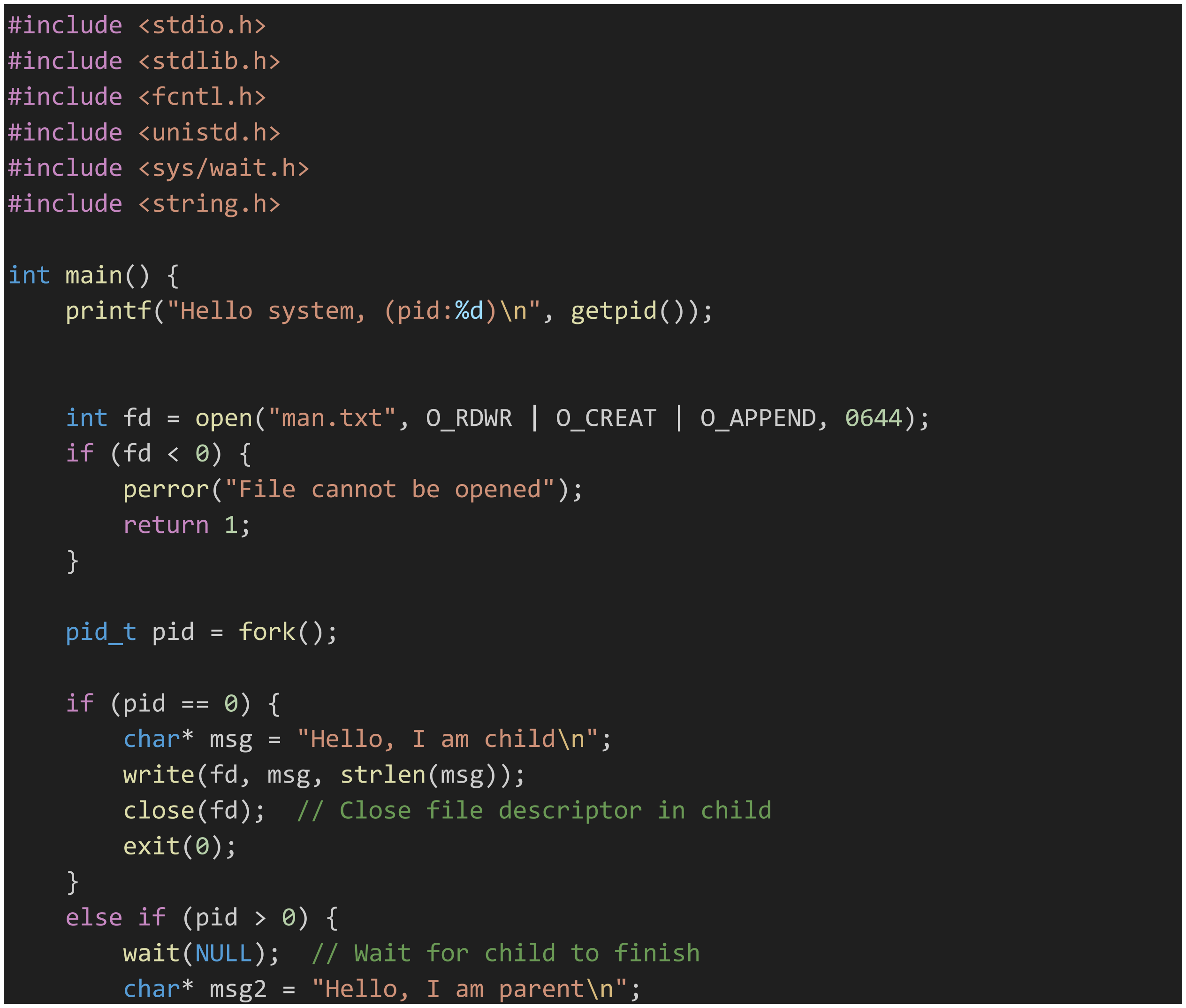
OPERATING SYSTEM

ASSIGNMENT # 1

Name: Manal Rana

Roll-Number: BSSE23099

Q#1

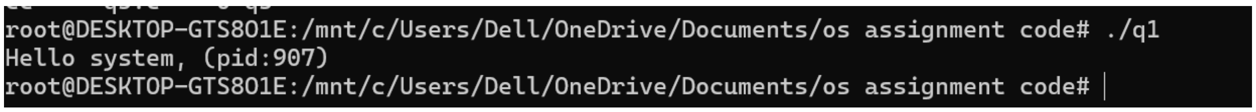


```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

int main() {
    printf("Hello system, (pid:%d)\n", getpid());


    int fd = open("man.txt", O_RDWR | O_CREAT | O_APPEND, 0644);
    if (fd < 0) {
        perror("File cannot be opened");
        return 1;
    }

    pid_t pid = fork();

    if (pid == 0) {
        char* msg = "Hello, I am child\n";
        write(fd, msg, strlen(msg));
        close(fd);  // Close file descriptor in child
        exit(0);
    }
    else if (pid > 0) {
        wait(NULL);  // Wait for child to finish
        char* msg2 = "Hello, I am parent\n";
        write(fd, msg2, strlen(msg2));
        close(fd);  // Close file descriptor in parent
    }
    else {
        perror("Fork failed");
    }

    return 0;
}
```

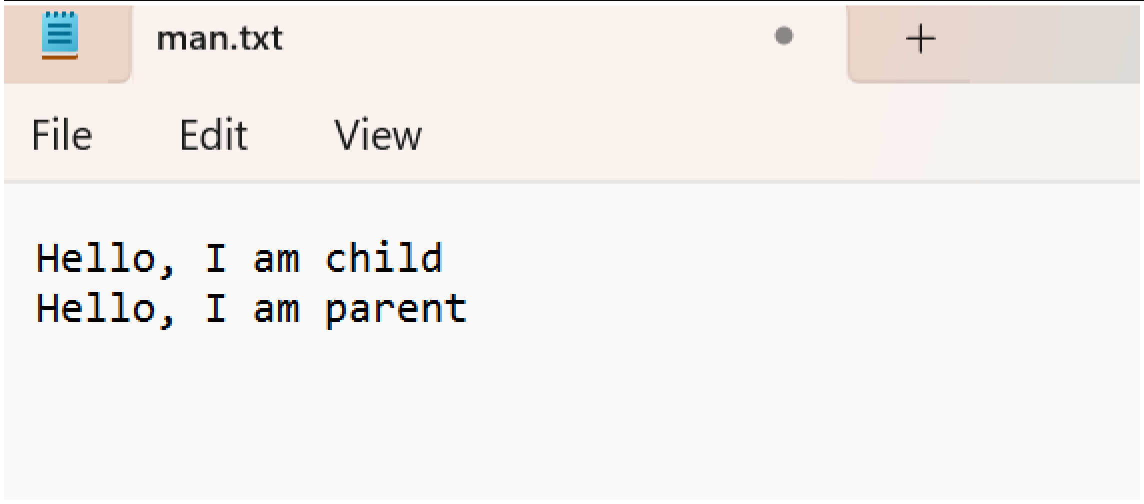
Explanation:

This program creates a child process using fork(), and both the parent and child write messages to the same file (man.txt). The child writes first, then the parent waits for the child to finish before writing its own message, ensuring ordered execution.

OUTPUT:



```
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q1
Hello system, (pid:907)
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code#
```



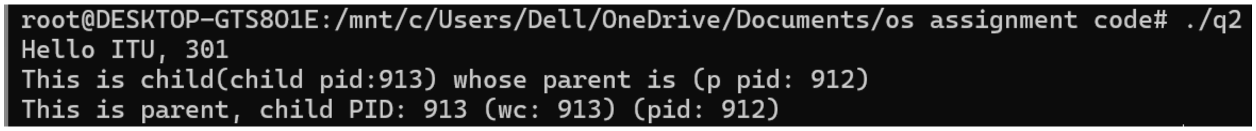
```
Hello, I am child
Hello, I am parent
```

Q#2:

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

int main() {
    int rc;


    printf("Hello ITU, %d\n", getppid());

    rc = fork();  // Fork a new child process

    if (rc < 0) {

        fprintf(stderr, "Fork failed\n");
        exit(1);
    }
    else if (rc == 0) {

        printf("This is child(child pid:%d) whose parent is (p pid: %d)\n",
getpid(), getppid());
    }
    else {

        int wc = wait(NULL);
        printf("This is parent, child PID: %d (wc: %d) (pid: %d)\n", rc, wc,
getpid());
    }

    return 0;
}
```

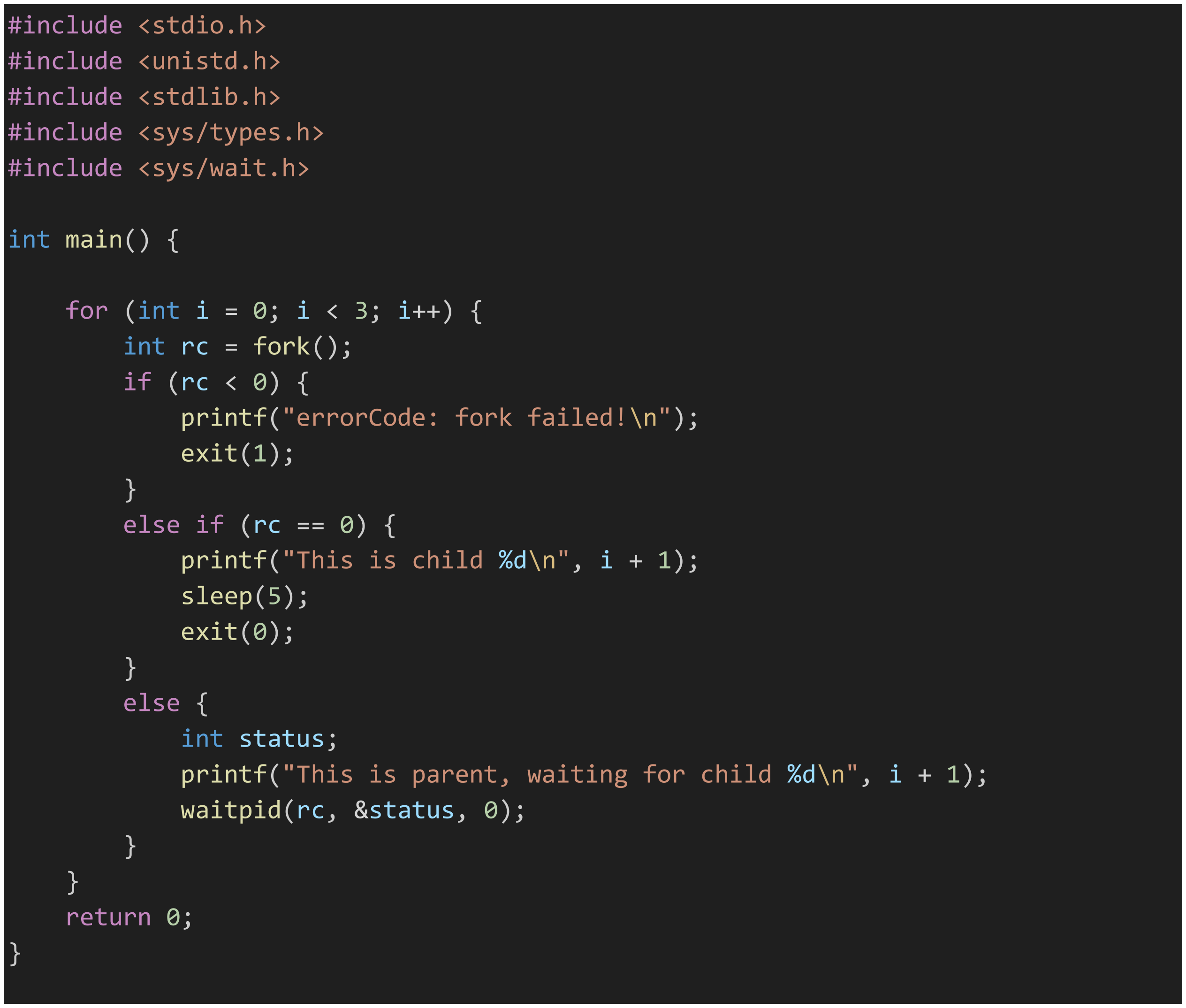
Explanation:
This program creates a child process using `fork()`, and both the parent and child print messages showing their process IDs (`PID`) and parent process IDs (`PPID`). The parent waits for the child to finish using `wait()`, ensuring the child runs first before the parent prints its message.

OUTPUT:



```
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q2
Hello ITU, 301
This is child(child pid:913) whose parent is (p pid: 912)
This is parent, child PID: 913 (wc: 913) (pid: 912)
```

Q#3:

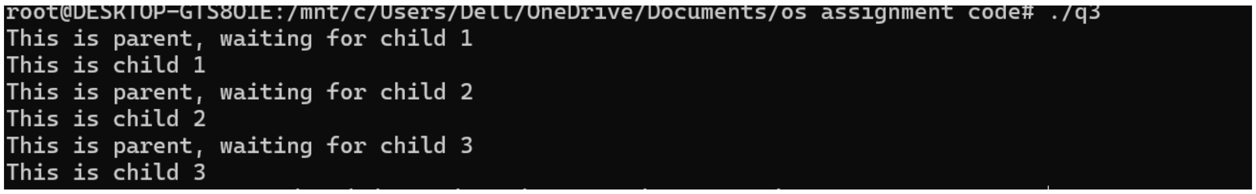


```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {

    for (int i = 0; i < 3; i++) {
        int rc = fork();
        if (rc < 0) {
            printf("errorCode: fork failed!\n");
            exit(1);
        }
        else if (rc == 0) {
            printf("This is child %d\n", i + 1);
            sleep(5);
            exit(0);
        }
        else {
            int status;
            printf("This is parent, waiting for child %d\n", i + 1);
            waitpid(rc, &status, 0);
        }
    }
    return 0;
}
```

Explanation:

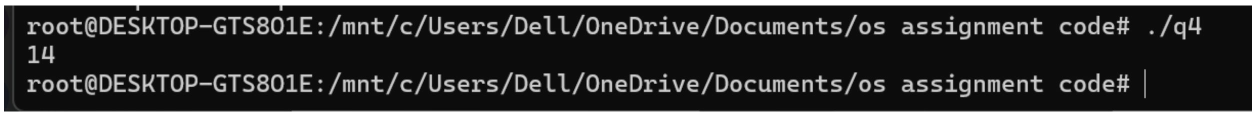
This program creates three child processes using a loop and `fork()`, where each child prints a message, sleeps for 5 seconds, and exits. The parent waits for each child to finish (`waitpid()`) before continuing to create the next child.

OUTPUT:



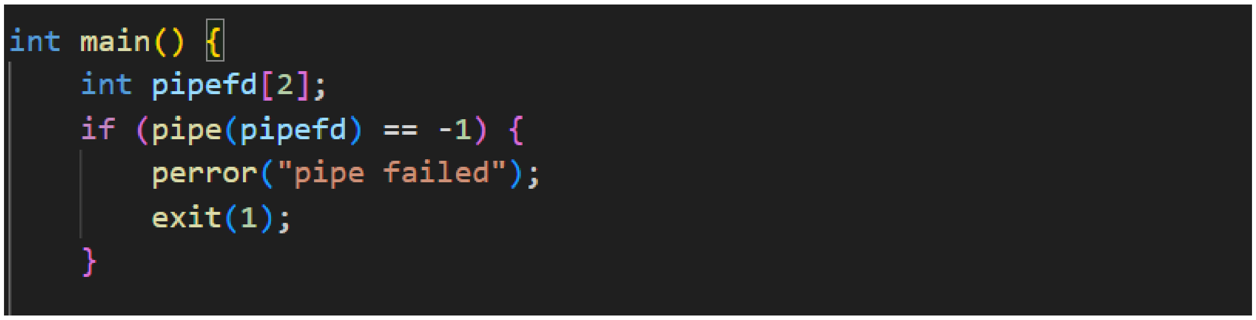
```
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q3
This is parent, waiting for child 1
This is child 1
This is parent, waiting for child 2
This is child 2
This is parent, waiting for child 3
This is child 3
```

Q#4:



```
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q4
14
root@DESKTOP-GTS801E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# 
```

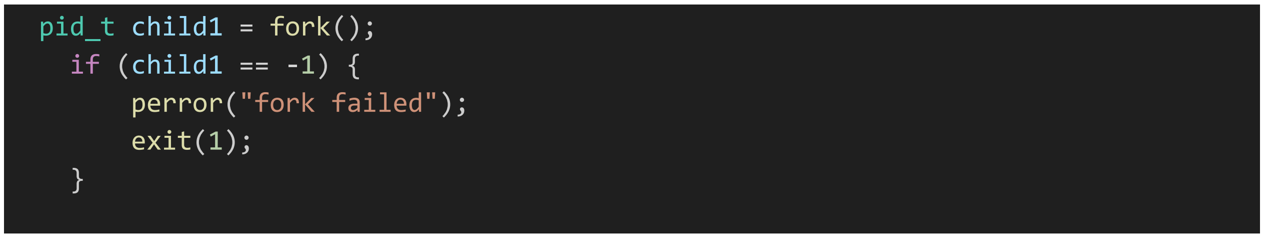
Explanation:



```c
int main() {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe failed");
        exit(1);
    }
```

pipe(pipefd) creates a pipe with pipefd[0] as the read end and pipefd[1] as the write end.

If pipe() fails, perror("pipe failed") prints an error, and exit(1) terminates the program.



```c
  pid_t child1 = fork();
    if (child1 == -1) {
        perror("fork failed");
        exit(1);
    }
```

fork() creates a new child process, and its PID is stored in child1.

If fork() fails (returns -1), an error is printed using perror(), and the program exits.

```
if (child1 == 0) {
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);

        execlp("echo", "echo", "Hello, world!", NULL);
        perror("execlp failed");
        exit(1);
    }
```

If child1 == 0, it means we are in the child process.

The child closes the unused read end (pipefd[0]), redirects stdout to the pipe's write end using dup2(), and then executes echo "Hello, world!", which means that this will not be printed on the terminal rather it will be written on the pipe .

```
  pid_t child2 = fork();

    if (child2 == -1) {
        perror("fork failed");
        exit(1);
    }
```

Now, the child process1 stos as the execlp is called but the parent process continues and the second fork() occurs.
fork() creates another child process, and its PID is stored in child2.

If fork() fails (returns -1), an error message is printed using perror(), and the program exits

```
  if (child2 == 0) {

        close(pipefd[1]);
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);

        execlp("wc", "wc", "-c", NULL);
        perror("execlp failed");
        exit(1);
```

```
    }
```

If child2 == 0, it means we are in the second child process.

The child closes the unused write end (pipefd[1]), redirects stdin to the pipe's read end using dup2(), and then executes wc -c (which counts characters).
If execlp() fails, an error is printed, and the child exits.
Now, we will see the output 14 that is the number of characters in the string written to the pip.
Now, each system program's stdout is by default connected to the terminal therefore the stdout of the command wc has not been redirected so we wil see the word count on the terminal.

```

   close(pipefd[0]);

    close(pipefd[1]);
    wait(NULL);
    wait(NULL);

    return 0;
```

The parent closes both ends of the pipe (pipefd[0] and pipefd[1]) since it no longer needs them.

wait(NULL) waits for the first child to finish, and the second wait(NULL) waits for the second child, ensuring both processes complete before the parent exits

Q#5:

a) part the cost of system call

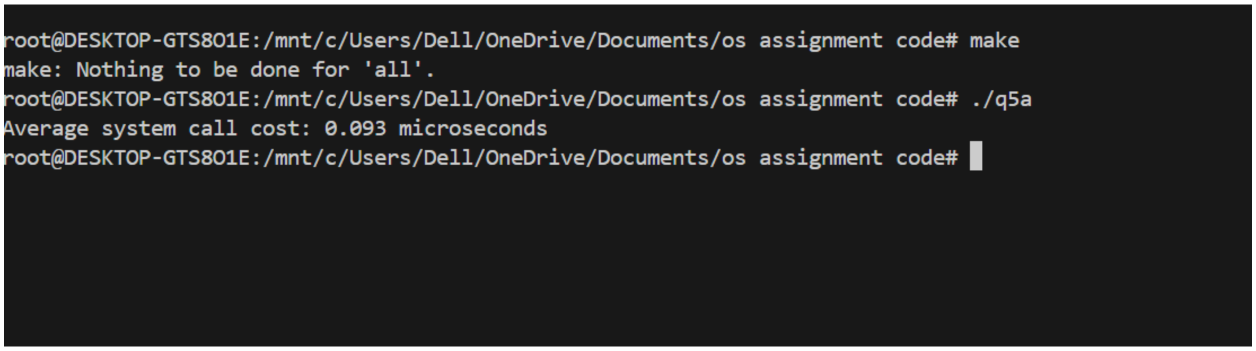
```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

#define ITERATIONS 1000000

int main() {
    struct timeval start, end;
    gettimeofday(&start, NULL);

    for (int i = 0; i < ITERATIONS; i++) {
        getpid();  //  system call
    }
```

```c
    gettimeofday(&end, NULL);

    double elapsed_time = (end.tv_sec - start.tv_sec) * 1e6 + (end.tv_usec -
start.tv_usec);
    printf("Average system call cost: %.3f microseconds\n", elapsed_time /
ITERATIONS);

    return 0;
}
```

Explanation:

This program measures the average time taken for a `getpid()` system call by repeatedly calling it one million times (`ITERATIONS`). It records the start and end times using `gettimeofday()`, calculates the total elapsed time in microseconds, and then divides it by the number of iterations to get the average system call cost.

OUTPUT:



```
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# make
make: Nothing to be done for 'all'.
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q5a
Average system call cost: 0.093 microseconds
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code#
```

b) cost for the context-switch

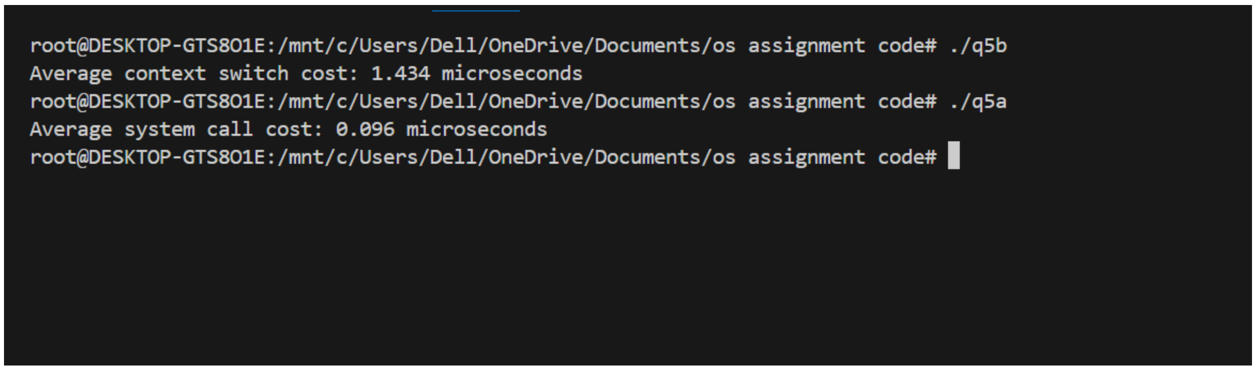
```c
#define _GNU_SOURCE

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

```
#define ITERATIONS 10000

void set_affinity() {
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(0, &set);  // Bind process to CPU 0
    sched_setaffinity(0, sizeof(cpu_set_t), &set);
}

int main() {
    int pipe1[2], pipe2[2];
    pipe(pipe1);
    pipe(pipe2);

    set_affinity();  // Bind parent process to a single CPU

    pid_t pid = fork();

    if (pid == 0) {
        set_affinity();  // Bind child process to the same CPU
        char buf;
        for (int i = 0; i < ITERATIONS; i++) {
            read(pipe1[0], &buf, 1);
            write(pipe2[1], &buf, 1);
        }
        exit(0);
    } else {
        struct timeval start, end;
        char buf = 'X';

        gettimeofday(&start, NULL);

        for (int i = 0; i < ITERATIONS; i++) {
            write(pipe1[1], &buf, 1);
            read(pipe2[0], &buf, 1);
        }

        gettimeofday(&end, NULL);

        double elapsed_time = (end.tv_sec - start.tv_sec) * 1e6 + (end.tv_usec -
start.tv_usec);
        printf("Average context switch cost: %.3f microseconds\n", elapsed_time /
(ITERATIONS * 2));

        return 0;
    }
}
```

Explanation:

This program measures the average cost of a context switch between a parent and child process using pipes for communication. It binds both processes to a single CPU using `sched_setaffinity()`, creates two pipes for bidirectional communication, and then times how long it takes to send and receive a message back and forth `ITERATIONS` times, calculating the average time per switch.

OUTPUT:



```
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q5b
Average context switch cost: 1.434 microseconds
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code# ./q5a
Average system call cost: 0.096 microseconds
root@DESKTOP-GTS8O1E:/mnt/c/Users/Dell/OneDrive/Documents/os assignment code#
```