Assignment 1

Operating system

Zunaira Abdul Aziz
BSSE23058
Section A

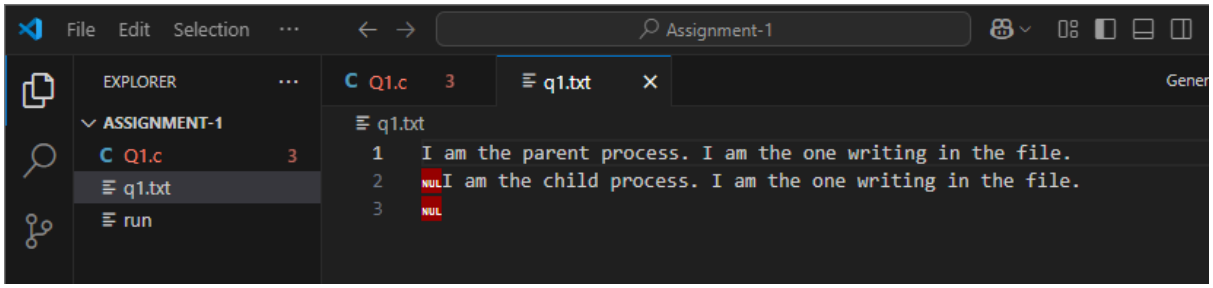
Contents

Q1.

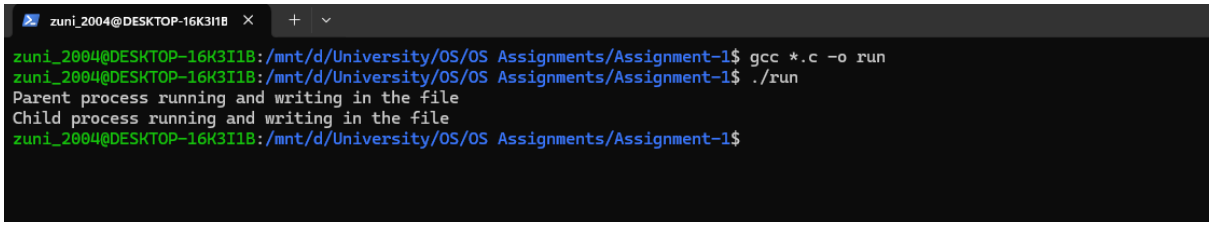
Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>    //for open()
#include <sys/wait.h> //for the wait()

int main(int argc, char *argv[])
{
   // Open the file with O_CREAT to create it if it doesn't exist, O_APPEND  for writing at the end of the file
   int fd = open("q1.txt", O_RDWR | O_CREAT | O_APPEND);
   // checks if file is opened or not, if system call value is -1 then it hasnt opened.
   if (fd < 0)
   {
      printf("Opening file failed, either it doesn't exist or hasn't opened\n");
      return -1;
   }

   int rc = fork();    // creates a new process of the pid
   if (rc < 0)
   {
      printf("Fork Failed\n");
      close(fd); //close the files before exiting
      return -1;
   }
   if (rc == 0)   // Child process
   {
      printf("Child process running and writing in the file\n");
      write(fd, "I am the child process. I am the one writing in the file.\n", 59);
      close(fd); //close the files before exiting
   }
   else   // Parent process
   {
      printf("Parent process running and writing in the file\n");
      write(fd, "I am the parent process. I am the one writing in the file.\n", 60);
      close(fd); //close the files before exiting
   }
   return 0;
}
```

Output in file:



Output on the terminal:



Answer:

Yes, both the parent and child processes share the same file descriptor. When fork () is called, the child process inherits a copy of the file descriptor table from the parent, meaning they both refer to the same open file description. Even if both the fork processes (child and parent) refer to the same file descriptor, that doesn't mean they write to the file at the same time because the processes run on at a time because writes do not interfere at the system call level. The order depends on the OS scheduler. If the parent executes first, then the parent's messages are written first and vice versa if the child executes first. If both processes write at the same time (or almost simultaneously), their output may be interleaved.

Q2.

Code:

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

int main(int argc, char *argv[])
{
   int pid = getpid(); // the pid of the current running program
   printf("PID is: %d\n", pid);
   int rc = fork(); // created copy of the process

   if (rc < 0)
   {
      printf("Fork Failed\n");
      exit(1);
   }

   if (rc == 0)
   { // Child process
      printf("I am child, pid: %d\n", getpid());
```

```
    sleep(2); // for simulating work in child
    printf("Child finished execution\n");
  }
  else
  {               // Parent process
    int wc = wait(NULL); // for child to finish its execution than parent runs
    printf("Parent here, My pid is %d, Wait system call value is %d\n", getpid(), wc);
  }
  return 0;
}
```

Output on terminal:

```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ gcc Q2.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ ./run
PID is: 1904
I am child, pid: 1905
Child finished execution
Parent here, My pid is 1904, Wait system call value is 1905
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$
```

If I use wait in child process:

The child has no child processes of its own.

Calling wait(NULL); inside the child will fail immediately and return -1.

The parent does not wait for the child because the child does not control the parent's execution.

Code:

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

int main(int argc, char *argv[])
{
  int pid = getpid(); // the pid of the current running program
  printf("PID is: %d\n", pid);
  int rc = fork(); // created copy of the process

  if (rc < 0)
  {
    printf("Fork Failed\n");
    exit(1);
  }

  if (rc == 0)
  { // Child process
    int wc = wait(NULL); // for child to finish its execution than parent runs
    printf("I am child, pid: %d\n", getpid());
    sleep(2); // for simulating work in child
    printf("Child finished execution\n");
  }
```

```
  else
  {               // Parent process
    printf("Parent here, My pid is %d\n", getpid());
  }
  return 0;
}
```

Output in terminal:

```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ gcc Q2.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ ./run
PID is: 1928
Parent here, My pid is 1928
I am child, pid: 1929
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ Child finished execution
```

Q3.

Code:

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

int main(int argc, char *argv[])
{
  int rc;          // to get the fork() system call values for the created new childs
  int pid = getpid(); // the pid of the current running program
  printf("PID is: %d\n", pid);
  for (int i = 1; i <= 3; i++) // for creating three kids
  {
    rc = fork(); // created copy of the parent
    if (rc < 0)
    {
      printf("Fork Failed\n");
      exit(1);
    }
    if (rc == 0)
    {          // Child process
      sleep(i); // for simulating work in child //sleeps 1s more than the previous
      printf("I am child %d, My pid: %d\n", i, getpid());
      exit(0); // exits successfully indicating work is done with no error
    }
  }
  for (int i = i; i <= 3; i++)
  {
    printf("This is parent, waiting for child %d\n", i);
    waitpid(-1, NULL, 0); // waitimg for any child to finish
  }
```

```
    printf("All three children finished working. Parent exiting.\n");
    return 0;
}
```

Output in terminal:

```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ gcc Q3.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ ./run
PID is: 2003
This is parent, waiting for child 1
I am child 1, My pid: 2004
This is parent, waiting for child 2
I am child 2, My pid: 2005
This is parent, waiting for child 3
I am child 3, My pid: 2006
All three children finished working. Parent exiting.
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$
```

Q4.

Output of code:

```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ gcc Q4.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ ./run
14
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$
```

- Create a Pipe: pipe(pipefd) creates two endpoints: one for reading (pipefd[0]) and one for writing (pipefd[1]).
- First Child Process (child1):
    - Forks a child process.
    - Closes the read end of the pipe.
    - Redirects its output to the write end: dup2(pipefd[1], STDOUT_FILENO).
    - Executes execlp("echo", "echo", "Hello, world!", NULL), writing "Hello, world!" into the pipe.
- Second Child Process (child2):
    - Forks another child.
    - Closes the write end of the pipe.
    - Redirects its input to the read end: dup2(pipefd[0], STDIN_FILENO).
    - Executes execlp("wc", "wc", "-c", NULL) to count characters from the pipe.
- Parent Process:
- Closes both ends of the pipe.
- Waits for both children to finish.

The first child writes "Hello, world!" (14 characters) to the pipe, and the second child counts those characters, returning 14.

Q5:

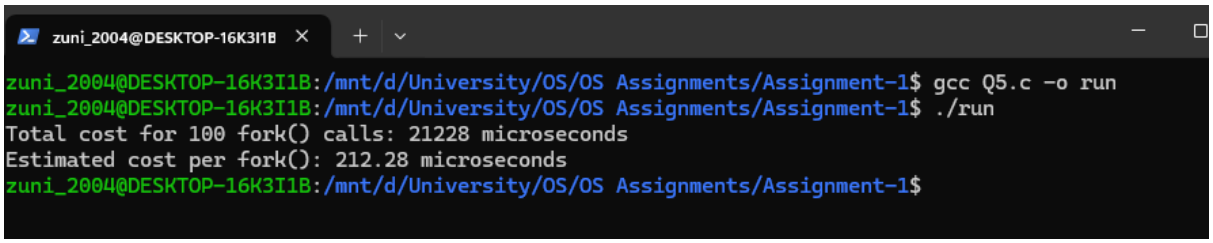
Code for cost of System call (fork()):

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

```c
int main()
{
    struct timeval start;    // To store start time
    struct timeval end;      // To store end time
    long total_fork_time = 0; // To store total time for fork() calls

    // Measure timer overhead (cost of calling gettimeofday() multiple times)
    long avg_timer_overhead = 0;
    for (int i = 0; i < 1000000; i++) // Repeat 1,000,000 times for accuracy
    {
        gettimeofday(&start, NULL); // Start time
        gettimeofday(&end, NULL);   // endd time
        // Calculate time taken by gettimeofday() itself
        avg_timer_overhead = avg_timer_overhead + ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec));
    }
    avg_timer_overhead = avg_timer_overhead / 1000000; // Get average overhead

    // Measure fork() cost
    gettimeofday(&start, NULL);   // Start time before creating child processes
    for (int i = 0; i < 100; i++) // Loop to create 100 child processes
    {
        int rc = fork();
        if (rc < 0)
        {
            _exit(1); // Exit the program with an error code
        }
        if (rc == 0) // Child process
        {
            _exit(0); // Child exits immediately
        }
        else if (rc > 0) // parent process
        {
            wait(NULL); // waiting for child to finish
        }
    }
    gettimeofday(&end, NULL); // time end

    // Calculate total time taken for all fork() calls
    total_fork_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    // Subtract timer overhead for more accurate fork() measurement
    total_fork_time -= avg_timer_overhead * 100;

    // Estimate cost per fork()
    double cost_per_fork = (double)total_fork_time / 100;
    // Calculate and display total cost and cost per fork()
    printf("Total cost for 100 fork() calls: %ld microseconds\n", total_fork_time);
    printf("Estimated cost per fork(): %.2f microseconds\n", cost_per_fork);
```

```
    return 0;
}
```

Output:



Code for cost of context switch:

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

int main()
{
    int pipe0[2]; // Pipe for communication from parent to child
    int pipe1[2]; // Pipe for communication from child to parent
    struct timeval start;
    struct timeval end; // Variables to store start and end times
    int rc;

    // Create pipes
    if (pipe(pipe0) == -1 || pipe(pipe1) == -1)
    {
        printf("pipe failed");
        exit(1);
    }
    // Giving me error so i am unable to use the CPU Affinity so I commented out
    // cpu_set_t mask;
    // int cpu = 0; // Set affinity to CPU 0
    // CPU_ZERO(&mask);
    // CPU_SET(cpu, &mask);
    // sched_setaffinity(0, sizeof(mask), &mask); // Bind to CPU 0

    // Create child process
    rc = fork();
    if (rc < 0)
    {
        printf("fork failed");
        exit(1); // exit woth error
    }
```

```
if (rc == 0)
{ // Child process
    // Close unused pipe ends
    close(pipe0[1]); // Close write end of pipe0
    close(pipe1[0]); // Close read end of pipe1

    char buf;
    gettimeofday(&start, NULL); // Start time for child process

    for (int i = 0; i < 1000000; i++)
    {
        // Write to pipe0
        write(pipe0[1], "x", 1);
        // Read from pipe1
        read(pipe1[0], &buf, 1);
    }

    gettimeofday(&end, NULL); // End time for child process
    close(pipe0[0]);        // Close read end of pipe0
    close(pipe1[1]);        // Close write end of pipe1

    // Calculate total time taken for context switches
    long total_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    printf("Child total time for %d iterations: %ld microseconds\n", 1000000, total_time);
    exit(0); // Exit child process
}
else
{ // Parent process
    // Close unused pipe ends
    close(pipe0[0]); // Close read end of pipe0
    close(pipe1[1]); // Close write end of pipe1

    char buf;
    gettimeofday(&start, NULL); // Start time for parent process

    for (int i = 0; i < 1000000; i++)
    {
        // Read from pipe0
        read(pipe0[0], &buf, 1);
        // Write to pipe1
        write(pipe1[1], "x", 1);
    }

    gettimeofday(&end, NULL); // End time for parent process
    close(pipe0[1]);        // Close write end of pipe0
    close(pipe1[0]);        // Close read end of pipe1

    // Wait for the child process to finish
```

```
    wait(NULL);

    // Calculate total time taken for context switches
    long total_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    printf("Parent total time for %d iterations: %ld microseconds\n", 1000000, total_time);
  }

  return 0; // Exit successfully
}
```

Output:

```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ gcc Q5a.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$ ./run
Child total time for 1000000 iterations: 142011 microseconds
Parent total time for 1000000 iterations: 139741 microseconds
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-1$
```