Embedded Operating System Assignment 01

Dep. AI Convergence Engineering

Student ID: 2023214019

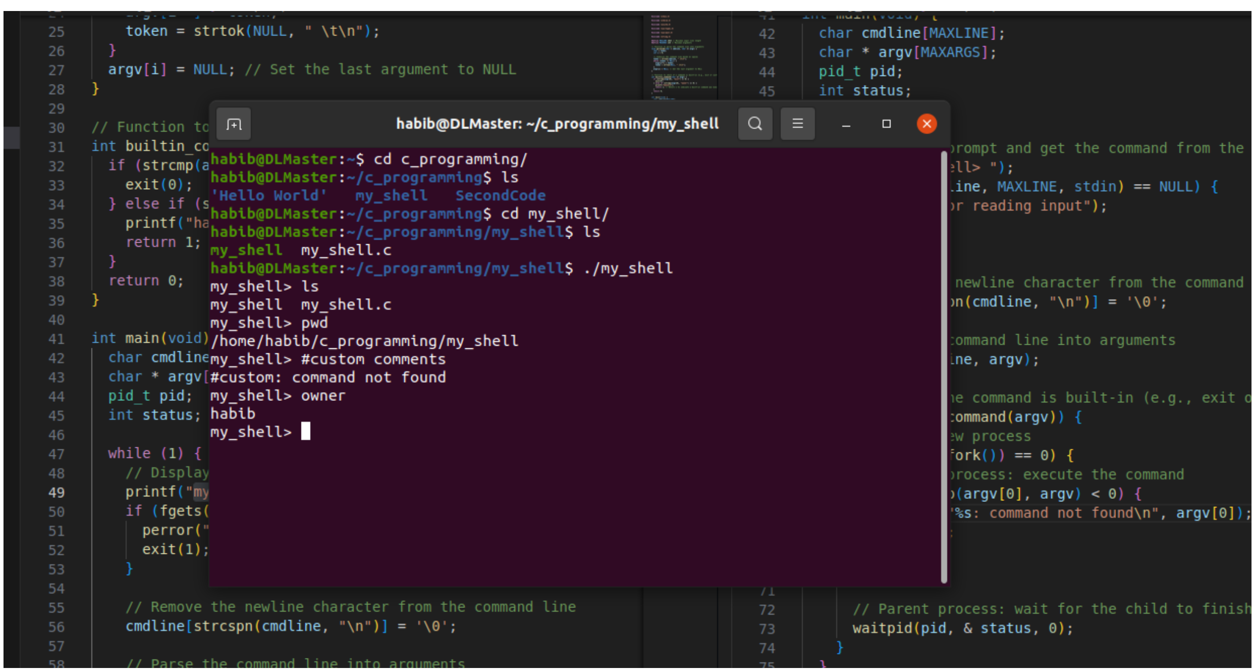
Name: Md Habibur Rahman

Code your own simplified shell program

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define MAXLINE 1024 // Maximum input line length
#define MAXARGS 128 // Maximum arguments

// Function to parse the command line into arguments
void parsecmd(char * cmdline, char ** argv) {
     char * token;
     int i = 0;

     // Tokenize the command line based on spaces
     token = strtok(cmdline, " \t\n");
     while (token != NULL) {
          argv[i++] = token;
          token = strtok(NULL, " \t\n");
     }
     argv[i] = NULL; // Set the last argument to NULL
}

// Function to check if a command is built-in (e.g., exit or custom
commands)
int builtin_command(char ** argv) {
     if (strcmp(argv[0], "exit") == 0) {
          exit(0);
     } else if (strcmp(argv[0], "owner") == 0) {
          printf("habib\n");
          return 1; // Return 1 to indicate a built-in command was executed
     }
     return 0;
}

int main(void) {
     char cmdline[MAXLINE];
     char * argv[MAXARGS];
     pid_t pid;
     int status;

     while (1) {
          // Display a prompt and get the command from the user
          printf("my_shell> ");
          if (fgets(cmdline, MAXLINE, stdin) == NULL) {
               perror("Error reading input");
               exit(1);
          }

          // Remove the newline character from the command line
          cmdline[strcspn(cmdline, "\n")] = '\0';
```

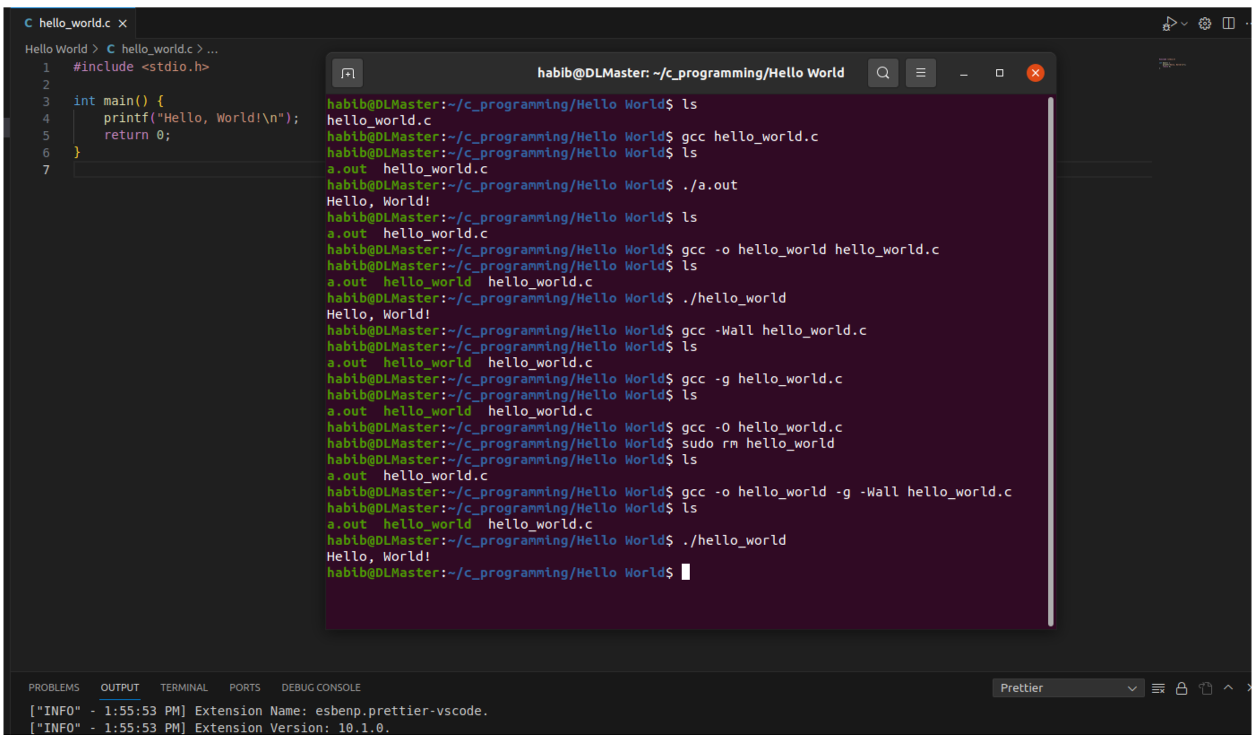
```
        // Parse the command line into arguments
        parsecmd(cmdline, argv);

        // Check if the command is built-in (e.g., exit or custom
commands)
        if (!builtin_command(argv)) {
            // Fork a new process
            if ((pid = fork()) == 0) {
                // Child process: execute the command
                if (execvp(argv[0], argv) < 0) {
                    printf("%s: command not found\n", argv[0]);
                    exit(0);
                }
            }

            // Parent process: wait for the child to finish
            waitpid(pid, & status, 0);
        }
    }

    return 0;
}
```

Output



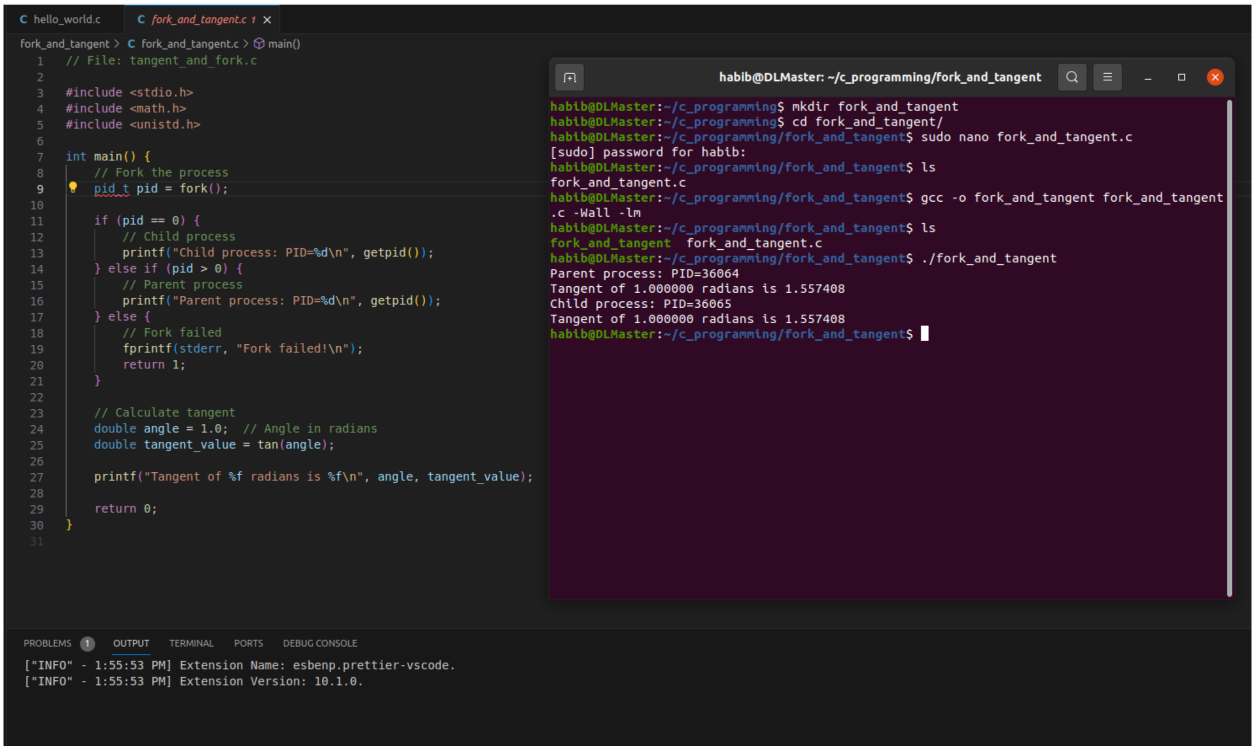
Read and code the Lab tutorial section

1. gcc hellow_world.c # Create a a.out file for execution of the code
2. ./a.out # Execute the code
3. gcc -o hw hw.c # -o: to specify the executable file name
4. gcc -Wall hw.c # -Wall: gives much better warnings
5. gcc -g hw.c # -g: to enable debugging with gdb
6. gcc -O hw.c # -O: to turn on optimization



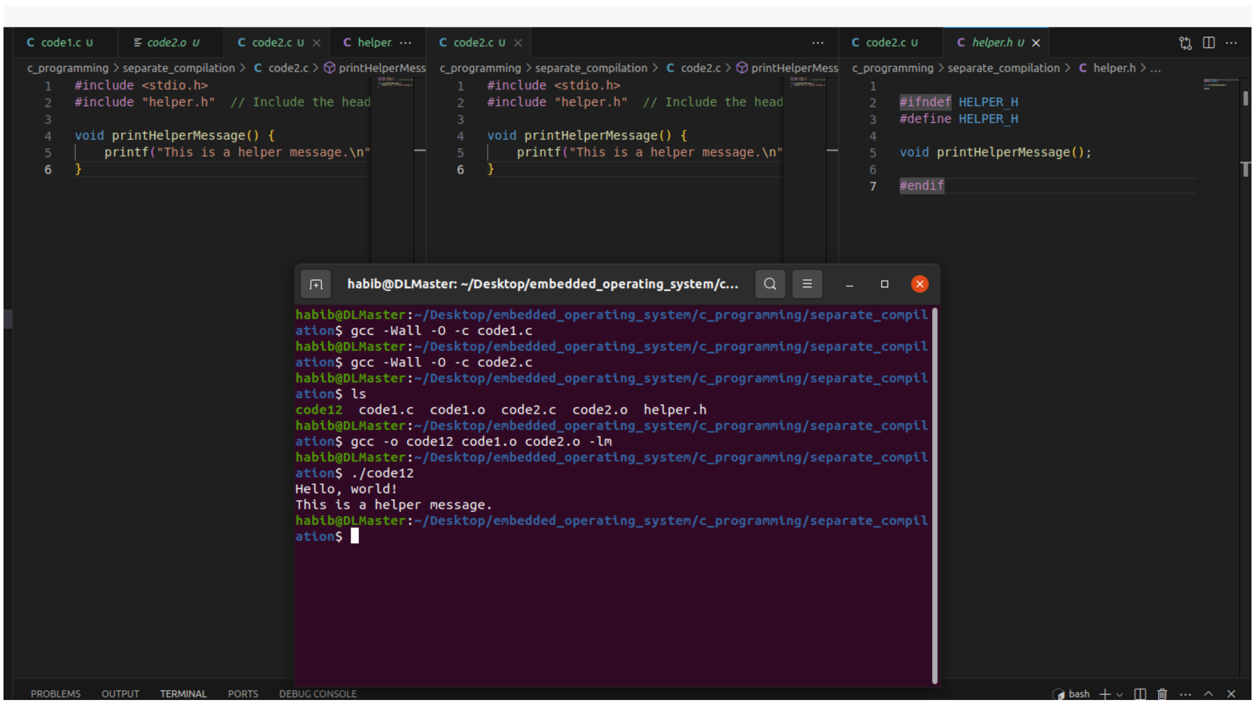
Linking with Libraries

1. gcc -o fork_and_tangent fork_and_tangent.c -Wall -lm # This will include the math library as header file and create a executable file
2. ./tangent_and_fork # This will execute the code.



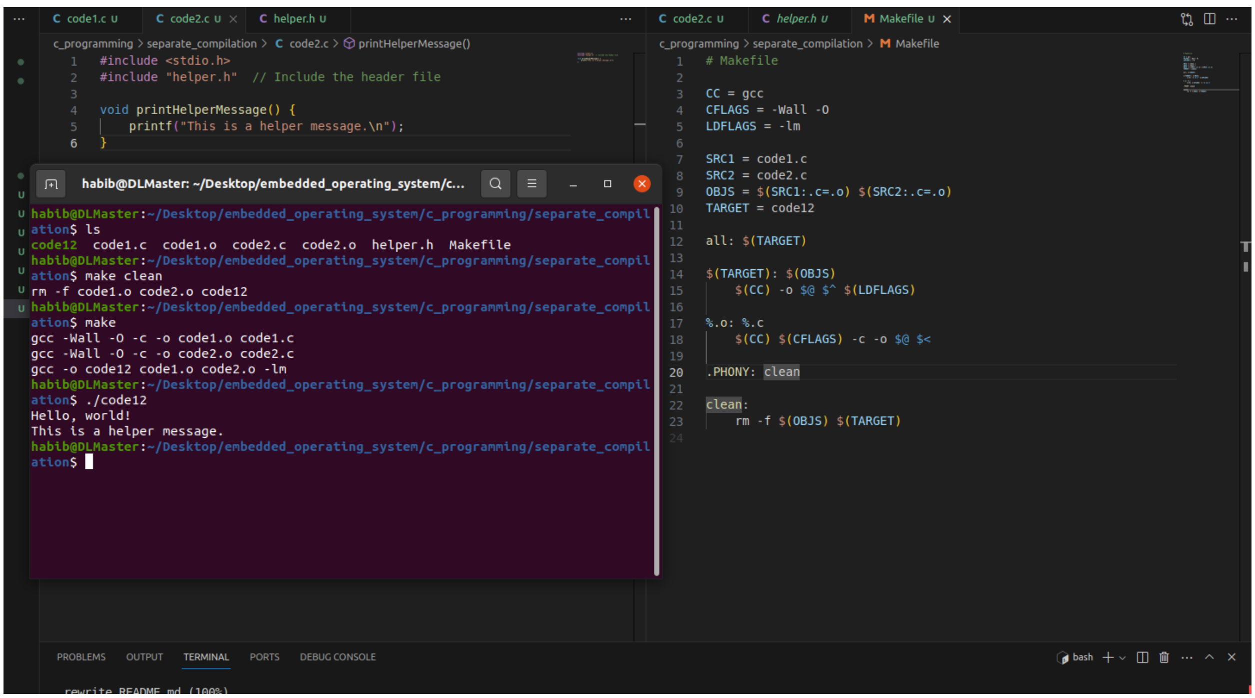
Separate Compilation

1. sudo nano helper.h # create a header file
2. gcc -Wall -O -c code1.c
3. gcc -Wall -O -c code2.c
4. gcc -o code12 code1.o code2.o -lm # This will make a execute file from two .c code
5. ./code12 # this will execute the combined file



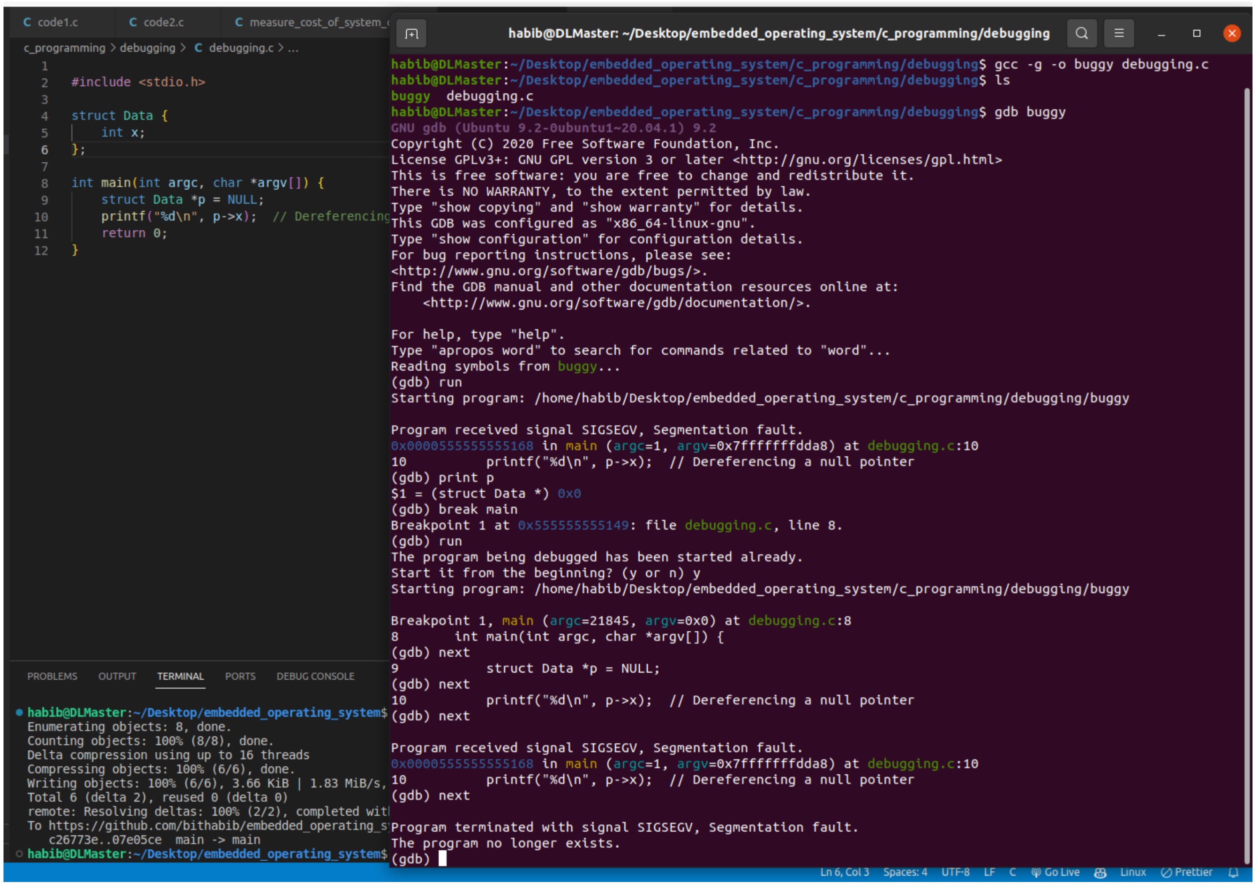
Makefiles

1. Sudo nano Makefile # create a make file
2. make # Build the executable using the Makefile
3. ./code12 # Run the executable
4. make clean # Clean up the generated files



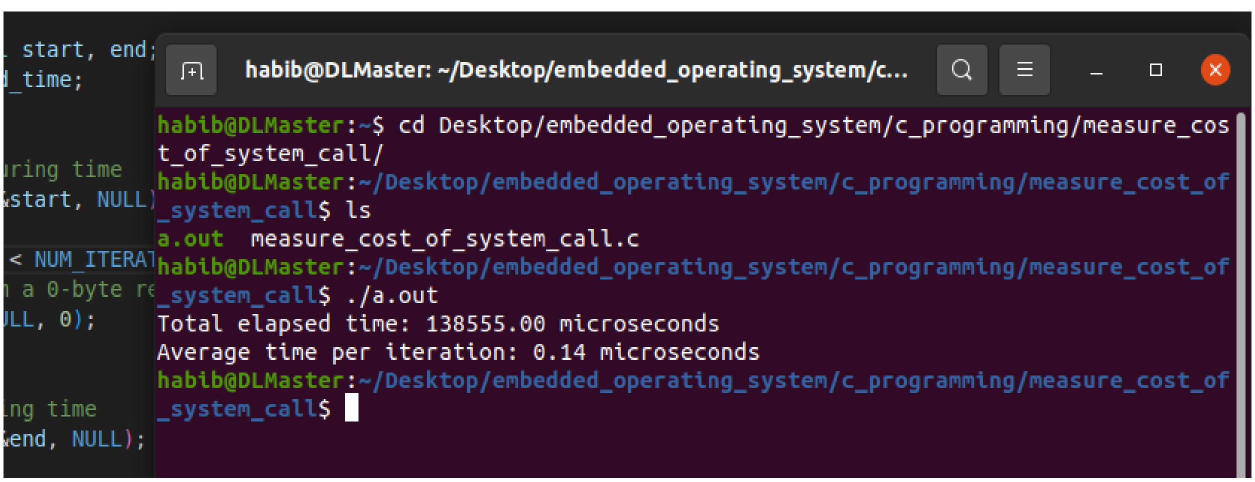
Debugging

1. gcc -g -o buggy buggy.c # compile this program with debugging information using the -g flag
2. gdb buggy # use gdb to debug the program and analyze the segmentation fault
3. (gdb) run # run the program and see a segmentation fault
4. (gdb) print p # see the segmentation fault
5. (gdb) break main # break main function
6. (gdb) run # run program to check step by step
7. (gdb) next # check the next line and go on you will encountered a segmentation fault



Measure costs of a system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

#define NUM_ITERATIONS 1000000

int main() {
     struct timeval start, end;
     double elapsed_time;
     int i;

     // Start measuring time
     gettimeofday(&start, NULL);

     for (i = 0; i < NUM_ITERATIONS; i++) {
          // Perform a 0-byte read as a simple system call
          read(0, NULL, 0);
     }

     // End measuring time
     gettimeofday(&end, NULL);

     // Calculate elapsed time in microseconds
     elapsed_time = (end.tv_sec - start.tv_sec) * 1000000.0;
     elapsed_time += (end.tv_usec - start.tv_usec);

     // Calculate average time per iteration
     double average_time_per_iteration = elapsed_time / NUM_ITERATIONS;

     printf("Total elapsed time: %.2f microseconds\n", elapsed_time);
     printf("Average time per iteration: %.2f microseconds\n",
average_time_per_iteration);

     return 0;
}
```



Calculate and explain the average turnaround time and response time

1. A and B need 50ms of CPU time each.
2. A runs for 10ms and then issues an I/O request
    a. I/Os each take 10ms
3. B simply uses the CPU for 50ms and performs no I/O
4. The scheduler runs A first, then B after

Answer:

1. **STCF Scheduling Algorithm:**
    a. Process A Turnaround Time (TAT) = Completion Time−Arrival Time
    = 90 - 0
    = 90
    b. Process B Turnaround Time (TAT) = Completion Time−Arrival Time
    = 140 - 0
    = 140

Average Turnaround Time = (90+140)/2
= 230/2
= 115

    c. Process A Response Time (RT) = the time it takes for a process to start executing once it's given access to the CPU
    = 0
    d. Process B Response Time (RT) = the time it takes for a process to start executing once it's given access to the CPU
    = 90

Average Response Time = (0+90)/2
= 90/2
= 45

2. **Round Robin Scheduling Algorithm:**
    a. Process A Turnaround Time (TAT) = Completion Time−Arrival Time
    = 90 - 0
    = 90
    b. Process B Turnaround Time (TAT) = Completion Time−Arrival Time
    = 100 - 0
    = 100

Average Turnaround Time = (90+100)/2
= 190/2
= 95

c. Process A Response Time (RT) = the time it takes for a process to start executing once it's given access to the CPU
   = 0
d. Process B Response Time (RT) = the time it takes for a process to start executing once it's given access to the CPU
   = 10

Average Response Time = (0+10)/2
= 10/2
= 5