting time for remaining processes
5. Calculate turnaround time
6. Display results and averages

```c
#include <stdio.h>

int main() {

   int n, i, j, temp;

   int bt[10], wt[10], tat[10];

   printf("Enter number of processes: ");

   scanf("%d", &n);

   for(i = 0; i < n; i++) {

       printf("Enter Burst Time for P%d: ", i+1);

       scanf("%d", &bt[i]);

   }

   // Sort burst times

   for(i = 0; i < n-1; i++) {

       for(j = i+1; j < n; j++) {

           if(bt[i] > bt[j]) {

               temp = bt[i];
```

```c
                bt[i] = bt[j];

                bt[j] = temp;

            }

        }

    }


    wt[0] = 0;

    for(i = 1; i < n; i++)

        wt[i] = wt[i-1] + bt[i-1];


    for(i = 0; i < n; i++)

        tat[i] = wt[i] + bt[i];


    printf("\nProcess\tBT\tWT\tTAT\n");

    for(i = 0; i < n; i++)

        printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);


    return 0;

}
```

Output



```
nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ··························· ✔ < 15:37:57
gcc sjf.c

nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ··························· ✔ < 15:38:02
./a.out
Enter number of processes: 3
Enter Burst Time for P1: 2
Enter Burst Time for P2: 5
Enter Burst Time for P3: 7

Process BT      WT      TAT
P1      2       0       2
P2      5       2       7
P3      7       7       14
```

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

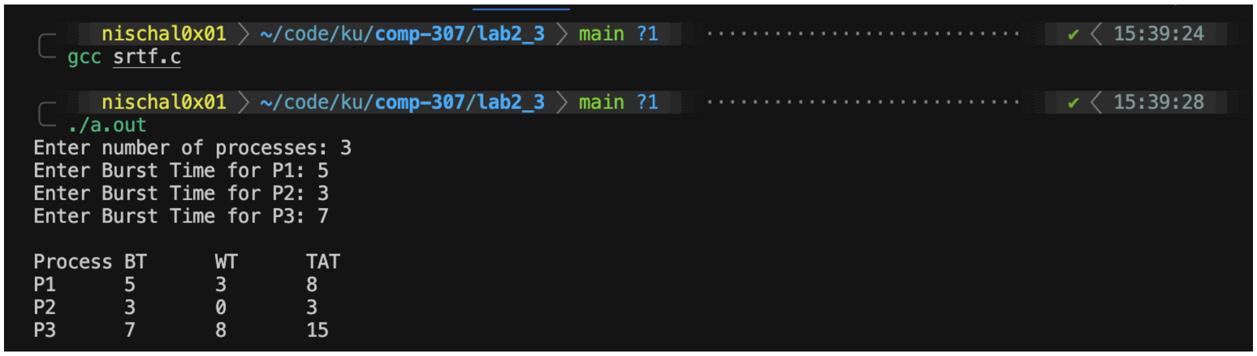
Shortest Remaining Time First (SRTF) is the preemptive version of SJF, where the CPU is allocated to the process with the least remaining execution time. If a new process arrives with a shorter remaining time than the current process, the CPU is preempted. This improves response time but increases context switching.

Algorithm

1. Input burst times of processes
2. Initialize remaining time equal to burst time
3. At each time unit, select process with shortest remaining time
4. Execute for one unit of time
5. Repeat until all processes finish
6. Calculate waiting and turnaround times

```c
#include <stdio.h>

int main() {

   int n, bt[10], rt[10], wt[10], tat[10];

   int time = 0, remain, smallest;


   printf("Enter number of processes: ");

   scanf("%d", &n);


   for(int i = 0; i < n; i++) {

       printf("Enter Burst Time for P%d: ", i+1);

       scanf("%d", &bt[i]);

       rt[i] = bt[i];

   }

   remain = n;

   while(remain != 0) {

       smallest = -1;

       for(int i = 0; i < n; i++) {
            if(rt[i] > 0 && (smallest == -1 || rt[i] < rt[smallest]))
                smallest = i;
        }

        rt[smallest]--;
        time++;

        if(rt[smallest] == 0) {
            remain--;
            wt[smallest] = time - bt[smallest];
            tat[smallest] = time;
        }
    }

    printf("\nProcess\tBT\tWT\tTAT\n");
    for(int i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);

    return 0;
}
```

Output:



```
nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ·························· ✔ < 15:39:24
gcc srtf.c

nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1 ·························· ✔ < 15:39:28
./a.out
Enter number of processes: 3
Enter Burst Time for P1: 5
Enter Burst Time for P2: 3
Enter Burst Time for P3: 7

Process BT      WT      TAT
P1      5       3       8
P2      3       0       3
P3      7       8       15
```

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

Round Robin is a preemptive scheduling algorithm designed for time-sharing systems. Each process is assigned a fixed time quantum, and the CPU cycles through processes in a circular queue. If a process does not complete within its time quantum, it is preempted and placed at the end of the queue, ensuring fairness.

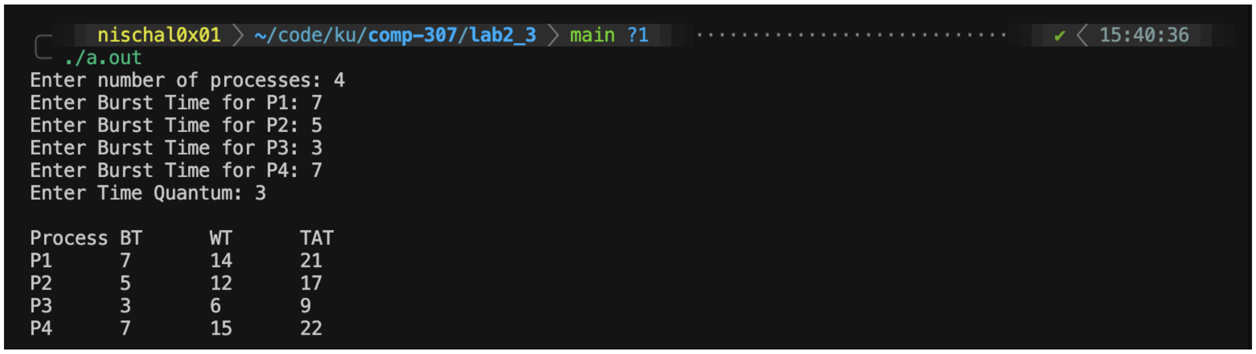
Brief Algorithm

1. Input number of processes and burst times
2. Input time quantum
3. Allocate CPU to each process for time quantum
4. Reduce remaining time accordingly
5. Repeat until all processes finish
6. Calculate waiting and turnaround times

```c
#include <stdio.h>

int main() {

   int n, tq, bt[10], rt[10], wt[10] = {0}, tat[10];

   int time = 0, done;

   printf("Enter number of processes: ");

   scanf("%d", &n);

   for(int i = 0; i < n; i++) {

       printf("Enter Burst Time for P%d: ", i+1);

       scanf("%d", &bt[i]);

       rt[i] = bt[i];

   }

   printf("Enter Time Quantum: ");

   scanf("%d", &tq);


   do {

       done = 1;
```

```
        for(int i = 0; i < n; i++) {
            if(rt[i] > 0) {
                done = 0;
                if(rt[i] > tq) {
                    time += tq;
                    rt[i] -= tq;
                } else {
                    time += rt[i];
                    wt[i] = time - bt[i];
                    rt[i] = 0;
                }
            }
        }
    } while(!done);
    for(int i = 0; i < n; i++)
        tat[i] = wt[i] + bt[i];
    printf("\nProcess\tBT\tWT\tTAT\n");
    for(int i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);
    return 0;
}
```

Output:



```
  nischal0x01 > ~/code/ku/comp-307/lab2_3 > main ?1  ........................... ✔ < 15:40:36
└ ./a.out
Enter number of processes: 4
Enter Burst Time for P1: 7
Enter Burst Time for P2: 5
Enter Burst Time for P3: 3
Enter Burst Time for P4: 7
Enter Time Quantum: 3

Process BT      WT      TAT
P1      7       14      21
P2      5       12      17
P3      3       6       9
P4      7       15      22
```

Github: https://www.github.com/nischal0x01/comp-307